MODBUS TCP IN A SCRIPT

1. Introduction

Senquip devices have a built in Modbus RTU peripheral that works over RS232 or RS485. This allows the Senquip devices to connect to multiple slave Modbus sensors. In some applications, a remote server or Modbus master may want to read data from a Senquip device as a slave. This can be useful as Senquip devices can connect to a multitude of non-Modbus sensors, and can through a script, present the sensor data in a structured way to an external Modbus master.

The Modbus master could be a serial device (Modbus RTU) or a remote device operating on the same network as the Senquip device (Modbus TCP). This application note describes how to implement Modbus TCP in a script.

Modbus TCP (or Modbus TCP/IP) allows Modbus devices to communicate over a network, making it easier to connect devices over longer distances or to connect to devices over a network. Modbus TCP is commonly used in industrial automation and control systems to connect devices such as PLCs, HMIs, and sensors.

It is assumed that the user has Admin privileges and scripting rights for the device being worked on. To request scripting rights, contact support@senquip.com.

2. References

The following documents were used in compiling this Application Note.

Reference	Document	Document Number
A	Acromag Introduction to MODBUS TCP	8500-765-A05C000
B	Modbus Register Addressing	Modbus Register Addressing, Continental Control Systems
C	Modbus 101 – Introduction to Modbus	Modbus 101 - Introduction to Modbus, Control Solutions, Minnesota
D	Modbus	Modbus, Wikipedia

3. MODBUS TCP

Modbus TCP is simply the Modbus RTU protocol with a TCP interface that runs on a network.

The Modbus messaging structure is the application protocol that defines the rules for organising and interpreting the data independent of the data transmission medium.

TCP/IP refers to the Transmission Control Protocol and Internet Protocol, which provides the transmission medium for Modbus TCP messaging.

Simply stated, TCP/IP allows blocks of binary data to be exchanged between computers. It is also a world-wide standard that serves as the foundation for the World Wide Web. The primary function of TCP is to ensure that all packets of data are received correctly, while IP makes sure that messages are correctly addressed and routed. Note that the TCP/IP combination is merely a transport protocol, and does not define what the data means or how the data is to be interpreted (this is the job of the application protocol, Modbus in this case).

In practice, Modbus TCP embeds a standard Modbus data frame into a TCP frame, without the Modbus checksum, as shown in Figure 1.

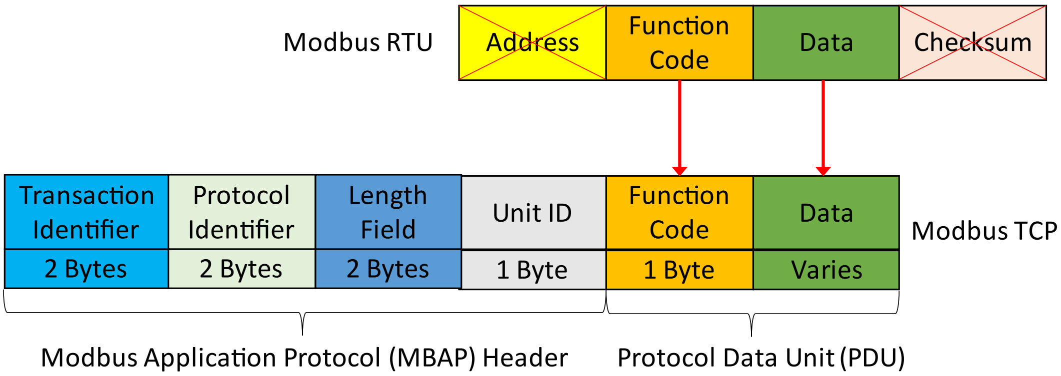

Figure 1 - Construction of Modbus TCP Frame

The Modbus function code and data are encapsulated into the Modbus TCP packet without modified. The Modbus error checking field (checksum) is not used as the standard TCP/IP link layer guarantees data integrity. The Modbus address field is no longer needed as the IP address of the device fulfils the function of uniquely identifying the Modbus TCP device.

From the figure, we see that the function code and data fields are absorbed in their original form. Thus, a Modbus TCP packet or Application Data Unit (ADU) takes the form of a 7-byte Modbus Application Protocol (MBAP) header (transaction identifier + protocol identifier + length field + unit identifier), and the protocol data unit (PDU) (function code + data). The MBAP header is 7 bytes long and includes the following fields:

- Transaction Identifier (2 Bytes): This identification field is used for transaction pairing when multiple messages are sent along the same TCP connection by a client without waiting for a prior response.
- Protocol Identifier (2 bytes): This field is always 0 for Modbus services and other values are reserved for future extensions.
- Length (2 bytes): This field is a byte count of the remaining fields and includes the unit identifier byte, function code byte, and the data fields.
- Unit Identifier (1 byte): This field is used to identify a remote server located on a non-TCP network (for serial bridging). In a typical Modbus TCP/IP server application, the unit ID is set to 00 or FF, ignored by the server, and simply echoed back in the response.

The PDU is made up of the following fields:

- Function Code (1 byte): Tells the slave device what kind of action to perform.
- Data (4 bytes): The start address of the register to be read (2 bytes) and the number of registers to read (2 bytes).

Function Code	**Function**	**Size**	**Access**
0x01 =01	Read coil	8 bits	Read

0x02 = 02	Read discrete	8 bits	Read only
0x03 = 03	Read unsigned holding	16 bits	Read
0x04 = 04	Read unsigned input	16 bits	Read only
0x05 = 05	Write coil	8 bits	Write
0x06 = 06	Write unsigned holding	16 bits	Write

Table 1 – Commonly Used Function Codes

The different fields of the of the Modbus TCP/IP ADU are encoded in Big Endian format. This means that the most significant byte in the sequence is stored at the lowest storage address (i.e., it is first). An example of reading 2 holding registers, starting at address 3, is given in Figure 2.

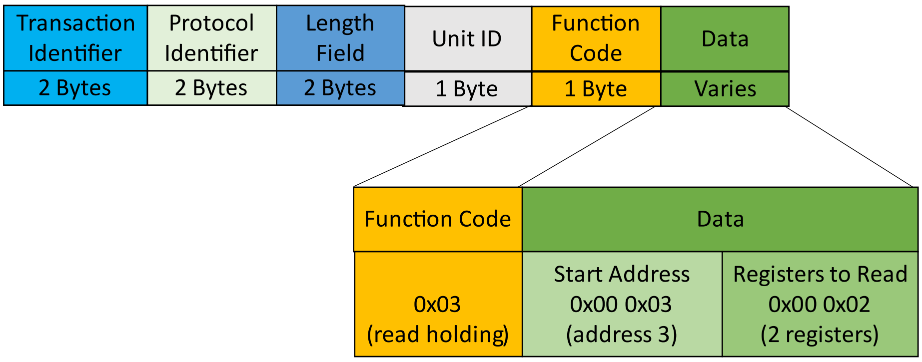

Figure 2 – Example Modbus TCP Read

The complete Modbus TCP Application Data Unit is embedded into the data field of a standard TCP frame and sent via TCP to well-known system port 502, which is specifically reserved for Modbus applications. Modbus TCP clients and servers listen and receive Modbus data via port 502.

An example of reading 2 holding registers, starting at address 3, and the response is shown in Figure 3.

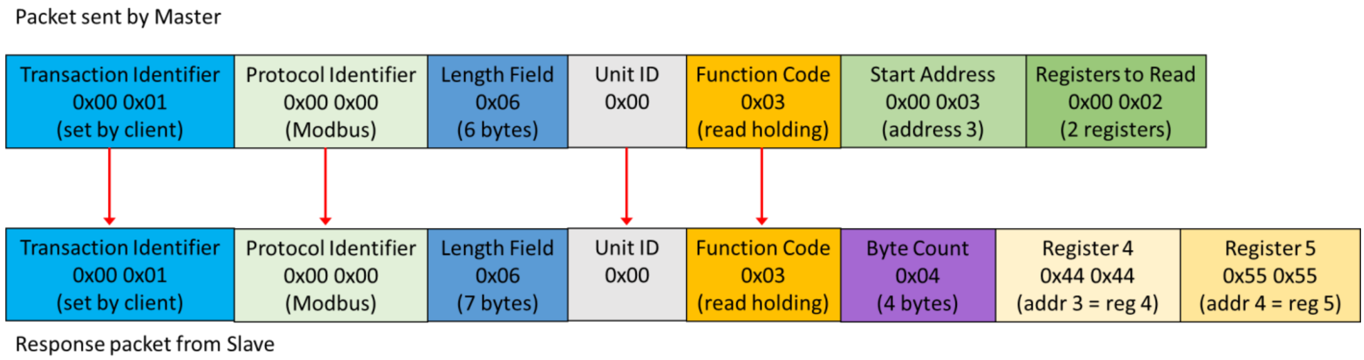

Figure 3 - Read Holding Register Example

For a refresher on the Modbus protocol, see references B, C, D.

4. Test Setup

In this application note, the Senquip device resides on the same Wi-Fi network as the Modbus master. The Senquip device (slave) will implement a TCP server and will wait for the remote TCP client (Modbus master) to connect. Once connected, the master will request Modbus data by sending Modbus TCP packets to the slave Senquip device. The Senquip device will respond to the master with the requested data.

Figure 4 - Test Setup

The Modbus master is implemented on a computer running [Modbus Master Tool]() from [ICP DAS]().

Figure 5 - Easy to Use Modbus TCP Master

An RS232 to USB converter is attached to the Senquip ORB to allow for debugging of the application. [Realterm]() is used as a terminal program, to receive serial messages from the ORB and to send serial message to the ORB.

5. A Note on Security

The devices in this network are on a secure private Wi-Fi network. If operation is required on a cellular or other public network, the security of the connection needs to be considered. It is not recommended that a TCP or other server be left operating on a Senquip device on a public network as it leaves the device and rest of network vulnerable to attack.

6. Implementation

We will implement a Modbus TCP slave on a Senquip ORB running the latest SFW002 operating system. The system will have the following capabilities:

- 8 coils holding 8 output values.

- 8 holding registers (16 bit).
- 8 input registers (16 bit).

The following functions are supported:

- 0x01 - read coil.
- 0x03 - read holding.
- 0x04 - read input.
- 0x05 write coil.
- 0x06 - write holding.

7. Script Development

The Modbus TCP application will be developed using the Senquip scripting language, mJS which is a restricted version of JavaScript. For more information on the libraries and other commands used in this application, see the Senquip Scripting Guide. The full application is given in Appendix 1. It is suggested that the Senquip device be upgraded to the latest firmware, and that a factory reset be performed before starting the project. The factory reset resets the settings file to default, removing any settings from previous projects.

The following library files are required:

- senquip.js: Functions for interacting with data collection and the device's peripherals.
- config.js: Functions for interacting with data collection and the device's peripherals.
- serial.js: Functions for reading/writing serial data over the RS232/RS485 interface.
- net.js: Low-level network configuration API.
- math.js: Adds additional mathematical functions.

```
load('senquip.js');
load('api_config.js');
load('api_serial.js');
load('api_net.js');
load('api_math.js');
```

The coil, holding, and input registers are defined as byte arrays. An *error* variable is used to store feedback when an error occurs. Errors will be sent to the Senquip Portal. Received TCP packets are held in the variable *receive*.

```
let coil = [1,1,0,0,0,1,1,1];               // 8 coil registers, each 1 bit
let holding = [10,11,12,13,14,15,16,17]; // 8 holding registers each 16 bit
let input   = [20,21,22,23,24,25,26,27]; // 8 input registers each 16 bit
let error = "Reset";   // holds error code
let receive = ""; // holds received TCP packet
```

The NET.server function sets up a TCP server on port 502. Optional functions onconnect, ondata, and onerror are called on connection, when data is received, and if an error occurs. All received messages are expected to be 12 bytes. Once 12 bytes has been received, the message is sent to be parsed. The parse function returns the required Modbus return string which is sent back to the requesting Modbus master. The result string is echoed to then serial port for debug purposes. This function should be further developed to handle error cases.

```
// TCP handler
Net.serve({
    addr: 'tcp://502', // standard port for MODBUS
    onconnect: function(conn) {
    receive = "";                       // Clear receive buffer
    },
    ondata: function(conn, data) {
      receive = receive + data;
      Net.discard(conn, data.length);  // Discard received data
      if(receive.length >= 12){
        let result = parse(data);
        Net.send(conn, result);              // Echo received data back
        SERIAL.write(1,result,result.length); // send to serial port for debug
        receive = "";
      }
    },
    onerror: function(conn) {
    },
  });
```

The SERIAL.handler function is required if serial communications are used in a script.

```
SERIAL.set_handler(1, function(channel) {
}, null);
```

The serial port settings are inherited from the device settings and is configured as scripted operation, RS232 115200 baud, 8N1.

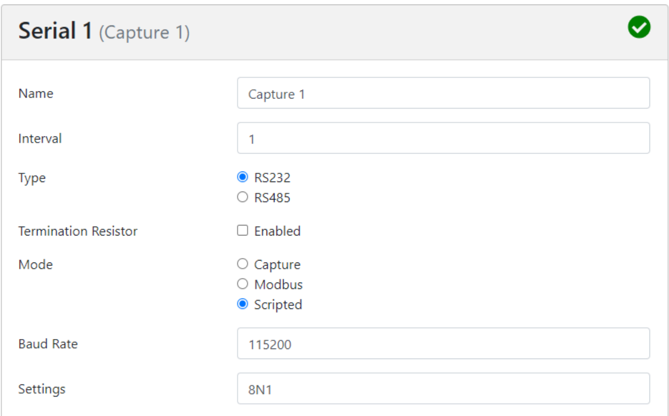

Figure 6 - Senquip Device Serial Port Settings

When a Modbus request is received, the packet is sent to a function parse() to be interpreted. Data will typically be received over TCP, but a serial receive function has been left in the application to allow testing from a serial terminal. The data that arrives is a string of bytes. The string is broken up into the respective fields using a slice command. Most fields will simply be echoed back, but some need to be analysed and so they are converted from strings to numbers using the at command that returns the numeric byte value at given string index.

For example, for a start address of 0x00 0x03, the string will contain the ascii characters for 0 and 3. The .at() command converts returns 0 for the high digit and 3 for the low digit. Since the full range of values available is 0x00 to 0xFF, the high value is multiplied by 256 and added to the low value.

```
// Parses the MODBUS request and if valid, generates a response
function parse(packet){

  let s = ""; // return packet
  let ti = packet.slice(0,2); // Transaction Identifier
  let pi = packet.slice(2,4); // Protocol Identifier
  let lf = packet.slice(4,6); // Length Field
  let ui = packet.slice(6,7); // Unit ID
  let ui1 = ui.at(0);
  let fc = packet.slice(7,8); // Function Code
  let data = packet.slice(8,packet.length);
  let sa = data.slice(0,2); // Register start address
  let sa1 = 256*sa.at(0)+sa.at(1); // Register start address as a number
  let nr = data.slice(2,4); // Number of registers to read
  let nr1 = 256*nr.at(0)+nr.at(1); // Number of registers as a number
```

A check is done to ensure that the Unit ID is correct. The Unit ID is read from custom variable 1 using the *Cfg.get* function. Custom variables allow users to change values in scripts without having to have access to the script. They are setup along with custom parameters in the scripting window and are accessed on the custom variable tab of the settings. In this case, the Unit ID is 55.

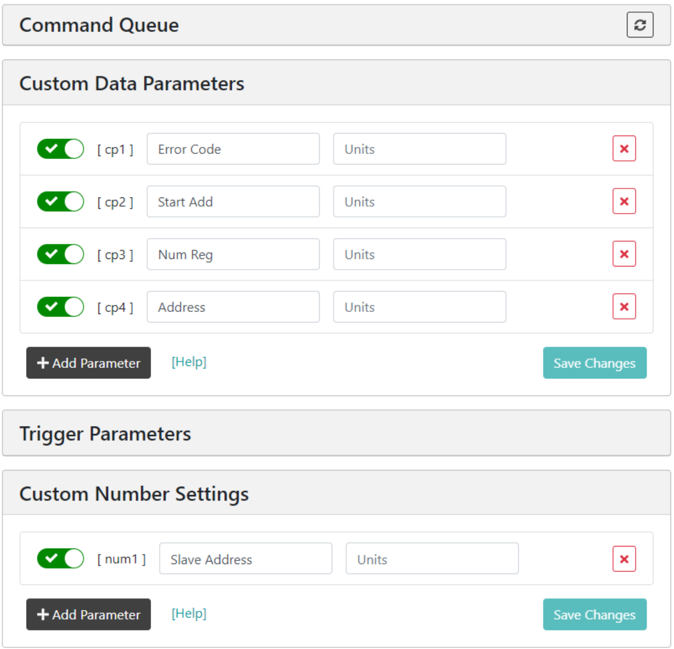

Figure 7 - Setting up Custom Parameters and a Custom Variable

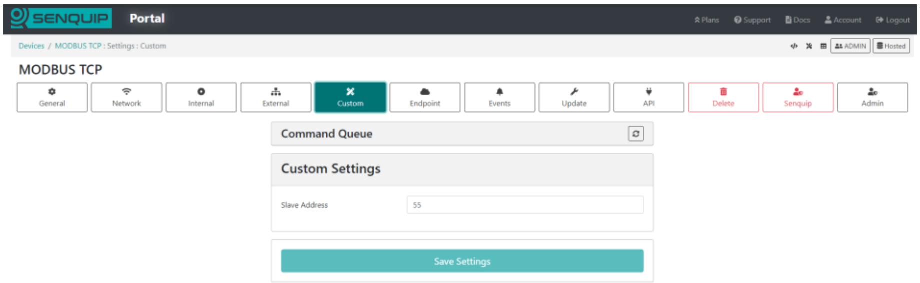

Figure 8 - Setting the Unit ID to 55

Some range checking is done on the start address and number of registers to be read. In each case, a descriptive serial message is sent to facilitate debugging. This could be further developed.

```
if (ui1 !== Cfg.get('script.num1')){
      error = "Incorrect MODBUS address";
      return "";
    }

if (sa1 > 7){
  error = "No such register";
  return "";
}
 if ((sa1+nr1) > 8 && (fc === "\x01" || fc === "\x04" || fc === "\x04")){
  error = "Register count too high";
  return "";
}
```

We now look at the function codes to determine what data should be returned to the Modbus master. If the function code is 1, then we return the coils requested in the request. Notice that the variable fc is still a string and so is compared with the escaped ASCII character for 0x01. The length field (lf2 = 4) and the number of bytes (np2 = 1) are always constant as there is ever only 1 register returned by a read coil request. The coil register *c* is assembled in a loop where for each coil requested, the value associated with that coil is added.

The Modbus return string (s) is assembled from the individual fields, ready to be returned via TCP to the master.

```
// read coil
if (fc === "\x01"){
  let lf2 = 4; // ui + fc + byte count + return bytes always 1
  let nr2 = 1; // always 1 byte returned
  let c = 0; // store in which to calculate return
  for (let i = sa1+nr1-1; i >= sa1; i--) {
    c = c + coil[i]*Math.pow(2,i);
  }
  s = ti+pi+chr(lf2/256)+chr(lf2%256)+ui+fc+chr(nr2)+chr(c);
}
```

The operation for function codes 3 and 4 are very similar to each other. The length field is calculated as 3 plus 2 times the number of 2-byte registers requested. The byte count is 2 times the number of registers to be returned.

Most of the return string is assembled and then a loop is used to add the high and low bytes of each register. The chr function is used to generate the ascii character for the high and low bytes of each register.

```
// read holding
else if (fc === "\x03"){
  let lf2 = 3+2*nr1; // ui + fc + byte count + return bytes
  let nr2 = 2*nr1; // 2 times as manybytes in return string as each register 2 bytes
  s = ti+pi+chr(lf2/256)+chr(lf2%256)+ui+fc+chr(nr2);  // start creating the response string
  for (let i = sa1; i < sa1+nr1; i++) { // add the registers to the return string
    s = s + chr(holding[i]/256)+chr(holding[i]%256);
  }
}

// read input
else if (fc === "\x04"){
  let lf2 = 3+2*nr1; // ui + fc + byte count + return bytes
  let nr2 = 2*nr1; // 2 times as manybytes in return string as each register 2 bytes
  s = ti+pi+chr(lf2/256)+chr(lf2%256)+ui+fc+chr(nr2);  // start creating the response string
  for (let i = sa1; i < sa1+nr1; i++) {  // add the registers to the return string
    s = s + chr(input[i]/256)+chr(input[i]%256);
  }
}
```

Writing to coil and holding registers is simple. The command to set a coil is 0xFF00 and to clear a coil is 0x0000. The set and clear command is stored in the field normally reserved for the number of registers to read.

When writing to a holding register, the data to be written is held in the field normally reserved for the number of registers to read.

In both cases the request string is echoed as a response.

If an invalid function code is received, an error is raised.

```
// write coil
else if (fc === "\x05"){
    s = packet;  // echo the received packet as a respose
    if (nr1 === 0xFF00){ // in this case, nr1 contains the instruction to set the bit
      coil[sa1] = 1; // set single bit
    }
    else if (nr1 === 0x0000){ // in this case, nr1 contains the instruction to clear the bit
      coil[sa1] = 0; // clear single bit
    }
}

// write holding
else if (fc === "\x06"){
   s = packet;  // echo the received packet as a respose
   holding[sa1] = nr1; // in this case nr1 holds the data to be written
}
```

The server function run independently of the main data handler that runs on each base interval. In this case, all the main handler does is to read the serial buffer and dispatch a few variables to the Senquip Portal. If the serial buffer contains a 12 byte long request, then it is sent to be parsed in the same way that data arriving over TCP is parsed. This was handy when testing the parse routine.

Any errors, the register start address, number or registers to read, and the Unit ID (Modbus address) are sent to the Senquip Portal for diagnostics.

In a real application, the coil, input and holding registers would likely be set in the data handler, based on measurements taken by the Senquip device and contained in the structure *obj*.

```
SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  let test = SERIAL.read(1);
  if (test.length === 12){
        parse(test);
  }


  SQ.dispatch(1,error);
  SQ.dispatch(2,sad);
  SQ.dispatch(3,nrd);
  SQ.dispatch(4,uid);
  error = "";

}, null);
```

8. Testing

Testing was performed with the Modbus master tool. A connection is established at the IP address of the Senquip device and on port 502. A scan time of 1 second was set arbitrarily. A more sensible scan time would be not faster than the update rate of the Senquip device (5 seconds in this case).

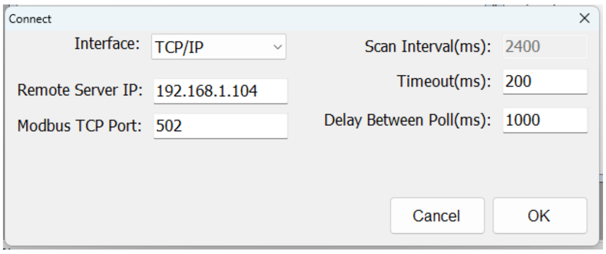

Figure 9 - Establishing a TCP Connection

The IP address was obtained from the Device Info widget on the Senquip Portal. In a real application, the device may be allocated a fixed IP address.

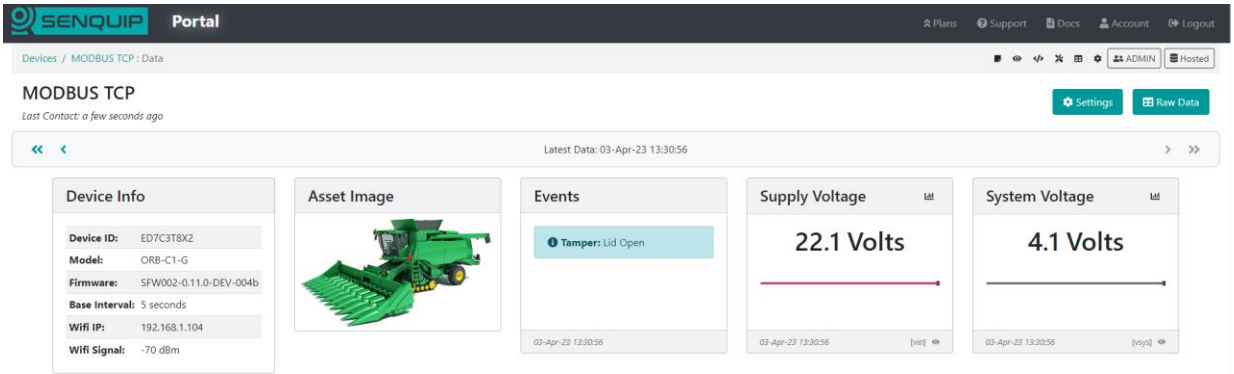

Figure 10 - Obtaining the Senquip Device IP Address

Reads were configured for different numbers of registers. In each case, the Unit ID was specified as 55.

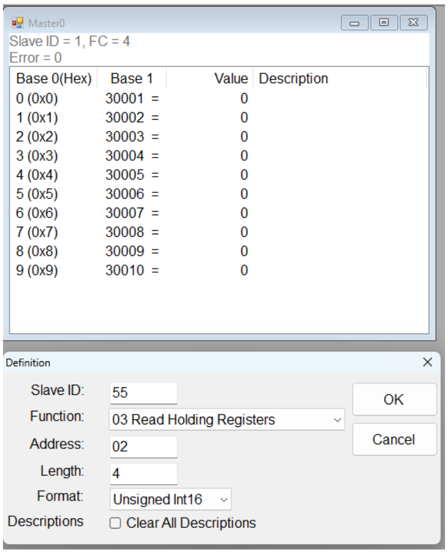

Figure 11 - Setting up a Register Read

The results were checked against the values loaded into the registers at boot of the Senquip device.

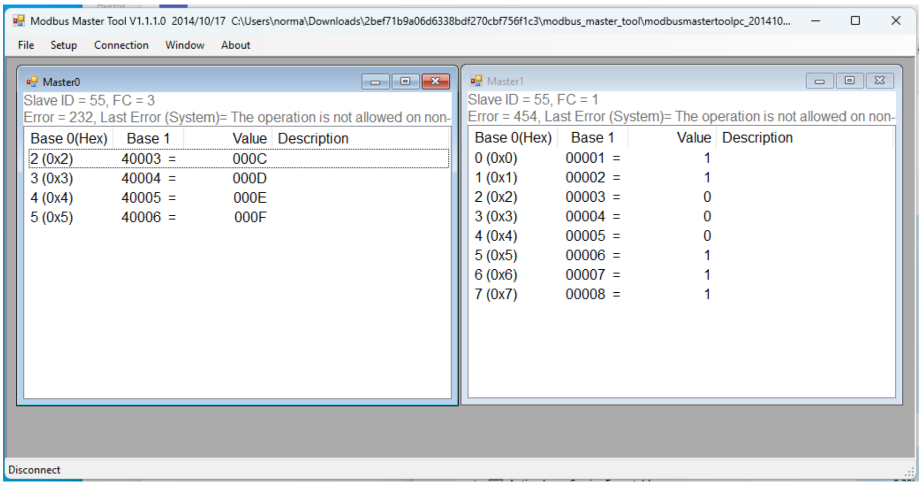

Figure 12 - Reading 4 Holding Registers Starting at Address 2

Writing to coils and holding registers was tested by clicking on values to write to them and then checking the subsequent reads returned the updated data.

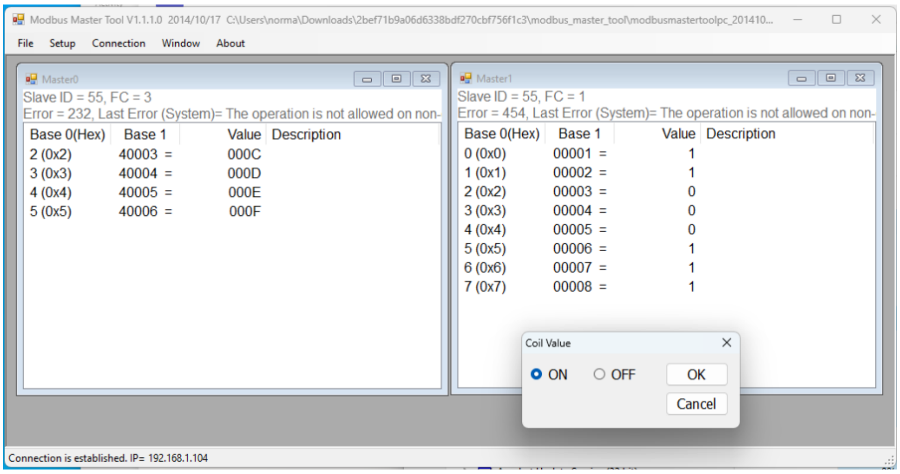

Figure 13 - Setting a Coil

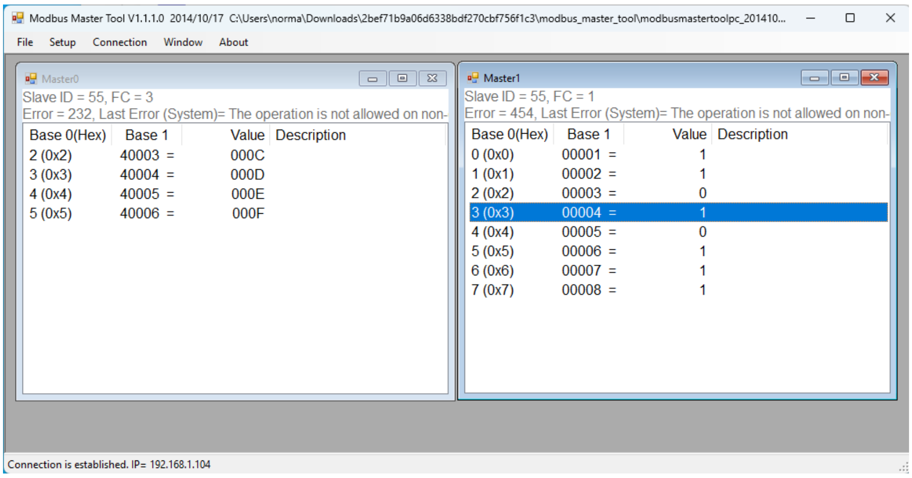

Figure 14 - Reading the Updated Coil Value

9. Conclusion

A Modbus TCP slave that supports read coil, read holding, read input, write coil, and write holding has been developed using the Senquip scripting language and library files. The application has been tested using Modbus Master Tool from ICP DAS.

Appendix 1: Source Code

```
/* This implementation of MODBUS TCP supports the following function codes:
0x01 - read coil
0x03 - read holding
0x04 - read input
0x05 write coil
0x06 - write holding
*/
load('senquip.js');
load('api_config.js');
load('api_serial.js');
load('api_net.js');
load('api_math.js');

let nrd = 0;
let sad = 0;
let uid = 0;

let press1 = 0;
let press2 = 0;
let pulses = 0;

let coil = [1,1,0,0,0,1,1,1];             // 8 coil registers, each 1 bit
let holding = [10,11,12,13,14,15,16,17]; // 8 holding registers each 16 bit
let input   = [20,21,22,23,24,25,26,27]; // 8 input registers each 16 bit
let error = "Reset";   // holds error code
let receive = ""; // holds received TCP packet

// TCP handler
Net.serve({
    addr: 'tcp://502', // standard port for MODBUS
    onconnect: function(conn) {
```

```
    receive = "";                           // Clear receive buffer
    },
      ondata: function(conn, data) {
      receive = receive + data;
      Net.discard(conn, data.length);  // Discard received data
      if(receive.length >= 12){
        let result = parse(data);
          Net.send(conn, result);             // Echo received data back
        SERIAL.write(1,result,result.length); // send to serial port for debug
        receive = "";
      }
    },
      onerror: function(conn) {
    },
  });


// Required when using scripted serial
SERIAL.set_handler(1, function(channel) {
}, null);


// Parses the MODBUS request and if valid, generates a response
function parse(packet){

  let s = ""; // return packet
  let ti = packet.slice(0,2); // Transaction Identifier
  let pi = packet.slice(2,4); // Protocol Identifier
  let lf = packet.slice(4,6); // Length Field
  let ui = packet.slice(6,7); // Unit ID
  let ui1 = ui.at(0);
  let fc = packet.slice(7,8); // Function Code
  let data = packet.slice(8,packet.length);
```

```
let sa = data.slice(0,2); // Register start address
let sa1 = 256*sa.at(0)+sa.at(1); // Register start address as a number
let nr = data.slice(2,4); // Number of registers to read
let nr1 = 256*nr.at(0)+nr.at(1); // Number of registers as a number

nrd = nr1; // just for test
sad = sa1;
uid = ui1;

if (ui1 !== Cfg.get('script.num1')){
      error = "Incorrect MODBUS address";
      return "";
    }

if (sa1 > 7){
  error = "No such register";
  return "";
}
 if ((sa1+nr1) > 8 && (fc === "\x01" || fc === "\x04" || fc === "\x04")){
  error = "Register count too high";
  return "";
}

// read coil
if (fc === "\x01"){
  let lf2 = 4; // ui + fc + byte count + return bytes always 1
  let nr2 = 1; // always 1 byte returned
  let c = 0; // store in which to calculate return
  for (let i = sa1+nr1-1; i >= sa1; i--) {
    c = c + coil[i]*Math.pow(2,i);
  }
  s = ti+pi+chr(lf2/256)+chr(lf2%256)+ui+fc+chr(nr2)+chr(c);
}
```

```
// read holding
else if (fc === "\x03"){
  let lf2 = 3+2*nr1; // ui + fc + byte count + return bytes
  let nr2 = 2*nr1; // 2 times as manybytes in return string as each register 2 bytes
  s = ti+pi+chr(lf2/256)+chr(lf2%256)+ui+fc+chr(nr2);  // start creating the response string
  for (let i = sa1; i < sa1+nr1; i++) { // add the registers to the return string
    s = s + chr(holding[i]/256)+chr(holding[i]%256);
  }
}

// read input
else if (fc === "\x04"){
  let lf2 = 3+2*nr1; // ui + fc + byte count + return bytes
  let nr2 = 2*nr1; // 2 times as manybytes in return string as each register 2 bytes
  s = ti+pi+chr(lf2/256)+chr(lf2%256)+ui+fc+chr(nr2);  // start creating the response string
  for (let i = sa1; i < sa1+nr1; i++) {  // add the registers to the return string
    s = s + chr(input[i]/256)+chr(input[i]%256);
  }
}

// write coil
else if (fc === "\x05"){
   s = packet;  // echo the received packet as a respose
   if (nr1 === 0xFF00){ // in this case, nr1 contains the instruction to set the bit
     coil[sa1] = 1; // set single bit
   }
   else if (nr1 === 0x0000){ // in this case, nr1 contains the instruction to clear the bit
     coil[sa1] = 0; // clear single bit
   }
}

// write holding
```

```
  else if (fc === "\x06"){
    s = packet;  // echo the received packet as a respose
    holding[sa1] = nr1; // in this case nr1 holds the data to be written
  }

  // error
  else {
    error = "No such function code";
    return;
  }
  return s;
}

SQ.set_data_handler(function(data)  {
  let obj = JSON.parse(data);

  let test = SERIAL.read(1);
  if (test.length === 12){
        parse(test);
  }


  SQ.dispatch(1,error);
  SQ.dispatch(2,sad);
  SQ.dispatch(3,nrd);
  SQ.dispatch(4,uid);
  error = "";

}, null);
```