

ARDUINO MODBUS SLAVE

Ver. 0.9

Written by: T. W. Shilling
Written On: 6 MAR 2012

Contents

PREFACE	5
DISCLAIMER.....	6
1. MODBUS PROTOCOL.....	7
1.1. Slave Address.....	7
1.2. Function	7
1.3. Address	8
1.4. Data.....	8
1.5. CRC.....	8
2. Hardware	9
2.1. Pins.....	9
2.1.1. Digital I/Os	9
2.1.2. Analog Inputs.....	10
2.1.3. Analog Outputs (PWM)	10
3. Arduino Sketch and Required Libraries	11
3.1. Setting up your Computer Environment	11
3.2. Programming Your Board	11
4. Customization and Settings	13
4.1. Digital Inputs and Outputs.....	13
4.2. Analog Outputs (PWM)	13
4.3. Analog Inputs.....	13
4.4. Kill Registry	14
4.5. Serial Port	15
5. Registers.....	16
5.1. Digital I/O – Register 0-4.....	16
5.2. PWM Value – Register 10-22.....	17
5.3. Analog Values – Register 30-45	17
5.4. Timer – Register 50.....	17
5.5. Config I/O – Register 60-64	17
5.6. Kill I/O - Register 70-74.....	18
5.7. PWM Enable – Register 80	18
5.8. Analog Input – Register 81	18
5.9. Config – Register 90.....	18

5.10. Error – Register 91	18
6. Mach 3 Interface	19
6.1. Getting Started	19
6.2. Setting Up your Inputs and Outputs.....	21
6.3. Brains	22
6.4. A Higher Level Brain	25
7. Conclusion	28
APPENDIX A – ModBusSlave.pde	28
APPENDIX B – Pin_Manipulator.pde.....	32
APPENDIX C – CRC.c	39
APPENDIX D – CRC.h	40
APPENDIX E – Modbus_Slave.c.....	42
APPENDIX F – Modbus_Slave.h	49

Figures

Figure 1 - Arduino Hardware	9
Figure 2 - Modbus Folder	11
Figure 3 - Arduino Library Folder	11
Figure 4 - Board Selection	12
Figure 5 - Serial Port Selection	12
Figure 6 - Upload	12
Figure 7 - IO Config Register	13
Figure 8 - Analog Config	13
Figure 9 - Kill Time	14
Figure 10 - Kill IO Register	14
Figure 11 – Baud rate	15
Figure 12 - ModBus Configuration Screen	19
Figure 13 - Devices and Printers	20
Figure 14 - ModBus Test	20
Figure 15 - ModBus Configuration	21
Figure 16 - Brain Editor	22
Figure 17 - Brain Editor, New	22
Figure 18 - Flash Brain, Adding an Input	22
Figure 19 - Flash Brain, Input	22
Figure 20 - Flash Brain, Timer Input Operation	23
Figure 21 - Flash Brain, Operation	23
Figure 22 - Flash Control	24
Figure 23 - Brain Control	24
Figure 24 - Brain Control - Enable	24
Figure 25 - Spindle Flash, Spindle On Input	25
Figure 26 - Spindle Flash, Spindle On	25
Figure 27 - Spindle Flash, AND Operation	26
Figure 28 - Spindle Flash, LED Out	26
Figure 29 - Spindle Flash, Spindle Off	27
Figure 30 - Spindle Flash Brain	27

Tables

Table 1 - Modbus Data Structure	7
Table 2 - Supported ModBus Functions	8
Table 3 - Pin Types	9
Table 4 - Register Map	16
Table 5 - Timer Bit Field	17
Table 6 - Error Codes	18

PREFACE

Modbus was originally developed in the 1970s and has since become one of the most widely used industrial communications control interfaces. A robust yet simple protocol allows for the efficient communication of settings and states between host and slave devices. Modbus is designed around a single host system controlling multiple slaves by manipulating their coils (1bit) and registers (16bit). Specifically for this document its use as an interface expansion solution for Mach 3 CNC controller software will be discussed along with how to customize the Arduino sketch and hardware to suite your individual CNC setup. The focus was to create an easy to setup and use system that relied on an established protocol and an open source, readily available hardware platform (i.e. Arduino).

DISCLAIMER

This software is provided as is and I assume no responsibility for any outcome of its use. All information within and referred to is released to the public domain with the express wish that references to Shilling Systems are left intact and the LOGO not removed.

Copyright (c) 2012 Tim W. Shilling (www.ShillingSystems.com)

Arduino Modbus Slave 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.

Arduino Modbus Slave 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.

To get a copy of the GNU General Public License see <<http://www.gnu.org/licenses/>>.

1. MODBUS PROTOCOL

The Modbus protocol is implemented with a simple query and reply scheme. The Host makes a request and the Slave responds accordingly. The Slave will never respond unless it is first queried. This means that all flow control is conducted by the Host. The protocol can utilize either RS232 or RS485 but since the Arduino board only supports RS232 via a USB to serial converter, this is the protocol implemented. Additionally the standard allows for Binary and ASCII transmission of data. The Binary version is the only one used in this system and ASCII is not supported in any form, this conforms with the target Host system. Data is transmitted in packets, usually constructed as follows:

Table 1 - Modbus Data Structure

Byte	Data
0	Slave Address (0-255)
1	Function
2	Address High
3	Address Low
4 -> X	Some Data...
X+1	CRC High
X+2	CRC Low

The data is then processed and the appropriate response given. If an error occurs, the data is dropped and no action is performed or a response transmitted. The only indication of an error will be via the Error Status LED located on PIN 13 of an Arduino board. A flashing LED indicates an error in transmission while a steady LED indicates a Lost communications status.

1.1. Slave Address

Multiple device can be located on the same serial bus. Meaning, all Slave devices can have their TX and RX pins connected to each other in parallel so that they all share the same serial port. This allows the Host to communicate to up to 247 devices with a single serial port. To differentiate which Slave the Host intends to communicate with, the very first byte sent is the Slave Address. Allowable values are 0-247. Zero holds special meaning however, it is a broadcast message. All devices will react to a broadcast query / command, but none will respond. This could be used to communicate a common setting or state to all devices at once.

1.2. Function

The Modbus protocol supports a wide range of functions. Functions tell the system what to do with the data it is presented. This implementation is limited to only nine as

they are the nine implemented by the target Host system and provide most, if not all the required functionality. This is not all of the functions supported by the Modbus protocol but are the only ones implemented in this system:

Table 2 - Supported ModBus Functions

Function	Description	Data Type
1	Read Coil Status	1 bit Coil
2	Read Input Status	1 bit Coil
3	Read Holding Registers	16 bit Register
4	Read Input Registers	16 bit Register
5	Force Single Coil	1 bit Coil
6	Preset Single Register	16 bit Register
7	Read Exception	8 bit Coil
15	Force Multiple Coils	1 bit Coil
16	Preset Multiple Registers	16 bit Register

1.3. Address

Internal to the Modbus slave, all data is stored in 16bit registers. Within each register, the 16bits are referred to as coils. What this address is refereeing to, weather register or coil depends on what function is being executed. If the address is that of a register, it would refer to the location of a 16bit word of data, counting from zero at 16bit increments. If the address is referring to a coil, it is the number of bits counting from zero to either read or write at. For instance, the 2nd register address would simply be 2, but if you wanted the 1st bit in the 2nd registry, you would address the 16th coil. Valid addresses are dependent on the Arduino implementation but are currently 0-99 registers. In the event of an invalid address an error will be set and no response will be given.

1.4. Data

The 16 bit data is organized into 2 bytes. The first byte is the high order byte and the second, the low.

1.5. CRC

Each message, query and response is ended with a checksum value (CRC). This is a 16 bit value computed by the Host and the Slave independently based on the data being exchanged. If this value does not match, the message is assumed to have errors and is discarded. An error will be set and no response will be given.

2. Hardware

There are four different types of Pins available. Digital output, digital input, analog output (PWM) and analog input. The first two, digital input and output are simply on or off, 1 or 0. The other two can have a range of values (Table 3 - Pin Types.). The remainder of the hardware is specific to the Arduino board being used and the user is encouraged to reference the appropriate documentation.

Table 3 - Pin Types

Pin Type	Allowable Values	Voltage Range
Digital Output	0, 1	0, 5
Digital Input	0, 1	0, 5
Analog Output (PWM)	0 – 255	0 – 5V
Analog Input	0 – 1024	0 – 5V

2.1.Pins

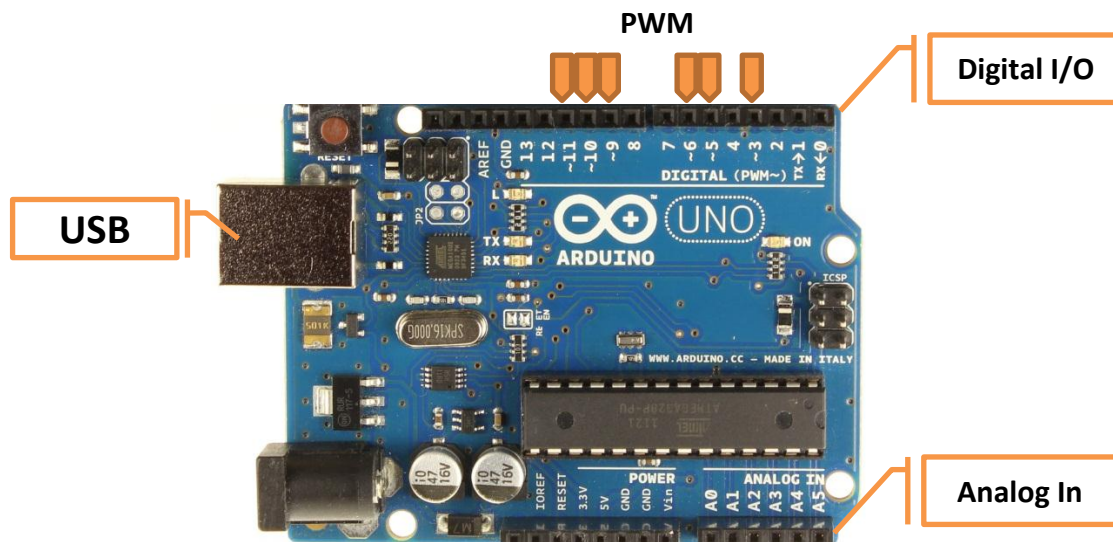


Figure 1 - Arduino Hardware

2.1.1. Digital I/Os

Digital Outputs are either on or off, 1 or 0. These can be used for lighting an LED, sounding a buzzer, or turning on a spindle through a relay if desired. It is important to realize that you cannot directly drive too large of a load from these. The limit is 40mA per pin, and 200mA for all pins. Keep this in mind when designing your system, you may need to use transistors or some other means of driving higher current items. By default, pins 2 – 12 are configured as digital outputs. Digital inputs are automatically pulled to high by an internal 20kohm resistor. This keeps the input from floating, so no external pull up resistor is required. This means that if your input is open, the input value will be 1 and

when the input is closed it will be 0. This function can be turned off, but requires greater knowledge of the Arduino language to change.

WARNING: DO NOT put a pull down resistor on any input unless you really know what you are doing.

2.1.2. Analog Inputs

The Arduino board has several analog to digital converter pins available. 6 on the UNO and 16 on the MEGA. These pins are by default analog inputs but can be configured as digital inputs or outputs. These pins will except 0-5V and convert the measured voltage to a value of 0 – 1024. This value can be used to set feed rates, spindle speeds or any other variable value. Additionally, in analog mode if the voltage is above 2.5 volts, the corresponding digital I/O bit will be set to 1, and 0 if less than 2.5 volts. It should be noted that in analog mode, there is no pull up resistor. This resistor is only activated if the pin is switched to a digital input.

2.1.3. Analog Outputs (PWM)

Analog output is facilitated through PWM (Pulse Width Modulation). PWM means that the pin is turned on and off very quickly at a rate of ~500hz. The PWM engine generates a square wave on the output pin. Based on how long the square wave is high (5V) vs. low (0V) is called the duty cycle. A Duty cycle of 0 would correspond with the pin being off all the time and 0 volts. If a Duty cycle is set to 255, then the pin is on all the time, 5V. Anywhere in between and the pin is on for some time and off for the rest. This can be used to drive a PWM motor controller or an analog output. If it were connected to an LED, you could adjust the brightness of the LED by changing the PWM duty cycle. PWM is referred to as an analog output because the voltage averages over time, so even though the pin is really just turning on and off, to most systems it will appear to be at some intermediate voltage. On the smaller Arduino boards, such as the UNO, pins 3, 5, 6, 9, 10 and 11 are available for PWM. These pins, by default are digital outputs unless otherwise specifically configured as PWM pins (See Configuration section for more info). On the Arduino Mega, it works on pins 2 through 13. It should be noted that the PWM outputs generated on pins 5 and 6 will have higher-than-expected duty cycles. This will be noticed mostly on low duty-cycle settings (e.g 0 - 10) and could result in a value of 0 not fully turning off the output on pins 5 and 6.

3. Arduino Sketch and Required Libraries

The Modbus slave program for Arduino consists of several parts. First, is the main Sketch and the second is the required Modbus Slave library. If you are unfamiliar with what either is, please reference the Arduino homepage <http://www.arduino.cc>. In order to customize your Slave's properties or to program your own Arduino board, you must have the Arduino editor installed. If you desire to do neither and have a board that was already programmed, you can skip this section entirely.

3.1. Setting up your Computer Environment

First, you are going to need the Arduino Integrated Development Environment (IDE). You can download the most recent version from <http://www.arduino.cc>. If you haven't done this, do it now, before continuing. It comes as a zip file and does not require any install. Simply unzip the download to your desired location. Perhaps C:\Arduino\. Now, download the zip file for the ModbusSlave. In this file you will find two folders, "Firmware" and "ModBusSlave" (Figure 2 - Modbus Folder).

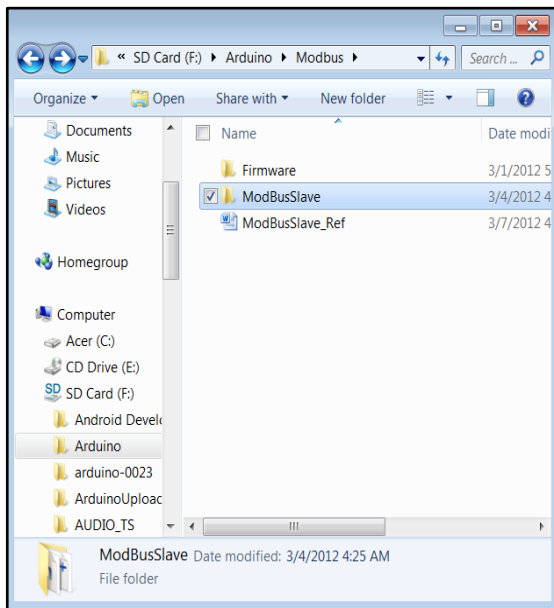


Figure 2 - Modbus Folder

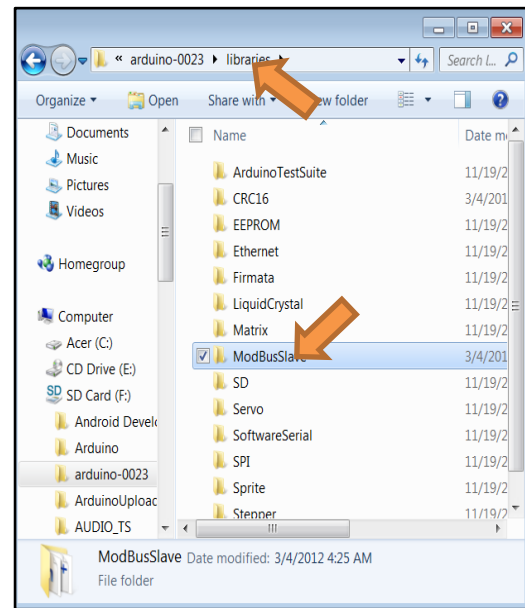



Figure 3 - Arduino Library Folder

Copy the "ModBusSlave" Folder and now navigate to your Arduino folder. Find the Libraries folder and open it up. Paste the "ModBusSlave" folder here (Figure 3 - Arduino Library Folder). That's it, pretty easy huh?

3.2. Programming Your Board

Pull out your Arduino board. Connect it to your computer via a USB cable. Open the Arduino IDE by going to your Arduino Folder and selecting the  arduino icon. Once loaded, open the ModBus program. It is located in the download under the Firmware folder, labeled ModBusSlave.pde, which is an Arduino sketch file. Open it and take a look. This is where you can edit the parameters of the ModBusSlave, but for now, lets

SHILLING SYSTEMS

just program our board and move on. With your board connected, allow windows to recognize it as a serial port. From the Tools dropdown menu, select your board. This Modbus slave was designed around the UNO and MEGA boards, so we will select the UNO (Figure 2.3). Next, make sure the correct serial port is selected (Figure 2.4).

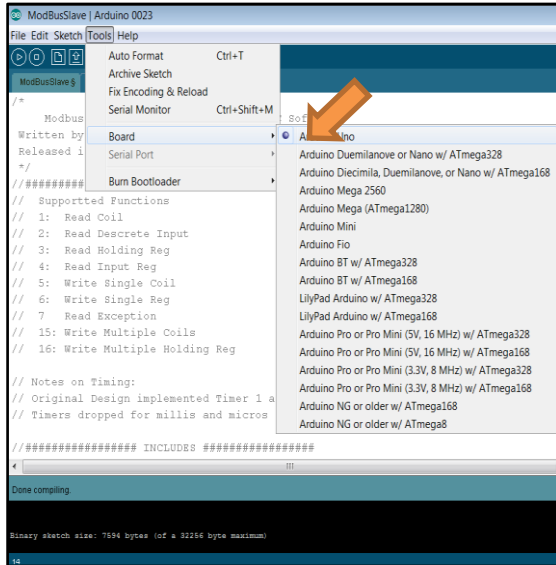


Figure 4 - Board Selection

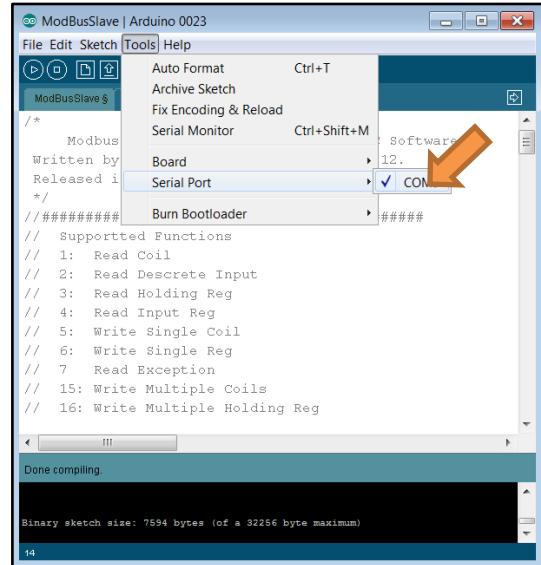


Figure 5 - Serial Port Selection

From here, it is as simple as pressing the upload button. Arduino should take care of the rest and your board will be ready to go (Uploading may take a minute or two, be patient).



Figure 6 - Upload

4. Customization and Settings

Because you have direct access to the source program, it is easy to customize your ports and pin behavior. You can go as far as completely rewriting what each pin does if you like, but for the scope of this document we will focus on pin configuration.

4.1. Digital Inputs and Outputs

Whether a pin is an Input or an Output is controlled by the IO_Config_Register (Figure 7 - IO Config Register). The registers are setup with Pin 0 being on the right and counting up to the left. Set the corresponding bit to a 1 for an input and 0 for an output. If you are using a smaller board, you will only need to touch the first IO_Config_Register. The next 3 are for larger boards and the last one (All 1's in the example) is for setting whether the A0-A16 pins are analog inputs or digital I/Os. Set these to 1 for analog input and 0 for digital I/O,

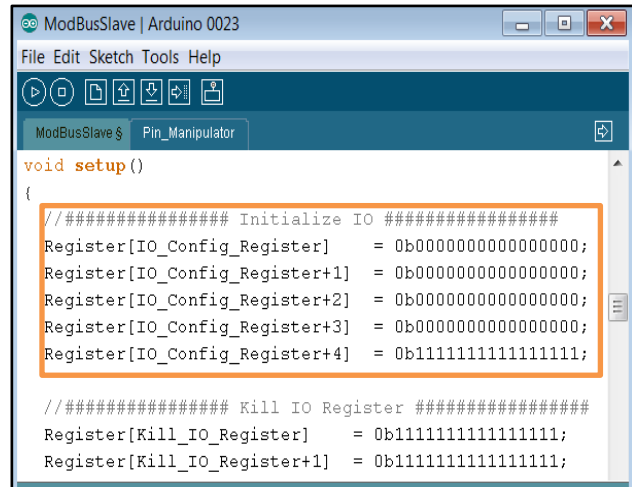


Figure 7 - IO Config Register

4.2. Analog Outputs (PWM)

Whether a pin is a digital I/O or an analog output (Pulse Width Modulation, PWM) is controlled by the PWMIOMap_Register (**Error! Reference source not found.**). Setting a bit to 0 will turn PWM off and a 1 will enable PWM. Please note, not all pins are capable of being PWM pins, see the Hardware section for which pins are used.

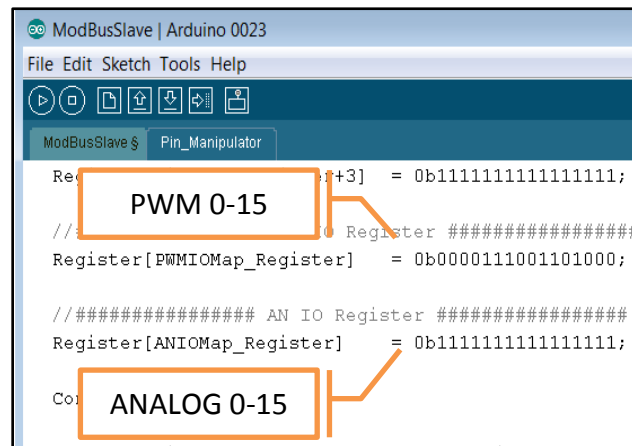


Figure 8 - Analog Config

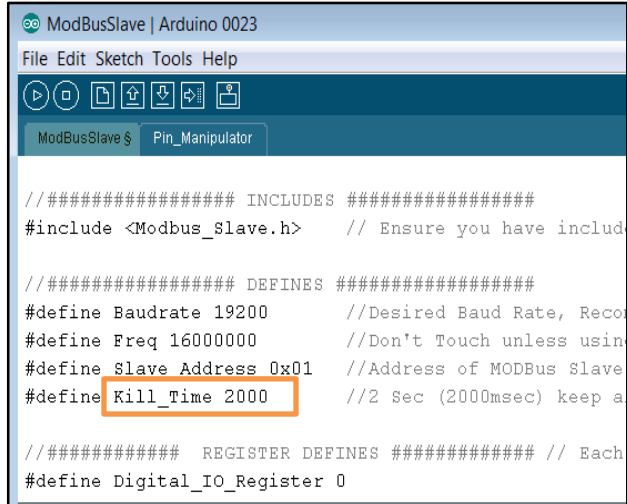
4.3. Analog Inputs

Analog input pins are configured using the ANIOMap_Register (**Error! Reference source not found.**). By default all capable pins are set to analog inputs (1). Setting a pin's corresponding bit to 1 will enable the analog functionality of the pin and a 0 will turn it

into a digital I/O pin. As a digital I/O pin, weather it is an input or an output is controlled just as any other digital I/O, using the 5th IO_Config_Register.

4.4.Kill Registry

As a safety function the board can be configured to turn certain pins off if it loses communication with the Host device after a given amount of time. The time that must elapse for communication to be assumed to have been lost is set under the define section “#define Kill_Time 2000” (Error! Reference source not found.). This value is in milliseconds and by default is 2sec. Which pins will be driven low when communications are lost are controlled by the Kill_IO_Register. By default all pins will be driven low on lost communications. Setting bits in the Kill_IO_Register to 1 will enable this function and 0 will disable it.



```

ModBusSlave | Arduino 0023
File Edit Sketch Tools Help

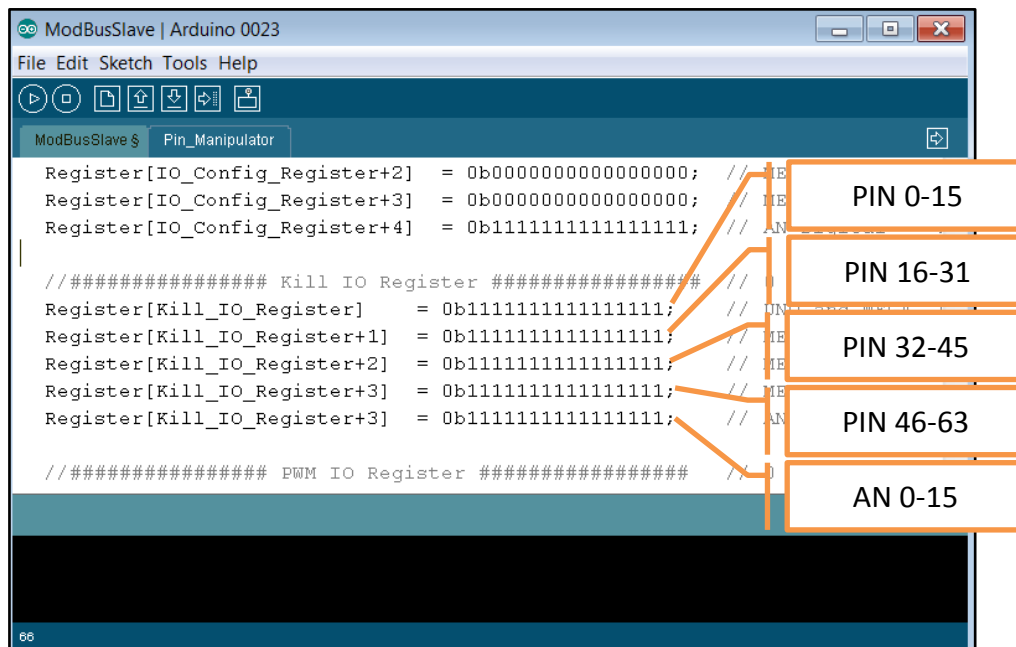
ModBusSlave$ Pin_Manipulator

//##### INCLUDES #####
#include <Modbus_Slave.h> // Ensure you have includ

//##### DEFINES #####
#define Baudrate 19200 //Desired Baud Rate, Reco
#define Freq 16000000 //Don't Touch unless usin
#define Slave Address 0x01 //Address of MODBus Slave
#define Kill_Time 2000 //2 Sec (2000msec) keep a

//##### REGISTER DEFINES ##### // Each
#define Digital_IO_Register 0
    
```

Figure 9 - Kill Time



```

ModBusSlave | Arduino 0023
File Edit Sketch Tools Help

ModBusSlave$ Pin_Manipulator

Register[IO_Config_Register+2] = 0b0000000000000000; // ME
Register[IO_Config_Register+3] = 0b0000000000000000; // ME
Register[IO_Config_Register+4] = 0b1111111111111111; // AN

//##### Kill IO Register #####
Register[Kill_IO_Register] = 0b1111111111111111; // ON
Register[Kill_IO_Register+1] = 0b1111111111111111; // ME
Register[Kill_IO_Register+2] = 0b1111111111111111; // ME
Register[Kill_IO_Register+3] = 0b1111111111111111; // ME
Register[Kill_IO_Register+3] = 0b1111111111111111; // AN

//##### PWM IO Register ##### // D
    
```

PIN 0-15

PIN 16-31

PIN 32-45

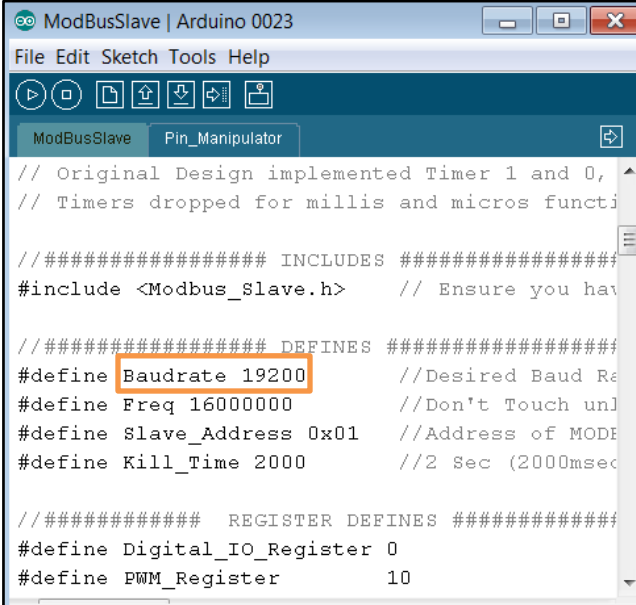
PIN 46-63

AN 0-15

Figure 10 - Kill IO Register

4.5. Serial Port

The Serial Port is pretty basic and only has one available setting. By default the baud rate is set to 19200bps, but can be set to anything in the range, 9200 – 115200bps.



```
ModBusSlave | Arduino 0023
File Edit Sketch Tools Help
ModBusSlave Pin_Manipulator
// Original Design implemented Timer 1 and 0,
// Timers dropped for millis and micros functi

##### INCLUDES #####
#include <Modbus_Slave.h> // Ensure you hav

##### DEFINES #####
#define Baudrate 19200 //Desired Baud Re
#define Freq 16000000 //Don't Touch unl
#define Slave_Address 0x01 //Address of MODE
#define Kill_Time 2000 //2 Sec (2000msec

##### REGISTER DEFINES #####
#define Digital_IO_Register 0
#define PWM_Register 10
```

Figure 11 – Baud rate

5. Registers

The Modbus protocol essentially keeps two sets of registers mirrored to each other. One lives on the Host and the other on the Slave. It is the Host's responsibility to know what each register means to the slave. The default setup of these registers is below. When a value is set, the slave automatically makes the required output changes and when an input happens, the corresponding register is set so it can be read. These registers govern all aspects of the Modbus Slave's state and operations.

Table 4 - Register Map

Register	Description	Register	Description	Register	Description
0	Digital I/O (Pin 0-15)	23-29	Unassigned	51-59	Unassigned
1	Digital I/O (Pin 16-31)	30	AN Value Pin A0	60	Config I/O (Pin 0-15)
2	Digital I/O (Pin 32-45)	31	AN Value Pin A1	61	Config I/O (Pin 16-31)
3	Digital I/O (Pin 46-63)	32	AN Value Pin A2	62	Config I/O (Pin 32-45)
4	Digital I/O (Pin A0-A15)	33	AN Value Pin A3	63	Config I/O (Pin 46-63)
5-9	Unassigned	34	AN Value Pin A4	64	Config I/O (Pin A0-A15)
10	PWM Value Pin 0	35	AN Value Pin A5	65-69	Unassigned
11	PWM Value Pin 1	36	AN Value Pin A6	70	Kill I/O Enable (Pin 0-15)
12	PWM Value Pin 2	37	AN Value Pin A7	71	Kill I/O Enable (Pin 16-31)
13	PWM Value Pin 3	38	AN Value Pin A8	72	Kill I/O Enable (Pin 32-45)
14	PWM Value Pin 4	39	AN Value Pin A9	73	Kill I/O Enable (Pin 46-63)
15	PWM Value Pin 5	40	AN Value Pin A10	74	Kill I/O Enable (Pin A0-A15)
16	PWM Value Pin 6	41	AN Value Pin A11	75-79	Unassigned
17	PWM Value Pin 7	42	AN Value Pin A12	80	PWMIOMap_Register
18	PWM Value Pin 8	43	AN Value Pin A13	81	ANIOMap_Register
19	PWM Value Pin 9	44	AN Value Pin A14	82-89	Unassigned
20	PWM Value Pin 10	45	AN Value Pin A15	90	General_Config
21	PWM Value Pin 11	46-49	Unassigned	91	Error_Register
22	PWM Value Pin 12	50	Timer	92-99	Unassigned

5.1.Digital I/O – Register 0-4

Registry 0-4 are readable and writable registry that set/get the current state of pins. Setting the corresponding bit to 1 will turn on an output while setting it to 0 will turn it off. As an input, the corresponding bit will be 1 when the pin is high and 0 when low. These bits are not set by PWM signals but are set from analog inputs. If a pin A0-A15 is configured as an analog input, the bit will be set to 1 if the voltage present is greater than 2.5V and 0 if less than.

5.2.PWM Value – Register 10-22

These registries are used to set the duty cycle of the PWM pins, if enabled. Allowable values range from 0 – 255, with 0 representing off 100% of the time and 255 representing on 100% of the time. Any value between is allowable but as an example, setting this register to 127 would give a duty cycle of 50%, meaning the pin would be high for half the time and low the remainder of the period. These are 16bit registries, however only the lower byte is used for PWM values.

5.3.Analog Values – Register 30-45

The value read by the analog pins is placed in there corresponding registers. This is a 10bit value ranging from 0 – 1024 (0-5V). This value is only set if analog functionality is enabled for the pin.

5.4. Timer – Register 50

Register 50 is available as an 1/8th second interval timer. This value will increment every 1/8th second once the board is turned on. The counter is 16bit and will count for over two hours before starting over. The 1/8th second structure was setup so that commonly used time intervals were easily available. 1/8th, 1/4th, 1/2, 1, 2... seconds are represented by a single bit in the register and that bit will toggle at the corresponding interval. Take a look at the 1/4sec column in the table below. You will notice that bit one is toggled between 1 and 0 every 1/4th second, meaning an LED controlled by this bit would flash twice a second and an LED controlled by bit 2 would flash once a second.

Table 5 - Timer Bit Field

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Time (sec)	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	1/2	1/4	1/8
1/8th sec	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1/4th sec	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
3/8th sec	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1/2 sec	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
5/8th sec	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
6/8th sec	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
7/8th sec	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
1 sec	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1.125 sec	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
1.25 sec	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
1.375 sec	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
1.5 sec	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

5.5.Config I/O – Register 60-64

The config registries are used to configure whether a pin is an input or an output. It is recommended that even for PWM and analog pins that you set these values appropriately (1 for analog in, and 0 for PWM) however this does not enable these functionalities. Setting these bits will setup the properties for each pin. If set to 0, the

pin will be set to an output and on startup driven low until set by a registry write from the host. If set to 1, the pin will be set to an input and a pull up resistor will be enabled (This pull up is disabled if subsequently configured as an analog input). Register 64 is used to configure the direction of the analog pins.

5.6.Kill I/O - Register 70-74

Kill I/O registries are used to control the behavior of a pin if the Kill timeout is activated. If a bit is set to 1, the corresponding output will be turned off as soon as a Kill timeout occurs. If however the bit is set to 0, the output will be left as is. This works for both digital outputs and PWM.

5.7.PWM Enable – Register 80

This 16bit register is used to enable PWM functionality on the corresponding pin. A value of 1 will enable PWM and 0 will disable it. Subsequently a PWM value must be set to generate a PWM signal. Setting this value on a pin that is unable to support PWM has no effect.

5.8.Analog Input – Register 81

Register 81 is used to enable (1) or disable (0) an analog input. If enabled, the analog read functionality takes precedence over all other settings. If disabled the system will look to the Config Register to determine the direction and functionality of the pin. This register only controls A0-A15, the analog inputs available on Arduino platforms

5.9.Config – Register 90

Currently unimplemented.

5.10. Error – Register 91

The error register stores the last error code to be encountered. This value is held until overwritten by another error or until the register is read. When it is successfully read it is returned to 0.

Table 6 - Error Codes

Error	Description
0	None
1	Invalid Slave Address
2	CRC Error
3	Invalid Register Address
255	Kill Timeout

6. Mach 3 Interface

If you are not using Mach 3 as your CNC controller this chapter will not directly apply, however some insight on interfacing with the Arduino Modbus Slave may be gleamed by seeing how it is used with at least one existing system. Mach 3, by Artsoft is a popular solution for the control of a CNC machine with a Windows system over a parallel port. Parallel ports have a fairly limited number of I/O pins and no PWM or Analog functionality. This is where the Modbus Slave comes in. It serves as an additional interface that does not use the parallel port and provides some extended functionality. What it is not for IS time critical, high data rate applications. It would not be advised to use a Modbus control for direct stepper control for instance.

6.1. Getting Started

Obviously you will need Mach 3 installed on your computer. It is recommended that you setup your system and get it functioning at a basic level before continuing. Once comfortable we can delve into Modbus control. Open Mach and click on the “Functions Cfg” dropdown menu. Then find and click on the “Setup Serial Modbus Control”. The following window should appear.

ModBus Configuration

Port Num: 9 Baud Rate: 19200 8-1-N ☐ Use RTS for transmit (RS485) Timeout: 100 ms Test ModBus

☒ Mod Run Can't Open Comm Port

	Enabled On/Off	Comment or Device	Port/Address	Slave # 0-10	Refresh 25ms Incr.	Address ModBus(Var)	# of Registers	Direction Input Output
Cfg #0	<input checked="" type="checkbox"/>		1	1	50	2	1	Output-Coils
Cfg #1	<input checked="" type="checkbox"/>		1	1	50	12	1	Input-Coils
Cfg #2	<input type="checkbox"/>		1	1	50	16	1	Input-Coils
Cfg #3	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #4	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #5	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #6	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #7	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #8	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #9	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #10	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #11	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #12	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #13	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #14	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #15	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #16	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #17	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #18	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #19	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #20	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #21	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #22	<input type="checkbox"/>		1	1	50	0	1	Input Reg

Apply OK

Figure 12 - ModBus Configuration Screen

Take note of the Port Num and Baud Rate. By default the Baud Rate should be 19,200bps and the Port Num should be whatever your computer sets. You can find this out by going to “Start” and then devices and printers. Scroll down to the bottom and you should see “Communication Port (COM _)”.

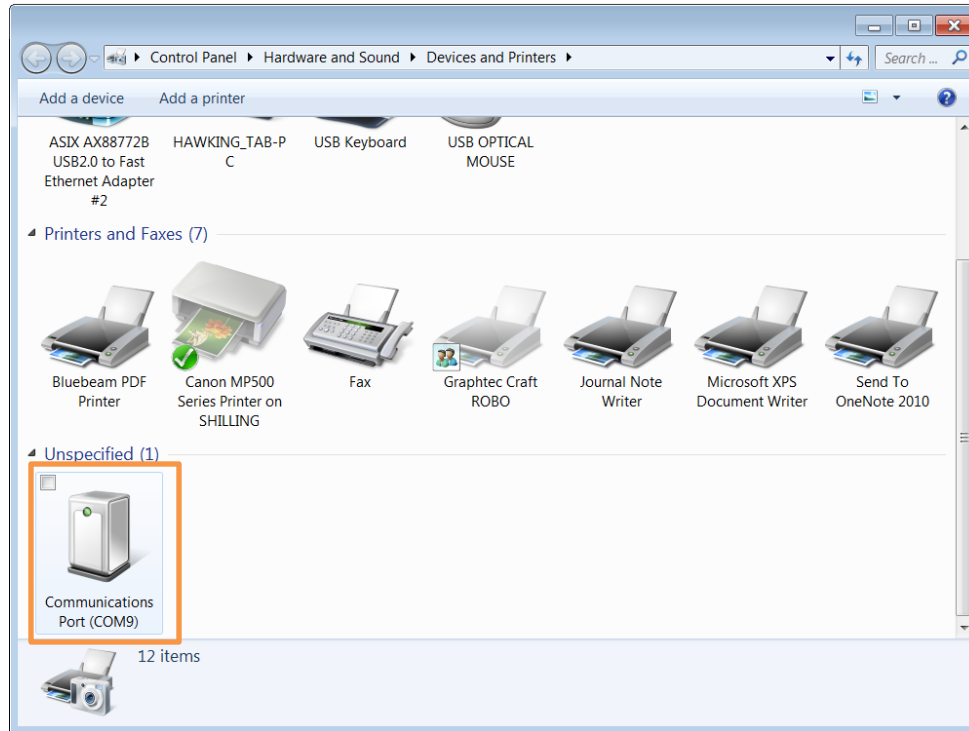


Figure 13 - Devices and Printers

Now, click on the “Test Modbus” button in the upper right corner of the screen. On this window, make sure the Baud rate (1) is set and the correct com port (2). Try pressing Open (3). The status text (4) should show “No Error”. Lets now set it up to read the timer register. The timer register is 16 bit, so it is a holding register. The slave is 1, the address is 50 and we want to read just one register (NumReg). There is a slide bar at the bottom. Slide it over a few places, this will run the board continuously. Now, press “Read” (5) and you will see the value on the right update continuously. This is the timer register updating.

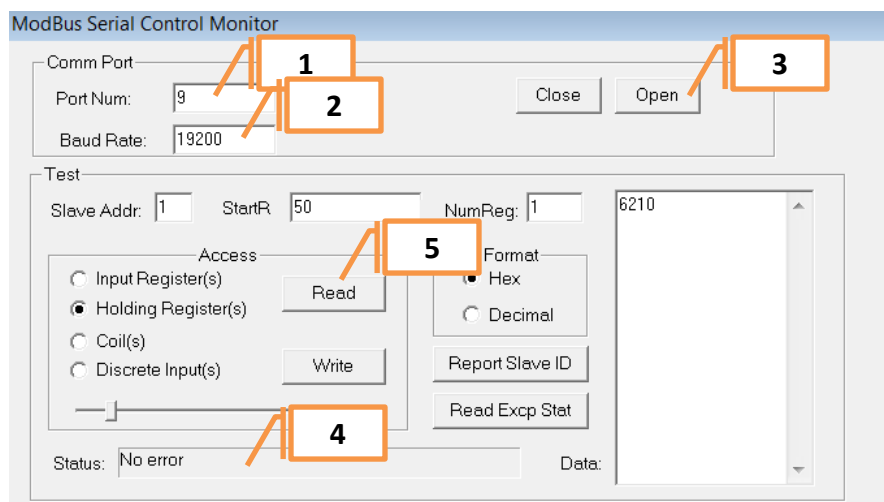


Figure 14 - ModBus Test

You have now communicated with the board for the first time. You can close the port and exit this screen.

6.2. Setting Up your Inputs and Outputs

Return to the Modbus Configuration screen. On the left side you see a list of CFG #_. Each of these represents an input or output for Mach 3. This is where the hardware interfaces with the software. First we will setup Mach to flash an LED, the traditional “Hello World” of the programming world. To do this we need to set two CFGs. Lets start with an output. We want to flash an LED on PIN 12. Set the port address to 0 for now. Slave # is the slave address, which by default is 1. Leave the refresh rate at 25ms and set the address to 12, for the 12th bit in the Registries, which corresponds to Pin 12 Digital I/O bit. We want to write one Coil, so # of Registers should be 1. We want to set a bit, so we will select “Output-Coils”. This will set 1 bit at bit 12 from the beginning of all registers. Next lets setup a read of the timer. From Table 4 - Register Map, you will see that the timer register is at address 50, so set the address to 50. Set the # of Registers to 1 and the direction to Input Holding. This will read the 50th whole 16 bit register.

ModBus Configuration

Port Num: 9 Baud Rate: 19200 8-1-N Use RTS for transmit (RS485) Timeout 100 ms Test ModBus

☒ ModBus Run No error

	Enabled On/Off	Comment or Device	Port/Address	Slave # 0-10	Refresh 25ms Incr.	Address ModBus Var	# of Registers	Direction Input Output
Cfg #0	<input checked="" type="checkbox"/>		0	1	25	12	1	Output-Coils
Cfg #1	<input checked="" type="checkbox"/>		0	1	25	50	1	Input Reg
Cfg #2	<input type="checkbox"/>		1	1	50	0	1	Input-Coils
Cfg #3	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #4	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #5	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #6	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #7	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #8	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #9	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #10	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #11	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #12	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #13	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #14	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #15	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #16	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #17	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #18	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #19	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #20	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #21	<input type="checkbox"/>		1	1	50	0	1	Input Reg
Cfg #22	<input type="checkbox"/>		1	1	50	0	1	Input Reg

Apply OK

Figure 15 - ModBus Configuration

Enable these configs and press apply. Your board should start flashing, but not the way you wanted. The TX and RX LEDs should be flashing, this lets you know the board is now constantly reading and writing bits, but it doesn't yet know what to do with the

data. Now we have to setup a brain. In the next section we will do a little surgery. Press Ok and let this window close.

6.3.Brains

Open the Brain editor and create a new brain called “Flash”:

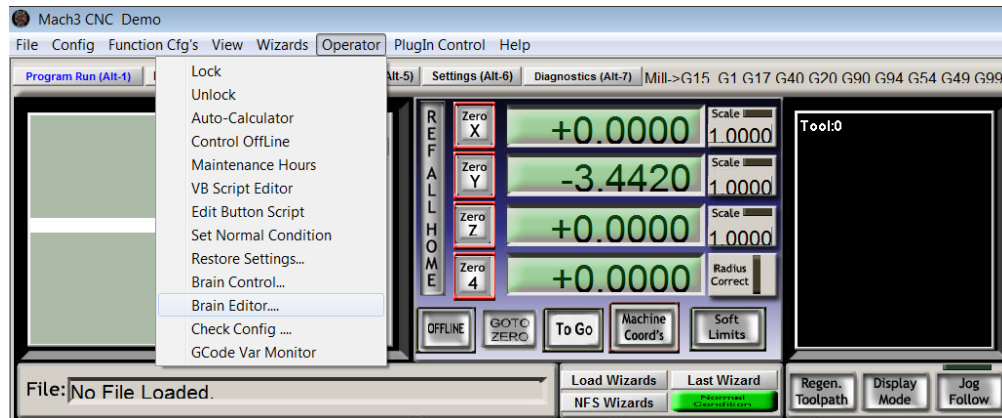


Figure 16 - Brain Editor

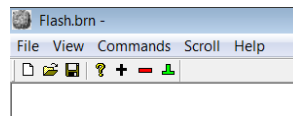


Figure 17 - Brain Editor, New

Press the “+” button and select “ModBus”. On this screen enter the data as seen below:

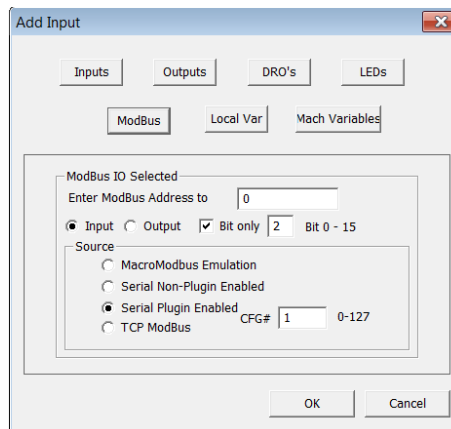


Figure 18 - Flash Brain, Adding an Input

Press Ok:

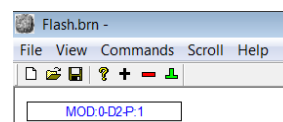


Figure 19 - Flash Brain, Input

We don't need to do anything with this input, we just want to pass it through. Select the node and press the "+" symbol again.

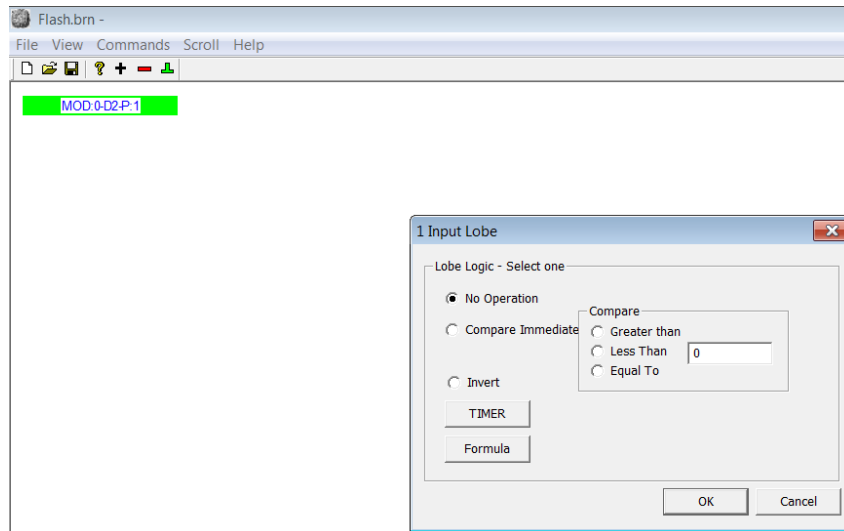


Figure 20 - Flash Brain, Timer Input Operation

On this dialog box, press "OK". Now select the operation (Second box created) and press the upside down "T".

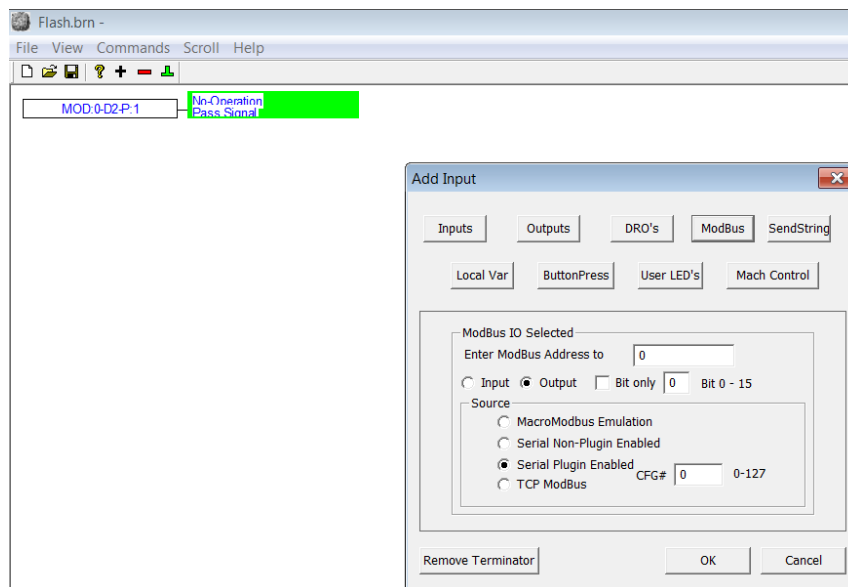


Figure 21 - Flash Brain, Operation

Select ModBus and enter the data as seen above. Press OK and that should be it. Your brain should look like this:

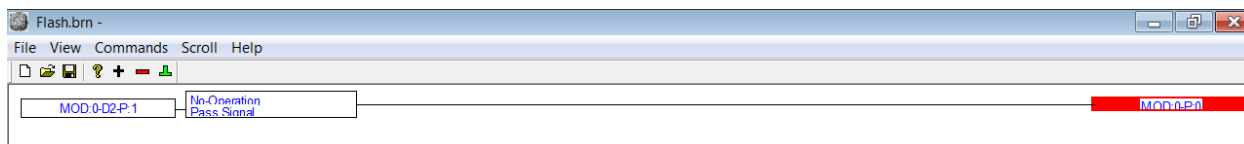


Figure 22 - Flash Control

Save the Brain and close the editor. Now open the Brain Control:

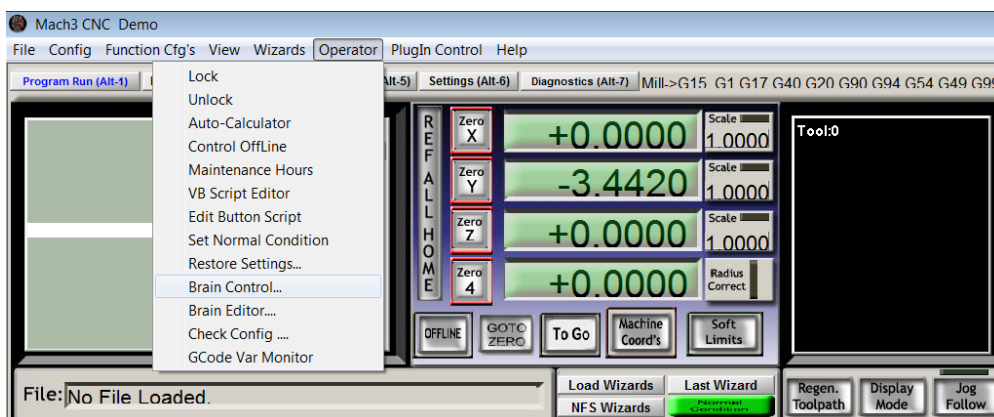


Figure 23 - Brain Control

This brings up a list of Brains. You may not see your Brain yet, so press “Reload All Brains”. You will need to do this every time you modify your Brain. Select your Brain “Flash” and check enabled.

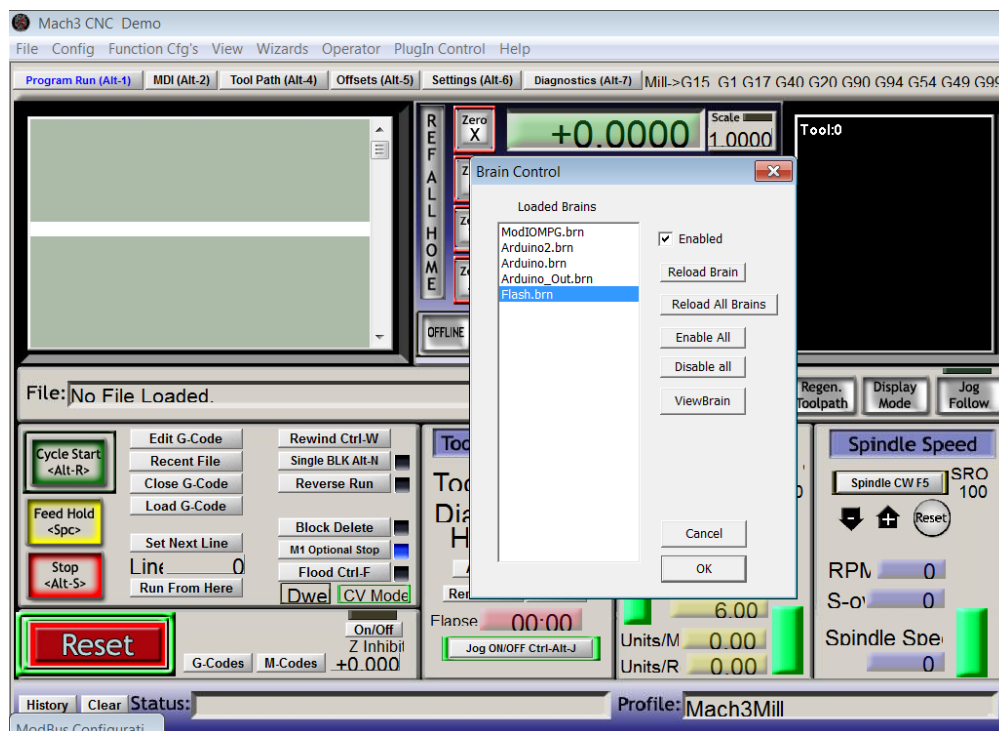


Figure 24 - Brain Control - Enable

Press OK and your LED should start flashing. WOW, a flashing LED... Lets learn a few things from the LED now. Watch it closely. Notice that every few blinks are a little longer or a little shorter than the last. Why is this you might ask? The system is only being updated every 25ms. The Timer is read every 25ms and then the LED is turned on or off. Things are not instantaneous and this is exactly the “glitch” you need to be cautious of when designing your system. If something is so critical that a small “glitch” would be devastating, then you need to be using a parallel port pin, otherwise, the Modbus system is still your solution. Feed override switches, status lights, MPGs and many others are all great options.

6.4.A Higher Level Brain

Lets get a little more complicated. We will make the LED only flash if the spindle is turned on. Open the Brain editor again. Create a new Brain called “Spindle_Flash”. Add the timer input just as before. Add the pass through operation as well but do not add the output termination. Now we will add the spindle on “input”. Press the “+” sign again.

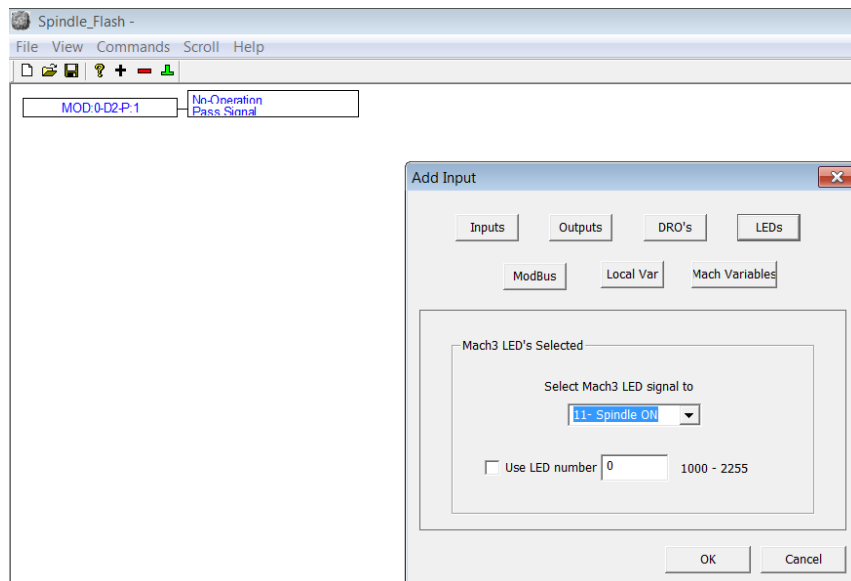


Figure 25 - Spindle Flash, Spindle On Input

Press the LED button and then select the “Spindle On” option from the dropdown. Press Ok. Select the new Input and add an operation “+”. Select no operation and OK.

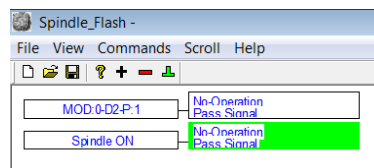


Figure 26 - Spindle Flash, Spindle On

Select Both Operations (The two boxes on the right) by holding ctrl and clicking on both and press the “+” symbol again. Select “And” and “OK”.

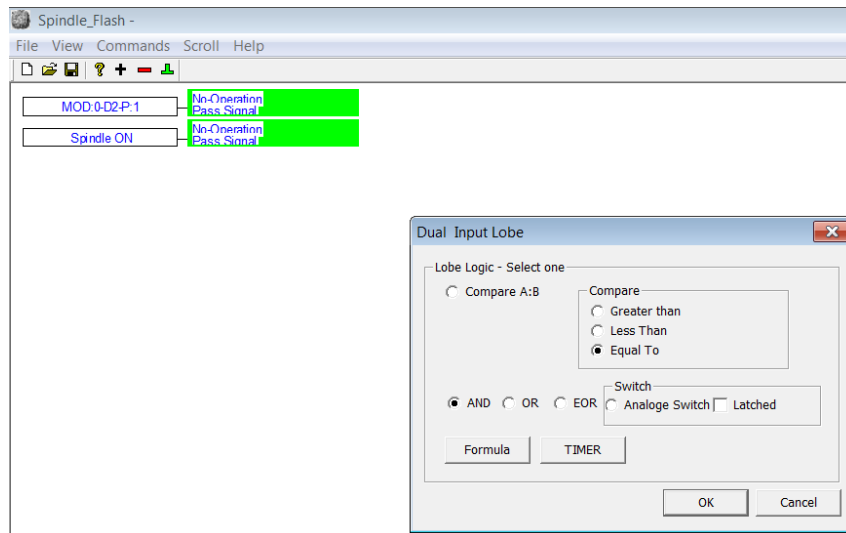


Figure 27 - Spindle Flash, AND Operation

This will compare both inputs, the 2nd bit of the Timer Register and the spindle state. If both are a 1, or on, then it will pass a 1. If either is off, it will pass a 0. We are going to now set it up so that this value gets passed to the LED on our Modbus board. Select the “AND” operation and press the upside down “T”.

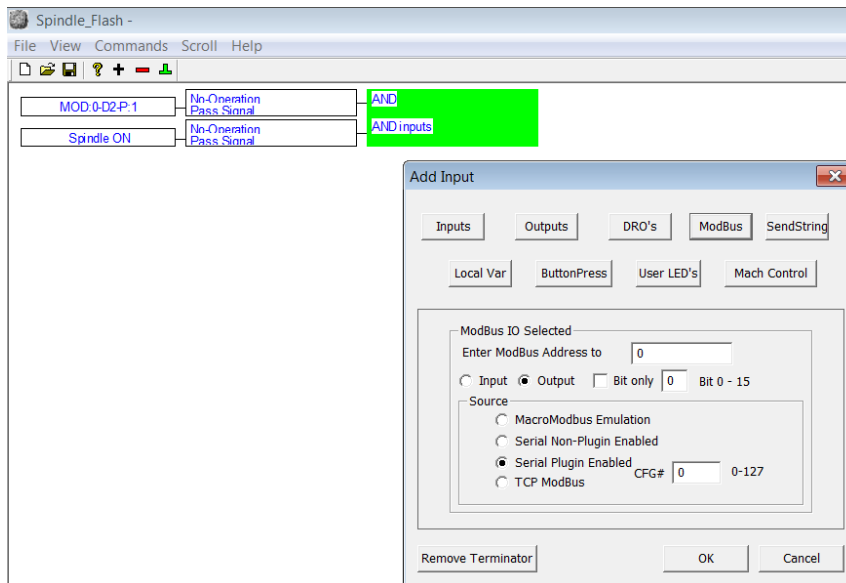
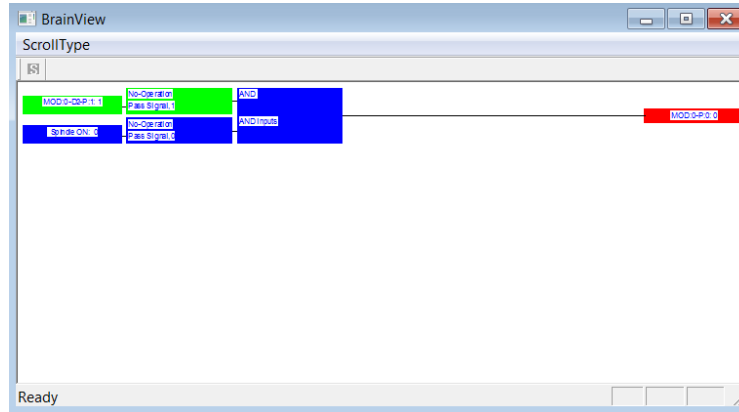


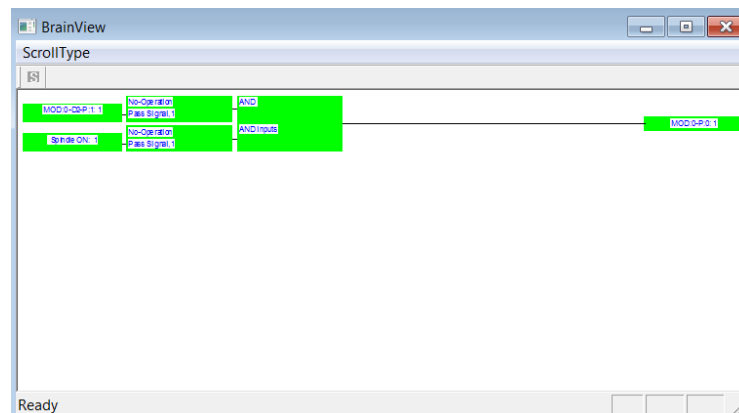
Figure 28 - Spindle Flash, LED Out

Select Modbus and configure it as above. Press “OK” and we are done. Save the brain and exit.

Open the Brain control again and “Reload All Brains”. Select your brain and enable it. Then press “view”



You will see that the lower blocks are blue, because the spindle is off and the upper block are flashing. No go into Mach and press the “Spindle” button on the “Program Run” tab. Your Brain view should now show the lower blocks as green and whenever all blocks on the left are green, you will see the very right box turn green.



If your LED is plugged in, then you will also see your LED flashing. When you turn the spindle off, the LED should stop. With this basic knowledge of Modbus and Mach 3 Brains you can do many things, you just have to play around and experiment. There are also some brains freely available for common setups.

7. Conclusion

Hopefully you have learned a lot about ModBus and Arduino. There are many strengths to the system and a few weaknesses. With a knowledge of both you should be able to construct an unlimited number of peripheral devices and controls for your CNC. Further expansion can be had through Mach 3's VB scripting and SDK which were not covered in this document. There are many ModBus solutions available but few offer quite as much customization potential and none can be picked up at the local electronics store. The document and code are provided as a stepping stone for further development, so please tinker and expand. Share what you find out and create.

APPENDIX A – ModBusSlave.pde

/*

Modbus Slave designed for Mach 3 CNC Software
Written by: Tim W. Shilling, March 4, 2012.

Copyright (c) 2012 Tim W. Shilling (www.ShillingSystems.com)

Arduino Modbus Slave 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.

Arduino Modbus Slave 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.

To get a copy of the GNU General Public License see <<http://www.gnu.org/licenses/>>.

*/

//#####

// Supported Functions

// 1: Read Coil

// 2: Read Discrete Input

// 3: Read Holding Reg

// 4: Read Input Reg

// 5: Write Single Coil

// 6: Write Single Reg

// 7 Read Exception

// 15: Write Multiple Coils

// 16: Write Multiple Holding Reg

// Notes on Timing:

// Original Design implemented Timer 1 and 0, but these conflict with the PWM generators

// Timers dropped for millis and micros functions. Not as accurate but fine for this purpose

//##### INCLUDES #####

#include <Modbus_Slave.h> // Ensure you have included the libraries in the Arduino Libraries folder

//##### DEFINES #####

```
#define Baudrate 19200    //Desired Baud Rate, Recommend, 9600, 19200, 56800, 115200
#define Freq 16000000    //Don't Touch unless using a differnt board, Freq of Processor
#define Slave_Address 0x01 //Address of MODBus Slave
#define Kill_Time 2000    //2 Sec (2000msec) keep alive, 0 => OFF

//##### REGISTER DEFINES ##### // Each Register is 16bit
#define Digital_IO_Register 0
#define PWM_Register 10
#define AN_Register 30
#define Timer_Register 50
#define IO_Config_Register 60
#define Kill_IO_Register 70
#define PWMIOMap_Register 80
#define ANIOMap_Register 81
#define General_Config 90
#define Error_Register 91
#define Digital_IO_Pins 14 //Total number of Digital IO pins, Limits update scanner pin count
#define Number_Of_Registers 100
//##### GLOBAL VARIABLES #####
unsigned char Data[256]; // All received data ends up in here, also used as output buffer
unsigned short Index = 0; // Current Location in Data
unsigned short Register[Number_Of_Registers]; // Where all user data, Coils/Registers are kept
ModBusSlave ModBus(Slave_Address, Register, Number_Of_Registers); // Initialize a new ModBus Slave, Again,
you must have my Arduino ModBusSlave Library
unsigned long Last_Time=0;
unsigned long Time = 0;
unsigned long LongBreakTime; //Time for 3.5 characters to be RX
//##### Setup #####
// Takes: Nothing
// Returns: Nothing
// Effect: Opens Serial port 1 at defined Baudrate
// Configures all Pins
// Initialized Timer 1
void setup()
{
    //##### Initialize IO ##### // 0 => Output, 1 => Input, opposite of normal Arduino,
    but my habit from other platforms, 0 looks like an O and 1 looks like an I
    Register[IO_Config_Register] = 0b0000000000000000; // UNO and MEGA PIN 00-15
    Register[IO_Config_Register+1] = 0b0000000000000000; // MEGA PIN 16-31
    Register[IO_Config_Register+2] = 0b0000000000000000; // MEGA PIN 32-47
    Register[IO_Config_Register+3] = 0b0000000000000000; // MEGA PIN 48-64
    Register[IO_Config_Register+4] = 0b1111111111111111; // AN Digital PIN A0-A16

    //##### Kill IO Register ##### // 0 => Leave, 1 => Kill
    Register[Kill_IO_Register] = 0b1111111111111111; // UNO and MEGA PIN 00-15
    Register[Kill_IO_Register+1] = 0b1111111111111111; // MEGA PIN 16-31
    Register[Kill_IO_Register+2] = 0b1111111111111111; // MEGA PIN 32-47
    Register[Kill_IO_Register+3] = 0b1111111111111111; // MEGA PIN 48-64
    Register[Kill_IO_Register+4] = 0b1111111111111111; // AN Digital PIN A0-A16

    //##### PWM IO Register ##### // 0 => Normal I/O, 1 => PWM I/O
    Register[PWMIOMap_Register] = 0b0000111001101000; // UNO and MEGA PWM 01-16
```

```
//##### AN IO Register ##### // 0 => Digital, 1=> Analog
Register[ANIOMap_Register] = 0b11111111111111; // UNO and MEGA

Config_IO();

LongBreakTime = (long)((long)28000000.0/(long)Baudrate);
if(Baudrate > 19200)
    LongBreakTime = 1750; // 1.75 msec
Serial.begin(Baudrate); // Open Serial port at Defined Baudrate
Time = micros(); // Preload Time variable with current System microsec
}

//##### Main Loop #####
// Takes: Nothing
// Returns: Nothing
// Effect: Main Program Loop

unsigned long LongKillTime = (long)((long)Kill_Time * (long)1000); //Time ellapsed between RX that causes
system Kill

unsigned long Keep_Alive = 0; // Keep Alive Counter Variable

void loop()
{
    Update_Time();
    if(Keep_Alive >= LongKillTime){ // Communications not Received in Kill_Time, Execute Lost Comm
Emergency Procedure
        if(Kill_Time != 0){ // If Kill_Time is 0, then disable Lost Comm Check
            Keep_Alive = LongKillTime; // Avoid Keep_Alive rollover
            ModBus.Error = 0xff; // Set error code to Lost Comm
            Register[Error_Register] = ModBus.Error; // Set Error Register
            Kill_IO(); // Kill what should be killed
            digitalWrite(13,1); // Indicate Error with Solid LED
        }
    }
    else
    {
        Update_Pin_States(); // Update Digital Pins
        Update_AN_States(); // Update Analog Pins
        if(ModBus.Error != 0){ // Flash Error LED is ModBus Error != 0
            digitalWrite(13,bitRead(Time,11)); // Toggle LED
        }
        else // If no Error
        {
            digitalWrite(13,0); // Turn off LED
        }
    }
}

//##### RX Data #####
if (Serial.available() > 0) // If Data Avilable
{
    Data[Index] = (unsigned char)Serial.read(); // Read Next Char
    Keep_Alive = 0; // Reset Keep Alive counter
    Last_Time = micros(); // Update Last_Time to current Time
}
```

```
    Index++;                // Move Index Counter Forward
}
//##### Process Data #####
if(Index > 0)                // If there are bytes to be read
{
    if(Keep_Alive >= LongBreakTime)    // Transmission Complete, 3.5 char spacing observed
    {
        ModBus.Process_Data(Data,Index);    // Process Data
        if(ModBus.Error == 0)                // If no Errors, then...
        {
            Keep_Alive = 0;                // Reset Keep Alive
            Last_Time = micros();            // Set Last_Time to current time
        }
        else                                // If there was an error
        {
            Register[Error_Register] = ModBus.Error; // Set Error Code
        }
        Index = 0;                // Reset Index to 0, no bytes to read
    }
}
//##### Update Time #####
// Takes: Nothing
// Returns: Nothing
// Effect: Updates Timer Register
//      Updates KeepAlive Count

void Update_Time()
{
    Last_Time = Time;                // Set Last_Time to Current Time
    Time = micros();
    if(Time < Last_Time)                // Counter has rolled over, should take 70 Min
    {
        Last_Time = 0;                // Introduces a small clitch in timing by not accounting for how bad the roll over
was. Happens every 70 Min                // Result is Delta will be smaller than it should be, Kill and Character Spacing will be usec
longer than they should
    }
    Keep_Alive += Time - Last_Time;    // Increment Keep Alive counter
    Register[Timer_Register] = (unsigned int)(millis() / 125.0); // Converts to 1/8sec and load into Register
}
//EOF
```

APPENDIX B – Pin_Manipulator.pde

```
/*
    Copyright (c) 2012 Tim W. Shilling (www.ShillingSystems.com)

    Arduino Modbus Slave 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.

    Arduino Modbus Slave 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.

    To get a copy of the GNU General Public License see <http://www.gnu.org/licenses/>.
*/
//##### Config IO #####
// Takes: Nothing
// Returns: Nothing
// Effect: Configures all Pins, Digital, PWM and Analog
//      Sets everything initially to OFF

void Config_IO()
{
    unsigned char Pin = 0;          // PIN Number indexer
    for(int Index = 0 ; Index < 10; Index++)    // Scan through all available Registers, 0-9
    {
        for(int Bit = 0; Bit < 16; Bit++)        // Scan through all 16 bits of each Register
        {
            Pin = (Index * 16)+Bit;              // Calculate corresponding Pin
            if(Pin >= Digital_IO_Pins)           // If we have reached the number of IO pins
            {
                Index = 11;                      // Set Index above max Index to force exiting of outer Loop
                break;                            // Break For Loop
            }
            pinMode(Pin,~bitRead(Register[Index+IO_Config_Register],Bit)); // Set Pin Mode to Input or Output
            digitalWrite(Pin,bitRead(Register[Index+IO_Config_Register],Bit)); // Turns on Internal Pull-Up Resistor if Input
        }
    }
    for(int Bit = 0; Bit < 16; Bit++)            // Cycle through all Analog pins
    {
        if(bitRead(Register[ANIOMap_Register],Bit)==0) // If I/O Pin
        {
            if(Bit == 0)                          // If Output
            {
                pinMode(A0,~bitRead(Register[IO_Config_Register+4],Bit)); // Set Pin Mode to Input or Output
                digitalWrite(A0,bitRead(Register[IO_Config_Register+4],Bit)); // Turns on Internal Pull-Up Resistor if Input
            }
            if(Bit == 1)
            {
                pinMode(A1,~bitRead(Register[IO_Config_Register+4],Bit));
                digitalWrite(A1,bitRead(Register[IO_Config_Register+4],Bit));
            }
            if(Bit == 2)
```



```
{
  pinMode(A2,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A2,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 3)
{
  pinMode(A3,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A3,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 4)
{
  pinMode(A4,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A4,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 5)
{
  pinMode(A5,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A5,bitRead(Register[IO_Config_Register+4],Bit));
}
#if defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__)    // If larger board
if(Bit == 6)
{
  pinMode(A6,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A6,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 7)
{
  pinMode(A7,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A7,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 8)
{
  pinMode(A8,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A8,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 9)
{
  pinMode(A9,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A9,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 10)
{
  pinMode(A10,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A10,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 11)
{
  pinMode(A11,~bitRead(Register[IO_Config_Register+4],Bit));
  digitalWrite(A11,bitRead(Register[IO_Config_Register+4],Bit));
}
if(Bit == 12)
{
  pinMode(A12,~bitRead(Register[IO_Config_Register+4],Bit));
```

```

        digitalWrite(A12,bitRead(Register[IO_Config_Register+4],Bit));
    }
    if(Bit == 13)
    {
        pinMode(A13,~bitRead(Register[IO_Config_Register+4],Bit));
        digitalWrite(A13,bitRead(Register[IO_Config_Register+4],Bit));
    }
    if(Bit == 14)
    {
        pinMode(A14,~bitRead(Register[IO_Config_Register+4],Bit));
        digitalWrite(A14,bitRead(Register[IO_Config_Register+4],Bit));
    }
    if(Bit == 15)
    {
        pinMode(A15,~bitRead(Register[IO_Config_Register+4],Bit));
        digitalWrite(A15,bitRead(Register[IO_Config_Register+4],Bit));
    }
#endif

}
}
}

//##### Kill IO #####
// Takes: Nothing
// Returns: Nothing
// Effect: Drive all desired PIN Low on execution

void Kill_IO()
{
    unsigned char Pin = 0;          // PIN Number indexer
    for(int Index = 0 ; Index < 10; Index++)    // Scan through all available Registers, 0-9
    {
        for(int Bit = 0; Bit < 16; Bit++)        // Scan through all 16 bits of each Register
        {
            Pin = (Index * 16)+Bit;              // Calculate corresponding Pin
            if(Pin >= Digital_IO_Pins)           // If we have reached the number of IO pins
            {
                Index = 11;                      // Set Index above max Index to force exiting of outer Loop
                break;                            // Break For Loop
            }
            if(bitRead(Register[Index+IO_Config_Register],Bit) == 0) // If Output, then...
            {
                if(bitRead(Register[Index+Kill_IO_Register],Bit) == 1) // If Kill Desired on Output, then...
                {
                    digitalWrite(Pin,0);         // Drive Output Low
                }
            }
        }
    }
    for(int Bit = 0; Bit < 16;Bit++)            // Cycle through all Analog Pins
    {
        if(bitRead(Register[ANIMap_Register],Bit) == 0)    // If Analog mode is Enabled, then...

```

```
{
  if(bitRead(Register[IO_Config_Register+4],Bit) == 0) // If Output, then...
  {
    if(bitRead(Register[Kill_IO_Register],Bit) == 1) // If Kill Desired on Output, then...
    {
      if(Bit == 0)
        digitalWrite(A0,LOW);          // Drive PIN Low
      if(Bit == 1)
        digitalWrite(A1,LOW);
      if(Bit == 2)
        digitalWrite(A2,LOW);
      if(Bit == 3)
        digitalWrite(A3,LOW);
      if(Bit == 4)
        digitalWrite(A4,LOW);
      if(Bit == 5)
        digitalWrite(A5,LOW);

#ifdef __AVR_ATmega1280__ || defined(__AVR_ATmega2560__) // If Larger Board
      if(Bit == 6)
        digitalWrite(A6,LOW);
      if(Bit == 7)
        digitalWrite(A7,LOW);
      if(Bit == 8)
        digitalWrite(A8,LOW);
      if(Bit == 9)
        digitalWrite(A9,LOW);
      if(Bit == 10)
        digitalWrite(A10,LOW);
      if(Bit == 11)
        digitalWrite(A11,LOW);
      if(Bit == 12)
        digitalWrite(A12,LOW);
      if(Bit == 13)
        digitalWrite(A13,LOW);
      if(Bit == 14)
        digitalWrite(A14,LOW);
      if(Bit == 15)
        digitalWrite(A15,LOW);
#endif
    }
  }
}

//##### Update Analog States #####
// Takes: Nothing
// Returns: Nothing
// Effect: Updates Analog Pin, either as Digital I/O pins or as Analog Read Pins
void Update_AN_States()
{
  for(int Bit = 0; Bit < 16; Bit++)          // Cycle through available pins
```

```
{
if(bitRead(Register[ANIOMap_Register],Bit)==1)    // If Analog mode, then...
{
    Register[Bit+AN_Register] = analogRead(Bit);    // Load Register with read value
    if(Register[Bit+AN_Register] >= 512)            // If Value is >= 2.5 Volts, then...
    {
        bitWrite(Register[Digital_IO_Register+4],Bit,1);// Set Digital I/O Register to 1
    }
    else                                            // If < 2.5 Volts, then...
    {
        bitWrite(Register[Digital_IO_Register+4],Bit,0);// Set Digital I/O Register to 0
    }
}
else                                            // If Digital Mode, then...
{
    if(bitRead(Register[IO_Config_Register+4],Bit) == 1) // If Input Pin
    {
        if(Bit == 0)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A0));
        if(Bit == 1)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A1));
        if(Bit == 2)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A2));
        if(Bit == 3)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A3));
        if(Bit == 4)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A4));
        if(Bit == 5)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A5));
#ifdef __AVR_ATmega1280__ || defined(__AVR_ATmega2560__)
        if(Bit == 6)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A6));
        if(Bit == 7)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A7));
        if(Bit == 8)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A8));
        if(Bit == 9)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A9));
        if(Bit == 10)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A10));
        if(Bit == 11)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A11));
        if(Bit == 12)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A12));
        if(Bit == 13)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A13));
        if(Bit == 14)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A14));
        if(Bit == 15)
            bitWrite(Register[Digital_IO_Register+4],Bit,digitalRead(A15));
#endif
    }
    else                                            // If Digital Pin, then...
```

```

{
    if(Bit == 0)
        digitalWrite(A0,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 1)
        digitalWrite(A1,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 2)
        digitalWrite(A2,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 3)
        digitalWrite(A3,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 4)
        digitalWrite(A4,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 5)
        digitalWrite(A5,bitRead(Register[Digital_IO_Register+4],Bit));

#ifdef __AVR_ATmega1280__ || defined(__AVR_ATmega2560__)
    if(Bit == 6)
        digitalWrite(A6,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 7)
        digitalWrite(A7,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 8)
        digitalWrite(A8,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 9)
        digitalWrite(A9,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 10)
        digitalWrite(A10,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 11)
        digitalWrite(A11,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 12)
        digitalWrite(A12,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 13)
        digitalWrite(A13,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 14)
        digitalWrite(A14,bitRead(Register[Digital_IO_Register+4],Bit));
    if(Bit == 15)
        digitalWrite(A15,bitRead(Register[Digital_IO_Register+4],Bit));
#endif
}
}
}

##### Update Pin States #####
// Takes: Nothing
// Returns: Nothing
// Effect: Updates digital I/O pins and PWM
void Update_Pin_States()
{
    Register[Error_Register] = ModBus.Error;
    unsigned char Pin = 0;          // PIN Number indexer
    for(int Index = 0 ; Index < 10; Index++)    // Scan through all available Registers, 0-9
    {
        for(int Bit = 0; Bit < 16; Bit++)        // Scan through all 16 bits of each Register
        {

```

```
Pin = (Index * 16)+Bit;           // Calculate corresponding Pin
if(Pin >= Digital_IO_Pins)        // If we have reached the number of IO pins
{
    Index = 11;                   // Set Index above max Index to force exiting of outer Loop
    break;                       // Break For Loop
}
if(bitRead(Register[Index+IO_Config_Register],Bit) == 1)    // If Input, then...
{
    bitWrite(Register[Index+Digital_IO_Register],Bit,digitalRead(Pin)); // Read input state into current Bit
}
else
{
    if(Pin < 16) // If PWM is even possible, all available PWM are on pins 0-15 on current Arduino devices
    {
        if(bitRead(Register[PWMIOMap_Register],Bit) == 0) //Not a PWM pin, then...
        {
            digitalWrite(Pin,bitRead(Register[Digital_IO_Register],Bit)); // Set Output to reflecc bit
        }
        else
        {
            analogWrite(Pin,lowByte(Register[PWM_Register+Pin])); // Set PWM Duty Cycle
        }
    }
    else
    {
        digitalWrite(Pin,bitRead(Register[Index+Digital_IO_Register],Bit)); // Set Output to reflecc bit
    }
}
}
}
}
//EOF
```

APPENDIX C – CRC.cpp

```
/*
    CRC16.cpp is a library for the Arduino System. It complies with ModBus protocol, customized for
    exchanging
    information between Industrial controllers and the Arduino board.

    Copyright (c) 2012 Tim W. Shilling (www.ShillingSystems.com)
    Arduino Modbus Slave 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.

    Arduino Modbus Slave 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.

    To get a copy of the GNU General Public License see <http://www.gnu.org/licenses/>.
*/
#include "WProgram.h"
#include "CRC16.h"

static unsigned char auchCRCHi[] = {
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
0x40
};

static char auchCRCLo[] = {
0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4,
0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09,
0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F, 0xDD,
0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xDD,
0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32, 0x36, 0xF6, 0xF7,
0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA, 0xEE,
0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2,
0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F,
```

```
0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x78, 0xB8, 0xB9, 0x79, 0xBB,
0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75, 0xB5,
0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0, 0x50, 0x90, 0x91,
0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59, 0x58, 0x98, 0x88,
0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80,
0x40
};
```

```
CRC::CRC()
{

}
```

```
unsigned short CRC::CRC16(unsigned char *puchMsg,unsigned short usDataLen)
{
    unsigned char uchCRCHi = 0xff;
    unsigned char uchCRCLo = 0xff;
    unsigned int uIndex;
    while(usDataLen--)
    {
        uIndex = uchCRCLo ^ *puchMsg++;
        uchCRCLo = uchCRCHi ^ auchCRCHi[uIndex];
        uchCRCHi = auchCRCLo[uIndex];
    }
    return (uchCRCHi<<8|uchCRCLo);
}
```

APPENDIX D – CRC.h

```
/*
```

```
    CRC16.cpp is a library for the Arduino System. It complies with ModBus protocol, customized for
    exchanging
    information between Industrial controllers and the Arduino board.
```

```
    Copyright (c) 2012 Tim W. Shilling (www.ShillingSystems.com)
```

```
    Arduino Modbus Slave 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.
```

```
    Arduino Modbus Slave 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.
```

```
    To get a copy of the GNU General Public License see <http://www.gnu.org/licenses/>.
```

```
*/
#ifndef CRC16_h
#define CRC16_h
#include "WProgram.h"
class CRC
{
```



```
public:
    CRC();
    unsigned short CRC16(unsigned char *puchMsg,unsigned short usDataLen);
};
#endif
```

APPENDIX E – Modbus_Slave.cpp

```
/*
    ModBusSlave.cpp is a library for the Arduino System. It complies with ModBus protocol, customized for
    exchanging
    information between Industrial controllers and the Arduino board.

    Copyright (c) 2012 Tim W. Shilling (www.ShillingSystems.com)
    Arduino Modbus Slave 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.

    Arduino Modbus Slave 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.

    To get a copy of the GNU General Public License see <http://www.gnu.org/licenses/>.
*/

#include "WProgram.h"
#include "Modbus_Slave.h"
#include <CRC16.h>

CRC CheckSum; // From Checksum Library, CRC15.h, CRC16.cpp

//##### ModBusSlave #####
// Takes: Slave Address, Pointer to Registers and Number of Available Registers
// Returns: Nothing
// Effect: Initializes Library

ModBusSlave::ModBusSlave(unsigned char _Slave, unsigned short *_Coils, unsigned short _Coil_Count)
{
    Slave = _Slave;
    Coils = _Coils;
    Error = 0;
    Coil_Count = _Coil_Count;
}

//##### Process Data #####
// Takes: Data stream buffer from serial port, number of characters to read
// Returns: Nothing
// Effect: Reads in and parses data

void ModBusSlave::Process_Data(unsigned char *Buffer, unsigned char Count)
{
    Count--; // Convert Byte count to
    byte index;
    unsigned char Slave_Address = Buffer[0]; // Grab Slave Address out of buffer
    if(Slave_Address > 247) // If Address greater than 247 =>
        Invalid
        {
            Error=1;
            return;
        }
}
```

```
    }
    if(Slave_Address != 0)                                // Slave = 0 for Broadcast Msg
    {
        if(Slave != Slave_Address)                        // If Msg is not for this Slave
        {
            return;
        }
    }
    unsigned char Function = Buffer[1];
        // Grab Function Code
    unsigned short CRC = ((Buffer[Count]<<8)| Buffer[Count-1]);    // Transmitted CRC
    unsigned short Recalculated_CRC = CheckSum.CRC16(Buffer,Count-1);    // Computer CRC
    if(Recalculated_CRC != CRC)
        // Compare CRC, if not equal, then...
    {
        Error = 2;
        return;
    }
    if(Function == 1)
        //Read Coils
    {
        Read_Coils(Buffer);
    }
    if(Function == 2)
        //Read Descrete Input
    {
        Read_Coils(Buffer);
    }
    if(Function == 3)
        //Read Reg
    {
        Read_Reg(Buffer);
    }
    if(Function == 4)
        //Read Input Reg
    {
        Read_Reg(Buffer);
    }
    if(Function == 5)
        //Write Single Coil
    {
        Write_Single_Coil(Buffer);
    }
    if(Function == 6)
        //Write Single Reg
    {
        Write_Single_Reg(Buffer);
    }
    if(Function == 7)
        //Read Exception Status
    {
        Read_Exception(Buffer);
    }
```

```
if(Function == 15)
    //Write Coils
{
    Write_Coils(Buffer);
}
if(Function == 16)
    //Write Reg
{
    Write_Reg(Buffer);
}

    Error = 0;
                                //We made it to the end, Set Error to 0
}

//##### Read Exception #####
// Takes:  In Data Buffer
// Returns: Nothing
// Effect: Sets Reply Data and sends Response
void ModBusSlave::Read_Exception(unsigned char *Data_In)
{
    Data_In[2] = Error;
    Send_Response(Data_In,3);
    Error = 0;
}
//##### Send Response #####
// Takes:  In Data Buffer, Length
// Returns: Nothing
// Effect: Sends Response over serial port
void ModBusSlave::Send_Response(unsigned char *Data_In,unsigned short Length)
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
//##### Write_Single_Reg #####
// Takes:  In Data Buffer
// Returns: Nothing
// Effect: Sets Reply Data and Register values, sends Response for Write Single Reg.
void ModBusSlave::Write_Single_Reg(unsigned char *Data_In) // Function code 1
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
//##### Read_Single_Reg #####
// Takes:  In Data Buffer
// Returns: Nothing
// Effect: Sets Reply Data and Register values, sends Response for Read Single Reg.
void ModBusSlave::Read_Single_Reg(unsigned char *Data_In) // Function code 2
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
//##### Write_Multiple_Reg #####
// Takes:  In Data Buffer, Length
// Returns: Nothing
// Effect: Sets Reply Data and Register values, sends Response for Write Multiple Reg.
void ModBusSlave::Write_Multiple_Reg(unsigned char *Data_In,unsigned short Length)
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
//##### Read_Multiple_Reg #####
// Takes:  In Data Buffer, Length
// Returns: Nothing
// Effect: Sets Reply Data and Register values, sends Response for Read Multiple Reg.
void ModBusSlave::Read_Multiple_Reg(unsigned char *Data_In,unsigned short Length)
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
//##### Write_Coil #####
// Takes:  In Data Buffer, Length
// Returns: Nothing
// Effect: Sets Reply Data and Register values, sends Response for Write Coil.
void ModBusSlave::Write_Coil(unsigned char *Data_In,unsigned short Length)
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
//##### Read_Coil #####
// Takes:  In Data Buffer, Length
// Returns: Nothing
// Effect: Sets Reply Data and Register values, sends Response for Read Coil.
void ModBusSlave::Read_Coil(unsigned char *Data_In,unsigned short Length)
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
//##### Write_Register #####
// Takes:  In Data Buffer, Length
// Returns: Nothing
// Effect: Sets Reply Data and Register values, sends Response for Write Register.
void ModBusSlave::Write_Register(unsigned char *Data_In,unsigned short Length)
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
//##### Read_Register #####
// Takes:  In Data Buffer, Length
// Returns: Nothing
// Effect: Sets Reply Data and Register values, sends Response for Read Register.
void ModBusSlave::Read_Register(unsigned char *Data_In,unsigned short Length)
{
    if(Data_In[0] == 0)
        // If Broadcast Msg, then no reply
        return;
    if(Length > 0)
        // If there is Data to be sent, then...
    {
        unsigned short MyCRC = CheckSum.CRC16(Data_In,Length);
        Data_In[Length++] = MyCRC & 0x00ff;
        byte into Buffer
        Data_In[Length++] = MyCRC >> 8;
        upper byte into Buffer
        for(int C = 0; C < Length;C++)
        {
            Serial.write(Data_In[C]);
        }
    }
}
```

```

unsigned short Addr_Hi = Data_In[2];
unsigned short Addr_Lo = Data_In[3];
unsigned short Value_Hi = Data_In[4];
unsigned short Value_Lo = Data_In[5];

unsigned short Address = (Addr_Lo + (Addr_Hi<<8));
    if(Address >= Coil_Count)                                     // Invalid
Address;
    {
        Error = 3;
        return;
    }
Coils[Address] = (Value_Hi<<8) + Value_Lo;
Send_Response(Data_In,6);
}
//##### Write_Single_Coil #####
// Takes:  In Data Buffer
// Returns: Nothing
// Effect: Writes single bit in Registers. Sets Reply Data and sends Response
void ModBusSlave::Write_Single_Coil(unsigned char *Data_In) // Function code 1
{
    unsigned short Addr_Hi = Data_In[2];
    unsigned short Addr_Lo = Data_In[3];
    unsigned short Value_Hi = Data_In[4];
    unsigned short Value_Lo = Data_In[5];
    unsigned short Address = Addr_Lo + (Addr_Hi<<8);
    unsigned short Write_Address = Address/16;
    unsigned char Write_Bit = Address&0x000F;
        if(Address >= Coil_Count*16)                             // Invalid Address;
        {
            Error = 3;
            return;
        }
    if(Value_Hi>0 | Value_Lo > 0)                                  // Real Protocol requires 0xFF00 = On and 0x0000 =
Off, Custom, using anything other than 0 => ON
    {
        Coils[Write_Address] |= (1<<Write_Bit);
    }
    else
    {
        Coils[Write_Address] &= ~(1<<Write_Bit);
    }
    Send_Response(Data_In,6);
}
//##### Write_Reg #####
// Takes:  In Data Buffer
// Returns: Nothing
// Effect: Writes Register in Registers. Sets Reply Data and sends Response
void ModBusSlave::Write_Reg(unsigned char *Data_In) // Function code 1
{
    unsigned short Addr_Hi = Data_In[2];
    unsigned short Addr_Lo = Data_In[3];
    unsigned short Cnt_Hi = Data_In[4];

```

```

unsigned short Cnt_Lo = Data_In[5];
unsigned char Byte_Count = Data_In[6];
unsigned short Address = (Addr_Lo + (Addr_Hi<<8));
    if(Address >= Coil_Count) // Invalid Address;
    {
        Error = 3;
        return;
    }
unsigned short Read_Byte = 7; // First entry in input Data_In
for(int C = 0; C < Byte_Count;C+=2)
{
    Coils[Address] = (Data_In[Read_Byte]<<8)+Data_In[Read_Byte+1];
    Address+=1;
    Read_Byte+=2;
}
Send_Response(Data_In,6);
}
//##### Write_Coils #####
// Takes: In Data Buffer
// Returns: Nothing
// Effect: Writes multiple bits in Registers. Sets Reply Data and sends Response
void ModBusSlave::Write_Coils(unsigned char *Data_In) // Function code 1
{
    unsigned short Addr_Hi = Data_In[2];
    unsigned short Addr_Lo = Data_In[3];
    unsigned short Cnt_Hi = Data_In[4];
    unsigned short Cnt_Lo = Data_In[5];
    unsigned char Byte_Count = Data_In[6];

    unsigned short Address = Addr_Lo + (Addr_Hi<<8);
    unsigned short Write_Bit_Count = Cnt_Lo + (Cnt_Hi<<8);
    if(Address >= Coil_Count*16) // Invalid Address;
    {
        Error = 3;
        return;
    }

    unsigned short Write_Address = Address/16;
    unsigned char Write_Bit = Address&0x000F;

    unsigned short Read_Byte = 7; // First entry in input Data_In
    unsigned char Read_Bit = 0;
    for(int C = 0; C < Write_Bit_Count;C++)
    {
        if((Data_In[Read_Byte]&(1<<Read_Bit))>0) // If set Bit is a 1, then, set corresponding bit in
        register
        {
            Coils[Write_Address] |= (1<<Write_Bit);
        }
        else // If set bit is a 0,
        clear the bit in the register
        {
            Coils[Write_Address] &= ~(1<<Write_Bit);
        }
    }
}

```

```

    }
    Read_Bit++;
Read Bit
    Write_Bit++;
    if(Read_Bit>=8)
data, so increment every 8th bit
    {
        Read_Bit = 0;
        Read_Byte++;
    }
    if(Write_Bit >= 16)
registers, so increment every 16th bit.
    {
        Write_Bit = 0;
        Write_Address++;
    }
}
Send_Response(Data_In,6);
}
//##### Read Reg #####
// Takes: In Data Buffer
// Returns: Nothing
// Effect: Reads bytes at a time and composes 16 bit replies. Sets Reply Data and sends Response
void ModBusSlave::Read_Reg(unsigned char *Data_In) // Function code 1
{
    unsigned short Addr_Hi = Data_In[2];
    unsigned short Addr_Lo = Data_In[3];
    unsigned short Cnt_Hi = Data_In[4];
    unsigned short Cnt_Lo = Data_In[5];
    //Read date
    unsigned short Byte_Count = Cnt_Lo + (Cnt_Hi<<8);
    Data_In[2] = Byte_Count * 2;

    unsigned short Address = (Addr_Lo + (Addr_Hi<<8));
    if(Address >= Coil_Count) // Invalid Address;
    {
        Error = 3;
        return;
    }
    unsigned short Item = 3;
    for(int Count = 0; Count < Byte_Count; Count++)
    {
        Data_In[Item+1] = lowByte(Coils[Address]);
        Data_In[Item] = highByte(Coils[Address]);
        Address++;
        Item+=2;
    }
    Send_Response(Data_In,Item);
}

//##### Read Coils #####
// Takes: In Data Buffer
// Returns: Nothing

```

```
// Effect: Reads a bit at a time, composing 8 bit replies. Sets Reply Data and sends Response
void ModBusSlave::Read_Coils(unsigned char *Data_In) // Function code 1
{
    unsigned short Addr_Hi = Data_In[2];
    unsigned short Addr_Lo = Data_In[3];
    unsigned short Cnt_Hi = Data_In[4];
    unsigned short Cnt_Lo = Data_In[5];
    //Read date
    unsigned short Bit_Count = Cnt_Lo + (Cnt_Hi<<8);
    unsigned char Byte_Count = 0;
    unsigned char Sub_Bit_Count = 0;
    unsigned char Working_Byte = 0;
    unsigned short Address = Addr_Lo + (Addr_Hi<<8);
    if(Address >= Coil_Count*16) // Invalid Address;
    {
        Error = 3;
        return;
    }
    unsigned short Item = 3;
    for(int Bit = 0; Bit < Bit_Count; Bit++)
    {
        Working_Byte = Working_Byte | ((Coils[Address/16]>>(Address&0x000f))<<Sub_Bit_Count);
        Address++;
        Sub_Bit_Count++;
        if(Sub_Bit_Count >=8)
        {
            Data_In[Item] = Working_Byte;
            Working_Byte = 0;
            Sub_Bit_Count=0;
            Byte_Count++;
            Item++;
        }
    }
    //If not a full byte of info
    if(Sub_Bit_Count != 0)
    {
        Data_In[Item] = Working_Byte;
        Working_Byte = 0;
        Sub_Bit_Count=0;
        Byte_Count++;
        Item++;
    }
    Data_In[2] = Byte_Count;
    Send_Response(Data_In,Item);
};
```


APPENDIX F – Modbus_Slave.h

```
/*
    ModBusSlave.cpp is a library for the Arduino System. It complies with ModBus protocol, customized for
    exchanging
    information between Industrial controllers and the Arduino board.

    Copyright (c) 2012 Tim W. Shilling (www.ShillingSystems.com)
    Arduino Modbus Slave 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.

    Arduino Modbus Slave 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.

    To get a copy of the GNU General Public License see <http://www.gnu.org/licenses/>.
*/
#ifndef Modbus_Slave_h
#define Modbus_Slave
#include "WProgram.h"
class ModBusSlave
{
    public:
        ModBusSlave(unsigned char _Slave, unsigned short *_Coils, unsigned short _Coil_Count);
        void Process_Data(unsigned char *Buffer, unsigned char Count);
        unsigned char Slave;
        unsigned short * Coils;
        unsigned char Error;
        unsigned short Coil_Count;
    private:
        void Send_Response(unsigned char *Data_In, unsigned short Length);
        void Read_Exception(unsigned char *Data_In); // Function code 7
        void Write_Single_Reg(unsigned char *Data_In); // Function code 1
        void Write_Single_Coil(unsigned char *Data_In); // Function code 1
        void Write_Reg(unsigned char *Data_In); // Function code 1
        void Write_Coils(unsigned char *Data_In); // Function code 1
        void Read_Reg(unsigned char *Data_In); // Function code 1
        void Read_Coils(unsigned char *Data_In); // Function code 1
};
#endif
```