Application Protocol for Veris E30 Panel-board Monitoring System

Jaein Jeong
Computer Science Division, UC Berkeley

September 10, 2009

1 Introduction

Veris E30 is a panel or branch-level monitoring instrument that can measure electrical characteristics such as voltage, current, power, energy and power factor at mains or branch level and provide data connectivity. To measure electrical characteristics, Veris E30 uses a collection of current sensors, called current transformers (CT). This allows a non-invasive measurement of current at multiple circuit breakers. To provide data connectivity, Veris E30 maps these electrical characteristics to a list of registers and exposes them through Modbus RTU protocol over RS485 serial bus. A PC or any other computing device connected to an E30 node over the serial bus can access these memory-mapped registers by issuing register-read or register-write request messages. Figure 1 shows a typical configuration for Veris E30 panel monitoring system. In the rest of this document, we describe how to use sensing and data interfaces of Veris E30.

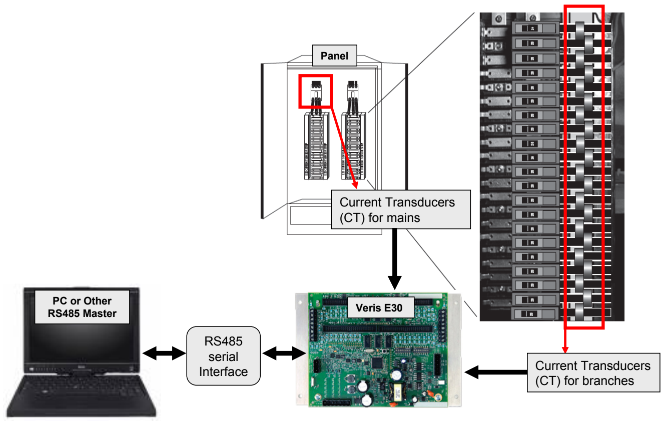

Panel
Current Transducers (CT) for mains
PC or Other RS485 Master
RS485 serial Interface
Veris E30
Current Transducers (CT) for branches

Figure 1: A typical configuration for Veris E30 panel monitoring system

2 Sensing Interface

Veris E30 provides sensing interfaces such as branch-CT board (branch-level current), mains-CT (mains-level current), and voltage terminal (voltage). Before they are used, they need to be configured for correct operation.

Installation Mode

E30 uses CT arrays for monitoring current at branch level. A CT array consists of 21 CTs and up to 4 CT arrays can be connected to an E30 node through ribbon cables, providing maximum 84 channels. One thing to note for branch-level current monitoring is the installation mode, which determines the mapping between physical and logical channels. This is to provide a single logical ordering over multiple CTs while there are four different combinations of layout: top-feed, bottom-feed, sequential, and even/odd. The installation mode can be read or set by accessing register 6 of E30 as shown in Figure 2(a).

CT Scale

For each channel of branch-CT and mains-CT, E30 associates CT scale, which is a calibration factor for the CT reading. The scaling factor for a branch-CT of E30 is a fixed value 20 (20A), whereas the scaling factor for a mains-CT is adjustable. The default value is 200 (200A) and we have used a mains-CT of 300 (300A). Figure 2(b) describes this.

Alarms

E30 allows setting an alarm when a current or voltage measurement channel goes above a certain threshold (Figure 2(c)). In order to use alarms, one needs to set up thresholds and delay timers for the alarms to trigger. *E30 Commissioning Guide* [6] explains details on how to set these up, and *E30 Series Modbus Point Map* [7] lists registers to be configured for setting alarms.

Demand

E30 maintains *demand*, which is statistics of current and power over the recent time window. Demand can be configured by setting the number of sub-intervals (register 71) and sub-interval length (register 72). Refer to [6, 7] for further details.

Register Map

E30 Series Modbus Point Map [7] lists all the registers used by Veris E30. You can access these registers by issuing register-read or register-write commands. One thing to note is that the integer address shown in this point map starts from 1, whereas the address used in a MODBUS command starts from 0:

$$\text{Address in a MODBUS command} = \text{Address in the point map} - 1$$

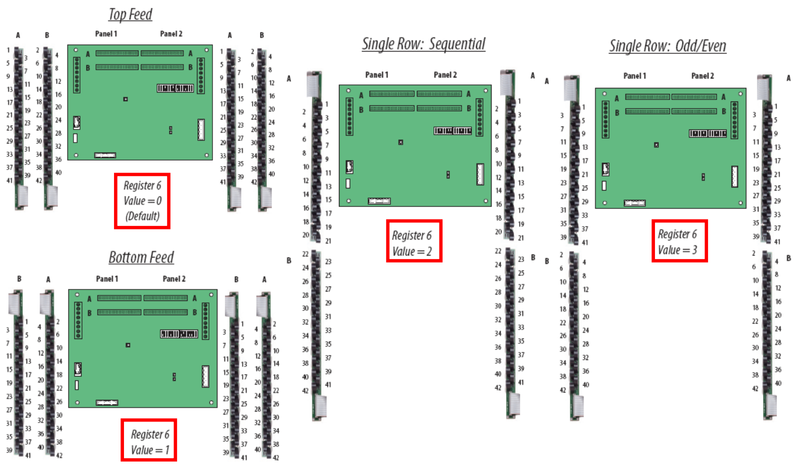
For example, register 6, which is the register that sets the installation mode of an E30 node, has address of 5 in a register-read or register-write command.

3 Communication Interface - Physical and Link Level Interface

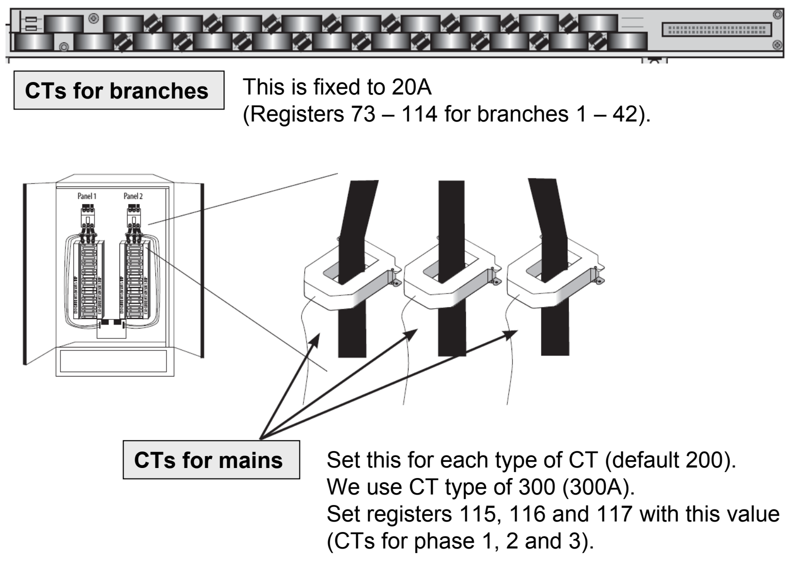
At physical and link level, Veris E30 uses RS485 multi-drop serial communication specification. A typical network consists of one or more RS485 slaves, an RS485 master and an RS485 serial interface (Figure 3).

RS485 Slave

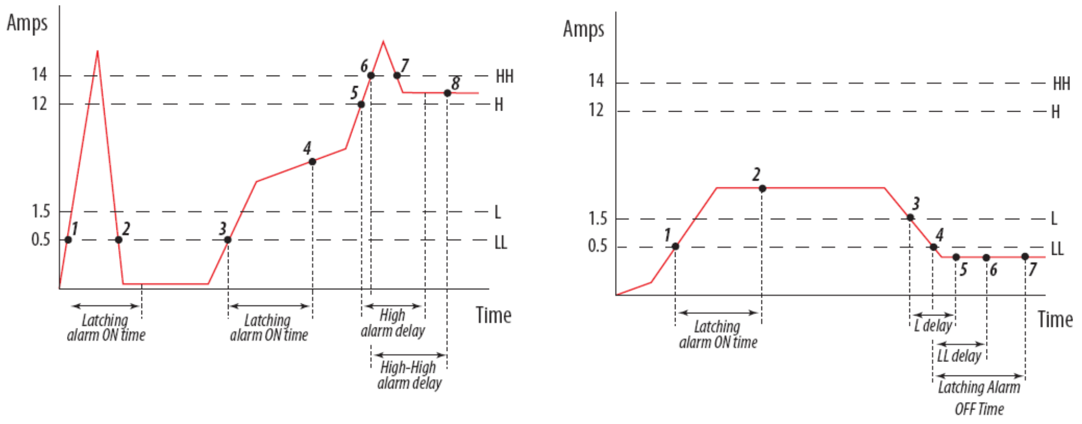
A Veris E30 device that can receive a request message and transmit a reply message. Since RS485 specification allows multiple RS485 slaves, each RS485 slave in a network should have a unique 8-bit address between 1 and 246 [5], and E30 provides an 8-bit dip switch for this purpose. Depending on whether there is a single or multiple RS485 slaves, an RS485 slave is connected either directly or daisy-chained to an RS485 master through an RS485 serial interface.



(a) Installation mode depending on the layout of branch-CTs


CTs for branches
This is fixed to 20A
(Registers 73 – 114 for branches 1 – 42).
CTs for mains
Set this for each type of CT (default 200).
We use CT type of 300 (300A).
Set registers 115, 116 and 117 with this value
(CTs for phase 1, 2 and 3).

(b) CT scale



(c) Alarm thresholds and delay timers

Figure 2: Sensing interface for Veris E30

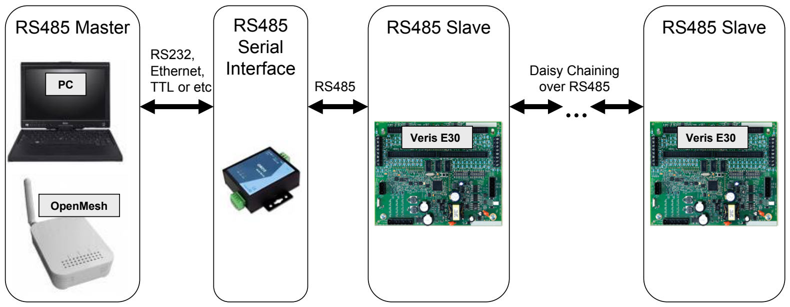


Figure 3: Physical and Link Level Interface

Table 1: Parameters of GW215 RS485-to-Ethernet Converter

Parameter		**Value**
Network protocol		TCP server
Serial port	baud rate	19200 bps
	parity	none
	data bits	8 bits
	stop bits	1 bit
	flow control	none
Inter-character time gap		10 msec
2/4 wire selection		2 wires

RS485 Master

A PC or other computing device that transmit a request and receive a reply message through its RS232, Ethernet or TTL interface. Since there is only a single RS485 master in a network, an RS485 master does not have an address unlike an RS485 slave device.

RS485 Serial Interface

While an RS485 is widely used in industrial automation, it is not directly supported by most PCs or other computing devices. An RS485 serial interface converts the RS485 communication interface to a more widely available interface such as RS232 or Ethernet. We have used a GW215 node as an RS485-to-Ethernet interface. There are several parameters to configure with the RS485-to-Ethernet interface and these are listed in Table 1. More details on how to set these parameters are explained in *GW215 Ethernet-Serial Server* [4].

Wiring

Modbus over Serial Line Specification and Implementation Guide [3] specifies that the RS485 slaves and serial interface should be wired using a 2-wire or 4-wire twisted pair cable (other than shielding / ground wire). We have used three wires of a CAT-5 cable for this purpose (Data+, Data-, Ground). As for the wiring of an RS485 serial bus, we also should consider line termination. [3] recommends that either end of an RS485 serial bus should be terminated with line terminators (150 Ohm resistor), and [4] recommends using a line terminator on the GW215 Ethernet-Serial Converter only when the serial bus spans long distance.

We have observed that our experimental setup of E30 successfully operated over the RS485 serial bus. This experimental setup has the following configuration:

- Length: 6-ft long.
- Line terminator: 150 Ohm resistor used on RS485 slaves (Veris E30) not on the RS485-to-Ethernet interface (GW215).

4 Communication Interface - Application Level Interface

4.1 Message Format

Once physical interface is set up, a Modbus client (RS485 master) can communicate with one or more Modbus servers (RS485 slaves) using Modbus application protocol over TCP/IP. Whether it is a request message (client to server) or reply message (server to client), a Modbus frame consists of slave address, function code, data and CRC as in the following table:

Slave Address	Function Code	Data	CRC
1 byte	1 byte	0 up to 252 byte(s)	2 bytes CRC Low, CRC Hi

Slave Address

Slave address is an 8-bit field that represents a unique address of an RS485 slave node within a bus. For E30, valid address is between 1 and 246. Slave address is used for both request and reply messages. This slave address should match the address set by the DIP switch of an E30 device.

Function Code

Function code is an 8-bit field that describes the type of operation. For example, Veris E30 supports *register read* (0x03), *single register write* (0x06), *multiple register write* (0x10), and *query ID* (0x11) commands. Details on these function codes are shown at Table 2.

Data

Data is an 0 to 252 byte field that depends on function code.

CRC

CRC is a 16-bit CRC checksum over slave address, function code and data. An implementation of CRC is described in [3].

4.2 Reference Implementation

C programs that demonstrate Modbus application interface with Veris E30 have been written as a reference implementation [1].

Table 2: Veris E30 Modbus Frame Format

(a) Register read (0x03)

Request

Function code	1 Byte	0x03
Starting address	2 Bytes	0x0000 to 0xFFFF
Quantity of registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	0x03
Byte count	1 Byte	2 x N (N = Quantity of resistors)
Register value	N x 2 Bytes	

Error

Error code	1 Byte	0x83
Exception code	1 Byte	01, 02, 03, or 04

(b) Single register write (0x06)

Request

Function code	1 Byte	0x06
Starting address	2 Bytes	0x0000 to 0xFFFF
Register value	2 Bytes	0x0000 to 0xFFFF

Response

Function code	1 Byte	0x06
Register address	2 Bytes	0x0000 to 0xFFFF
Register value	2 Bytes	0x0000 to 0xFFFF

Error

Error code	1 Byte	0x86
Exception code	1 Byte	01, 02, 03, or 04

(c) Multiple register write (0x10)

Request

Function code	1 Byte	0x10
Starting address	2 Bytes	0x0000 to 0xFFFF
Quantity of registers	2 Bytes	1 to 123 (0x7B)
Byte count	1 Byte	2 x N (N = Quantity of resistors)
Register value	N x 2 Bytes	value

Response

Function code	1 Byte	0x10
Starting address	2 Bytes	0x0000 to 0xFFFF
Quantity of registers	2 Bytes	1 to 123 (0x7B)

Error

Error code	1 Byte	0x90
Exception code	1 Byte	01, 02, 03, or 04

(d) Query ID (0x11)

Request

Function code	1 Byte	0x11

Response

Function code	1 Byte	0x11
Byte count	1 Byte	N
Slave ID	1 Byte	
Run indicator	1 Byte	0x00 = OFF, 0xFF = ON
Additional data	M Bytes (M = N - 2)	

Error

Error code	1 Byte	0x91
Exception code	1 Byte	01 or 04

TCPModbusClient: E30 Modbus Client

This is a TCP client program that sends a request and receives a reply from Veris E30. Currently all function codes for the request and reply messages except exception handling codes are supported: *register read* (0x03), *single register write* (0x06), *multiple register write* (0x10), and *query ID* (0x11) commands.

- **Register read**: Figure 4 illustrates how TCPModbusClient program is used to get the CT scales and current measurements for CT arrays. This example first issues a register-read command to read current scale of CTs (register address 1000 - 1041), then it issues another register-read command to read current of CTs (register address 1335 - 1377). We can see that all the CTs have scale factor of -2 for the current, and channel 2 has current reading of 24 while all other channels have current reading of 0. This gives 0.24A ($= 24 \times 10^{-2}$) for channel 2 and 0A for all other channels. Notice that the address given to the program is one smaller, as we pointed out in Section 2.
- **Single register write**: Figure 5 shows how to set the installation mode by using a write command. We can see that the E30 node is configured for top feed (register 6 with value 0). We set the register 6 with value 1 to configure the E30 node for bottom feed.
- **Multiple register write**: Figure 6 shows how to set the CT scales by using a mwrite command. We can see that initially four mains CTs of the E30 node is configured for 200A (register 115 - 118). We set these registers to 300A so that these values match the properties of CTs we use.

Instructions on how to TCPModbusClient are listed at Figure 7.

TCPModbusServer: E30 Modbus Server

This is a TCP server program that emulates the operation of Veris E30. Function codes that are currently supported are *register read* (0x03) and *query ID* (0x11) commands. An instruction on how to TCPModbusClient is listed at Figure 8.

5 Related Work

- E30 Series Modbus Point Map [7] list all the registers used by Veris E30.
- E30 Commissioning Guide [6] list several parameters of E30 such as CT (Current Transformer) strip and CT size, which should be configured at the initial step for correct data sampling.
- E30 Installation Guide [5] list procedures for physical wiring and hardware configuration.
- Modbus Application Protocol Specification [2]
- Modbus over Serial Line Specification and Implementation Guide [3]
- GW215 Ethernet-Serial Server [4]

References

[1] Jaein Jeong. Veris E30 C Socket Reference Implementation. `http://www.cs.berkeley.edu/~jaein/Veris_E30_CSockets.tgz`.

[2] Modbus-IDA. Modbus Application Protocol Specification. `http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf`.

```
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient r
E30 TCP Modbus Client
Usage: ./TCPModbusClient read <Server IP> <Server Port> <Modbus Addr> <Register Addr>
[<Qty Registers>]
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient r 10.0.50.100 4660 1 999 42
Number of transmitting bytes: 8
01 03 03 E7 00 2A 74 66
Number of received bytes: 89
01 03 54 FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE
FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF
FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE FF FE 95 14
Response received:
  Modbus addr: 1
  Modbus function: 3
  Modbus value bytes: 84
  registers (hex):
FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE
FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE FFFE
FFFE FFFE FFFE FFFE
  registers (unsigned dec):
65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534
65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534 65534
65534 65534 65534 65534 65534 65534 65534 65534 65534 65534
  registers (signed dec):
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
-2 -2 -2 -2 -2 -2 -2 -2 -2 -2
  CRC (hex): 1495
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient r 10.0.50.100 4660 1 1335 42
Number of transmitting bytes: 8
01 03 05 37 00 2A 75 17
Number of received bytes: 89
01 03 54 00 00 00 18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FC 93
Response received:
  Modbus addr: 1
  Modbus function: 3
  Modbus value bytes: 84
  registers (hex):
0000 0018 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
  registers (unsigned dec):
0 24 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  registers (signed dec):
0 24 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  CRC (hex): 93FC
```

Figure 4: An example of using TCPModbusClient: using read command to get the CT scales and current measurements for CT arrays

```
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient w
E30 TCP Modbus Client
Usage: ./TCPModbusClient write <Server IP> <Server Port> <Modbus Addr> <Register Addr>
<Register Val>
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient r 10.0.50.100 4660 1 5 1
Number of transmitting bytes: 8
01 03 00 05 00 01 94 0B
Number of received bytes: 7
01 03 02 00 00 B8 44
Response received:
  Modbus addr: 1
  Modbus function: 3
  Modbus value bytes: 2
  registers (hex):
0000
  registers (unsigned dec):
0
  registers (signed dec):
0
  CRC (hex): 44B8
0
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient w 10.0.50.100 4660 1 5 1
Number of transmitting bytes: 8
01 06 00 05 00 01 58 0B
Number of received bytes: 8
01 06 00 05 00 01 58 0B
Response received:
  Modbus addr: 1
  Modbus function: 6
  Modbus register address: 5
  Modbus register value (hex): 0001
  Modbus register value (unsigned dec): 1
  Modbus register value (signed dec): 1
  CRC (hex): B58
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient r 10.0.50.100 4660 1 5 1
Number of transmitting bytes: 8
01 03 00 05 00 01 94 0B
Number of received bytes: 7
01 03 02 00 01 79 84
Response received:
  Modbus addr: 1
  Modbus function: 3
  Modbus value bytes: 2
  registers (hex):
0001
  registers (unsigned dec):
1
  registers (signed dec):
1
  CRC (hex): 8479
1
```

Figure 5: An example of using TCPModbusClient: using write command to set the installation mode

```
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient m
E30 TCP Modbus Client
Usage: ./TCPModbusClient writem <Server IP> <Server Port> <Modbus Addr> <Register Addr>
<Register Qty> <Val 1> <Val 2> ...
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient r 10.0.50.100 4660 1 114 4
Number of transmitting bytes: 8
01 03 00 72 00 04 E4 12
Number of received bytes: 13
01 03 08 00 C8 00 C8 00 C8 00 C8 BD A3
Response received:
  Modbus addr: 1
  Modbus function: 3
  Modbus value bytes: 8
  registers (hex):
00C8 00C8 00C8 00C8
  registers (unsigned dec):
200 200 200 200
  registers (signed dec):
200 200 200 200
  CRC (hex): A3BD
200 200 200 200
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient m 10.0.50.100 4660 1 114 4
300 300 300 300
Number of transmitting bytes: 17
01 10 00 72 00 04 08 01 2C 01 2C 01 2C 01 2C 72 A3
Number of received bytes: 8
01 10 00 72 00 04 61 D1
Response received:
  Modbus addr: 1
  Modbus function: 16
  Modbus register address: 114
  Modbus register quantity: 4
  CRC (hex): D161
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient r 10.0.50.100 4660 1 114 4
Number of transmitting bytes: 8
01 03 00 72 00 04 E4 12
Number of received bytes: 13
01 03 08 01 2C 01 2C 01 2C 01 2C E9 B6
Response received:
  Modbus addr: 1
  Modbus function: 3
  Modbus value bytes: 8
  registers (hex):
012C 012C 012C 012C
  registers (unsigned dec):
300 300 300 300
  registers (signed dec):
300 300 300 300
  CRC (hex): B6E9
300 300 300 300
```

Figure 6: An example of using TCPModbusClient: using mwrite command to set the CT scale for mains CTs

```
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient h
E30 TCP Modbus Client
Usage: ./TCPModbusClient {(q)uery | (r)ead | (w)rite | write(m) | (h)elp} ...
  query  - queries the slave ID
  read   - read one or multiple registers
  write  - write to a register
  writem - write to one or multiple registers
  help   - print this message
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient q
E30 TCP Modbus Client
Usage: ./TCPModbusClient query <Server IP> <Server Port> <Modbus Addr>
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient r
E30 TCP Modbus Client
Usage: ./TCPModbusClient read <Server IP> <Server Port> <Modbus Addr> <Register Addr>
[<Qty Registers>]
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient w
E30 TCP Modbus Client
Usage: ./TCPModbusClient write <Server IP> <Server Port> <Modbus Addr> <Register Addr>
<Register Val>
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusClient m
E30 TCP Modbus Client
Usage: ./TCPModbusClient writem <Server IP> <Server Port> <Modbus Addr> <Register Addr>
<Register Qty> <Val 1> <Val 2> ...
```

Figure 7: Command line formats for TCPModbusClient

```
jaein@redcat:~/latex_doc/veris/code/CSockets$ ./TCPModbusServer h
Usage:  ./TCPModbusServer <Server Port> <Modbus Addr>
```

Figure 8: Command line formats for TCPModbusServer

[3] Modbus-IDA. MODBUS over Serial Line Specification and Implementation Guide. http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf.

[4] neteon. Ethernet-Serial Server - GW212/GW215 User's Manual. http://http://www.neteon.net/download/216/GW21x%20Ethernet%20Serial%20Server%20user_manual.pdf.

[5] Veris Industries. E30 Installation Guide. http://www.veris.com/file_uploads/e30_i0f2.pdf.

[6] Veris Industries. E30 Series Modbus Point Map. http://www.veris.com/file_uploads/e30%20commissioning_revA.pdf.

[7] Veris Industries. E30 Series Modbus Point Map. http://www.veris.com/file_uploads/E30_Modbus_Point_Map-Rev_R.pdf.

Appendix

Code for Modbus Client and Server

- Modbus Client: E30ModbusMsg.h TCPModbusClient.c utility.c crc16.c DieWithError.c
- Modbus Server: E30ModbusMsg.h TCPModbusServer.c HandleModbusTCPClient.c crc16.c DieWithError.c

E30ModbusMsg.h

```
#ifndef E30_MODBUS_MSG_H
#define E30_MODBUS_MSG_H

#include <stdint.h>

#define SUCCESS     0
#define FAIL        1

#define CRC16_SIZE  2

/* Modbus function codes supported by E30 */
#define MODBUS_FUNC_READ_REG        0x03
#define MODBUS_FUNC_WRITE_REG       0x06
#define MODBUS_FUNC_WRITE_MULTIREG  0x10
#define MODBUS_FUNC_REPORT_SLAVEID  0x11
#define MODBUS_ERR_READ_REG         0x83
#define MODBUS_ERR_WRITE_REG        0x86
#define MODBUS_ERR_WRITE_MULTIREG   0x90
#define MODBUS_ERR_REPORT_SLAVEID   0x91

/* Byte position within a message */
#define BYTEPOS_MODBUS_ADDR             0
#define BYTEPOS_MODBUS_FUNC             1
#define BYTEPOS_MODBUS_EXCEPTION_CODE   2

#define MODBUS_REG_READ_QTY_DEFAULT   1
#define MODBUS_REG_READ_QTY_MIN       1
#define MODBUS_REG_READ_QTY_MAX       125

/* Run indicator for report_slaveid msg */
#define MODBUS_RUN_INDICATOR_ON       0xff
#define MODBUS_RUN_INDICATOR_OFF      0x00


/* Align message structures within byte boundary */
#pragma pack(push)
#pragma pack(1)

/* Message structures for requests excluding CRC16 */
```

```
typedef struct modbus_req_read_reg {
  uint8_t   modbus_addr;
  uint8_t   modbus_func;
  uint16_t  modbus_reg_addr;
  uint16_t  modbus_reg_qty;
} modbus_req_read_reg;

typedef struct modbus_req_write_reg {
  uint8_t   modbus_addr;
  uint8_t   modbus_func;
  uint16_t  modbus_reg_addr;
  uint16_t  modbus_reg_val;
} modbus_req_write_reg;

typedef struct modbus_req_write_multireg {
  uint8_t   modbus_addr;
  uint8_t   modbus_func;
  uint16_t  modbus_reg_addr;
  uint16_t  modbus_reg_qty;
  uint8_t   modbus_val_bytes;
  uint16_t  modbus_reg_val[0];
} modbus_req_write_multireg;

typedef struct modbus_req_report_slaveid {
  uint8_t   modbus_addr;
  uint8_t   modbus_func;
} modbus_req_report_slaveid;


/* Message structures for replies excluding CRC16 */

typedef struct modbus_reply_read_reg {
  uint8_t   modbus_addr;
  uint8_t   modbus_func;
  uint8_t   modbus_val_bytes;
  uint16_t  modbus_reg_val[0];
} modbus_reply_read_reg;

typedef struct modbus_reply_write_reg {
  uint8_t   modbus_addr;
  uint8_t   modbus_func;
  uint16_t  modbus_reg_addr;
  uint16_t  modbus_reg_val;
} modbus_reply_write_reg;

typedef struct modbus_reply_write_multireg {
  uint8_t   modbus_addr;
  uint8_t   modbus_func;
  uint16_t  modbus_reg_addr;
  uint16_t  modbus_reg_qty;
} modbus_reply_write_multireg;

typedef struct modbus_reply_report_slaveid {
  uint8_t   modbus_addr;
  uint8_t   modbus_func;
  uint8_t   modbus_val_bytes;
  uint8_t   modbus_slaveid;
  uint8_t   modbus_run_indicator;
  uint8_t   modbus_additional[0];
} modbus_reply_report_slaveid;


#pragma pack(pop)

/* Function declarations */

/* Error handling function */
void DieWithError(char *errorMessage);

/* Calculate 16-bit CRC */
```

```
uint16_t calc_crc16(uint8_t* modbusframe, uint16_t length);

/* The function reads 16-bit CRC from the byte array */
uint16_t read_crc16(uint8_t* byteArr, uint16_t byteOffset);

/* Print the contents of the buffer */
void print_received_msg(uint8_t *buf, int buflen);
void print_modbus_reply_read_reg(uint8_t *buf, int buflen);
void print_modbus_reply_write_reg(uint8_t *buf, int buflen);
void print_modbus_reply_write_multireg(uint8_t *buf, int buflen);
void print_modbus_reply_report_slaveid(uint8_t *buf, int buflen);

#endif
```

TCPModbusClient.c

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#include "E30ModbusMsg.h"

#define RCVBUFSIZE 1024   /* Size of receive buffer */

#define ARGS_QUERY  5
#define ARGS_READ   6
#define ARGS_WRITE  7
#define ARGS_WRITEM_REGVAL_POS 7

void print_usage_top(char *str);
void print_usage_query(char *str);
void print_usage_read(char *str);
void print_usage_write(char *str);
void print_usage_writem(char *str);
int prepare_msg_query(int argc, char* argv[], char* buf,
                      struct sockaddr_in *pServAddr);
int prepare_msg_read(int argc, char* argv[], char* buf,
                      struct sockaddr_in *pServAddr);
int prepare_msg_write(int argc, char* argv[], char* buf,
                      struct sockaddr_in *pServAddr);
int prepare_msg_writem(int argc, char* argv[], char* buf,
                      struct sockaddr_in *pServAddr);

int main(int argc, char *argv[])
{
    int sock;                      /* Socket descriptor */
    struct sockaddr_in servAddr;  /* Echo server address */
    char txBuf[RCVBUFSIZE];       /* String to send to echo server */
    char rxBuf[RCVBUFSIZE];       /* Buffer for echo string */
    uint32_t txBufLen;            /* Length of string to echo */
    int bytesRcvd;                /* Bytes read in single recv() */
    int c;

    txBufLen = 0;

    /* Zero out the server address structure */
    memset(&servAddr, 0, sizeof(servAddr));

    if (argc < 2) {
      print_usage_top(argv[0]);
    }
    /* Check arguments for help command */
    else if (strcmp(argv[1], "help") == 0 || strcmp(argv[1], "h") == 0) {
      print_usage_top(argv[0]);
    }
```

```
/* Check arguments for query command */
else if (strcmp(argv[1], "query") == 0 || strcmp(argv[1], "q") == 0) {
  if (argc >= 3 &&
      (strcmp(argv[2], "help") == 0 || strcmp(argv[2], "h") == 0)) {
    print_usage_query(argv[0]);
  }
  /* Test for correct number of arguments */
  else if (argc != ARGS_QUERY) {
    print_usage_query(argv[0]);
  }

  /* Prepare tx buffer for query command */
  txBufLen = prepare_msg_query(argc, argv, txBuf, &servAddr);
}
/* Check arguments for read command */
else if (strcmp(argv[1], "read") == 0 || strcmp(argv[1], "r") == 0) {
  if (argc >= 3 &&
      (strcmp(argv[2], "help") == 0 || strcmp(argv[2], "h") == 0)) {
    print_usage_read(argv[0]);
  }
  /* Test for correct number of arguments */
  else if (argc != ARGS_READ && argc != ARGS_READ + 1) {
    print_usage_read(argv[0]);
  }

  /* Prepare tx buffer for read command */
  txBufLen = prepare_msg_read(argc, argv, txBuf, &servAddr);
}
/* Check arguments for write command */
else if (strcmp(argv[1], "write") == 0 || strcmp(argv[1], "w") == 0) {
  if (argc >= 3 &&
      (strcmp(argv[2], "help") == 0 || strcmp(argv[2], "h") == 0)) {
    print_usage_write(argv[0]);
  }
  /* Test for correct number of arguments */
  else if (argc != ARGS_WRITE) {
    print_usage_write(argv[0]);
  }

  /* Prepare tx buffer for write command */
  txBufLen = prepare_msg_write(argc, argv, txBuf, &servAddr);
}
/* Check arguments for writem command */
else if (strcmp(argv[1], "writem") == 0 || strcmp(argv[1], "m") == 0) {
  if (argc >= 3 &&
      (strcmp(argv[2], "help") == 0 || strcmp(argv[2], "h") == 0)) {
    print_usage_writem(argv[0]);
  }
  /* Test for correct number of arguments */
  else if (argc < ARGS_WRITEM_REGVAL_POS) {
    print_usage_writem(argv[0]);
  }

  /* Prepare tx buffer for writem command */
  txBufLen = prepare_msg_writem(argc, argv, txBuf, &servAddr);
}
else {
  print_usage_top(argv[0]);
}


/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");

/* Establish the connection to the echo server */
if (connect(sock, (struct sockaddr *) &servAddr, sizeof(servAddr)) < 0)
    DieWithError("connect() failed");

/* Print the size of message */
```

```
    fprintf(stderr, "Number of transmitting bytes: %d\n", txBufLen);

    /* Print the echo buffer */
    fprintf(stderr, "%s\n", txBuf);

    /* Display the received message as hex arrays */
    for (c = 0; c < txBufLen; c++) {
      fprintf(stderr, "%02X ", (uint8_t)*(txBuf + c));
    }
    fprintf(stderr, "\n");

    /* Send the string to the server */
    if (send(sock, txBuf, txBufLen, 0) != txBufLen)
        DieWithError("send() sent a different number of bytes than expected");

    bytesRcvd = 0;
    /* Receive the same string back from the server */
    while (bytesRcvd == 0)
    {
        /* Receive up to the buffer size bytes from the sender */
        if ((bytesRcvd = recv(sock, rxBuf, RCVBUFSIZE - 1, 0)) <= 0)
            DieWithError("recv() failed or connection closed prematurely");
        rxBuf[bytesRcvd] = '\0';  /* Terminate the string! */
    }

    print_received_msg((uint8_t *)rxBuf, bytesRcvd);

    close(sock);
    exit(0);
}


void print_usage_top(char *str) {
  fprintf(stderr,"E30 TCP Modbus Client\n");
  fprintf(stderr,"Usage: %s {(q)uery | (r)ead | (w)rite | write(m) | (h)elp} ...\n",str);
  fprintf(stderr,"  query  - queries the slave ID\n");
  fprintf(stderr,"  read   - read one or multiple registers\n");
  fprintf(stderr,"  write  - write to a register\n");
  fprintf(stderr,"  writem - write to one or multiple registers\n");
  fprintf(stderr,"  help   - print this message\n");
  exit(1);
}

void print_usage_query(char *str) {
  fprintf(stderr,"E30 TCP Modbus Client\n");
  fprintf(stderr,"Usage: %s query ", str);
  fprintf(stderr,"<Server IP> <Server Port> <Modbus Addr>\n");
  exit(1);
};

void print_usage_read(char *str) {
  fprintf(stderr,"E30 TCP Modbus Client\n");
  fprintf(stderr,"Usage: %s read ", str);
  fprintf(stderr,"<Server IP> <Server Port> <Modbus Addr> <Register Addr> [<Qty Registers>]\n");
  exit(1);
}

void print_usage_write(char *str) {
  fprintf(stderr,"E30 TCP Modbus Client\n");
  fprintf(stderr,"Usage: %s write ", str);
  fprintf(stderr,"<Server IP> <Server Port> <Modbus Addr> <Register Addr> <Register Val>\n");
  exit(1);
}

void print_usage_writem(char *str) {
  fprintf(stderr,"E30 TCP Modbus Client\n");
  fprintf(stderr,"Usage: %s writem ", str);
  fprintf(stderr,"<Server IP> <Server Port> <Modbus Addr> <Register Addr> <Register Qty> <Val 1> <Val 2> ... \n");
  exit(1);
}
```

```
int prepare_msg_query(int argc, char* argv[], char* buf,
                      struct sockaddr_in *pServAddr) {
  uint16_t servPort;           /* Server port */
  char *servIP;                /* Server IP address (dotted quad) */
  int bufLen;                  /* Length of string to echo */
  uint8_t modbus_addr;         /* 8-bit modbus addr */
  uint32_t crc_temp;
  uint32_t crc_offset;

  modbus_req_report_slaveid* req_msg = NULL;

  servIP = argv[2];            /* First arg: server IP address */
  servPort = atoi(argv[3]);    /* Use given port, if any */
  modbus_addr = (uint8_t) atoi(argv[4]);   /* modbus address */

  pServAddr->sin_family      = AF_INET;     /* Internet address family */
  pServAddr->sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
  pServAddr->sin_port        = htons(servPort);   /* Server port */

  req_msg = (modbus_req_report_slaveid*) buf;
  req_msg->modbus_addr = modbus_addr;
  req_msg->modbus_func = MODBUS_FUNC_REPORT_SLAVEID;

  /* Calculate CRC16 for the request msg */
  crc_offset = sizeof(modbus_req_report_slaveid);
  crc_temp = calc_crc16((uint8_t*) buf, crc_offset);
  buf[crc_offset]   = (uint8_t) crc_temp & 0x0ff; /* lower 8bit */
  buf[crc_offset+1] = (uint8_t) (crc_temp >> 8) & 0x0ff;  /* upper 8bit */
  buf[crc_offset+2] = 0; /* end of string */

  bufLen = crc_offset + CRC16_SIZE;

  return bufLen;
}

int prepare_msg_read(int argc, char* argv[], char* buf,
                     struct sockaddr_in *pServAddr) {
  uint16_t servPort;           /* Server port */
  char *servIP;                /* Server IP address (dotted quad) */
  int bufLen;                  /* Length of string to echo */
  uint8_t modbus_addr;         /* 8-bit modbus addr */
  uint32_t crc_temp;
  uint32_t crc_offset;
  uint16_t reg_addr;           /* 16-bit register addr */
  uint16_t reg_qty;            /* quantity of registers to read (1-125) */

  modbus_req_read_reg* req_msg = NULL;

  servIP = argv[2];        /* First arg: server IP address (dotted quad) */
  servPort = atoi(argv[3]); /* Use given port, if any */
  modbus_addr = (uint8_t) atoi(argv[4]);   /* modbus address */
  reg_addr = (uint16_t) atoi(argv[5]);     /* register address */

  pServAddr->sin_family      = AF_INET;     /* Internet address family */
  pServAddr->sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
  pServAddr->sin_port        = htons(servPort);   /* Server port */

  if (argc == 7) {
    reg_qty = (uint16_t) atoi(argv[6]);    /* quantity of registers */
    /* Ensure that reg_qty between 1 to 125. */
    if (reg_qty < MODBUS_REG_READ_QTY_MIN ||
        reg_qty > MODBUS_REG_READ_QTY_MAX) {
      reg_qty = MODBUS_REG_READ_QTY_DEFAULT;
    }
  }
  else {
    reg_qty = MODBUS_REG_READ_QTY_DEFAULT;
  }
  /* Fill in each field of buf */
  req_msg = (modbus_req_read_reg*) buf;
  req_msg->modbus_addr = modbus_addr;
  req_msg->modbus_func = MODBUS_FUNC_READ_REG;
  req_msg->modbus_reg_addr = htons(reg_addr);
  req_msg->modbus_reg_qty  = htons(reg_qty);

  /* Calculate CRC16 for the request msg */
  crc_offset = sizeof(modbus_req_read_reg);
  crc_temp = calc_crc16((uint8_t*) buf, crc_offset);
  buf[crc_offset]   = (uint8_t) crc_temp & 0x0ff; /* lower 8bit */
  buf[crc_offset+1] = (uint8_t) (crc_temp >> 8) & 0x0ff;  /* upper 8bit */
  buf[crc_offset+2] = '\0'; /* end of string */

  bufLen = crc_offset + CRC16_SIZE;

  return bufLen;
}

int prepare_msg_write(int argc, char* argv[], char* buf,
                      struct sockaddr_in *pServAddr) {
  uint16_t servPort;            /* Server port */
  char *servIP;                 /* Server IP address (dotted quad) */
  int bufLen;                   /* Length of string to echo */
  uint8_t modbus_addr;          /* 8-bit modbus addr */
  uint32_t crc_temp;
  uint32_t crc_offset;
  uint16_t reg_addr;            /* 16-bit register addr */
  uint16_t reg_val;             /* 16-bit register value to write with */

  modbus_req_write_reg* req_msg      = NULL;

  servIP = argv[2];         /* First arg: server IP address (dotted quad) */
  servPort = atoi(argv[3]); /* Use given port, if any */
  modbus_addr = (uint8_t) atoi(argv[4]);   /* modbus address */
  reg_addr = (uint16_t) atoi(argv[5]);     /* register address */
  reg_val = (uint16_t) atoi(argv[6]);      /* register value */

  pServAddr->sin_family      = AF_INET;     /* Internet address family */
  pServAddr->sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
  pServAddr->sin_port        = htons(servPort);   /* Server port */

  /* Fill in each field of buf */
  req_msg = (modbus_req_write_reg*) buf;
  req_msg->modbus_addr = modbus_addr;
  req_msg->modbus_func = MODBUS_FUNC_WRITE_REG;
  req_msg->modbus_reg_addr = htons(reg_addr);
  req_msg->modbus_reg_val  = htons(reg_val);

  /* CRC: first lower 8-bit, then upper 8-bit */
  /* Exception to Big-endianess */
  crc_offset = sizeof(modbus_req_write_reg);
  crc_temp = calc_crc16((uint8_t*) buf, crc_offset);
  buf[crc_offset]   = (uint8_t) crc_temp & 0x0ff; /* lower 8bit */
  buf[crc_offset+1] = (uint8_t) (crc_temp >> 8) & 0x0ff;  /* upper 8bit */
  buf[crc_offset+2] = '\0'; /* end of string */

  bufLen = crc_offset + CRC16_SIZE;

  return bufLen;
}

int prepare_msg_writem(int argc, char* argv[], char* buf,
                       struct sockaddr_in *pServAddr) {
  uint16_t servPort;            /* Server port */
  char *servIP;                 /* Server IP address (dotted quad) */
  int bufLen;                   /* Length of string to echo */
  uint8_t modbus_addr;          /* 8-bit modbus addr */
  uint32_t crc_temp;
  uint32_t crc_offset;
  uint16_t reg_addr;            /* 16-bit register addr */
  uint16_t reg_qty;             /* quantity of registers to write */
  uint16_t reg_val;             /* 16-bit register value to write with */
  int c;

  modbus_req_write_multireg* req_msg = NULL;

  servIP   = argv[2];       /* First arg: server IP address (dotted quad) */
  servPort = atoi(argv[3]); /* Use given port, if any */
  modbus_addr = (uint8_t) atoi(argv[4]);  /* modbus address */
  reg_addr    = (uint16_t) atoi(argv[5]); /* register address */
  reg_qty     = (uint16_t) atoi(argv[6]); /* quantity of registers */

  pServAddr->sin_family      = AF_INET;     /* Internet address family */
  pServAddr->sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
  pServAddr->sin_port        = htons(servPort);   /* Server port */

  /* Test for correct number of register values */
  if (argc - ARGS_WRITEM_REGVAL_POS != reg_qty)  {
    fprintf(stderr, "Usage: %s ", argv[0]);
    fprintf(stderr, "<Server IP> <Server Port> <Modbus Addr> <Register Addr> <Register Qty> <Val 1> <Val 2> ... \n");
    fprintf(stderr, "<Register Qty> and number of registers <Val 1>, <Val 2>, ... should match. \n");
    exit(1);
  }

  /* Fill in each field of buf */
  req_msg = (modbus_req_write_multireg*) buf;
  req_msg->modbus_addr = modbus_addr;
  req_msg->modbus_func = MODBUS_FUNC_WRITE_MULTIREG;
  req_msg->modbus_reg_addr = htons(reg_addr);
  req_msg->modbus_reg_qty  = htons(reg_qty);
  req_msg->modbus_val_bytes = (uint8_t) 2 * reg_qty;

  for (c = 0; c < reg_qty; c++) {
    reg_val = (uint16_t) atoi(argv[ARGS_WRITEM_REGVAL_POS + c]);
    req_msg->modbus_reg_val[c] = htons(reg_val);
  }

  /* CRC: first lower 8-bit, then upper 8-bit */
  /* Exception to Big-endianess */
  crc_offset = sizeof(modbus_req_write_multireg) + 2 * reg_qty;
  crc_temp = calc_crc16((uint8_t*) buf, crc_offset);
  buf[crc_offset]   = (uint8_t) crc_temp & 0x0ff; /* lower 8bit */
  buf[crc_offset+1] = (uint8_t) (crc_temp >> 8) & 0x0ff;  /* upper 8bit */
  buf[crc_offset+2] = '\0'; /* end of string */

  bufLen = crc_offset + CRC16_SIZE;

  return bufLen;
}
```

utility.c

```
#include <stdio.h>
#include <stdint.h>
#include <netinet/in.h>
#include <string.h>
#include "E30ModbusMsg.h"

#define RCVBUFSIZE 1024

char rxBuf[RCVBUFSIZE];       /* buffer for the reply message */
int rxBufLen = 0;             /* length of reply message */
```

```
void print_received_msg(uint8_t *buf, int buflen) {
  int c;

  /* Print the size of message */
  fprintf(stderr, "Number of received bytes: %d\n", buflen);

  /* Print the echo buffer */
  fprintf(stderr, "%s\n", buf);

  /* Display the received message as hex arrays */
  for (c = 0; c < buflen; c++) {
    fprintf(stderr, "%02X ", (uint8_t)*(buf + c));
  }
  fprintf(stderr, "\n");

  switch (buf[BYTEPOS_MODBUS_FUNC]) {
  case MODBUS_FUNC_READ_REG:
    print_modbus_reply_read_reg(buf, buflen);
    break;

  case MODBUS_FUNC_WRITE_REG:
    print_modbus_reply_write_reg(buf, buflen);
    break;

  case MODBUS_FUNC_WRITE_MULTIREG:
    print_modbus_reply_write_multireg(buf, buflen);
    break;

  case MODBUS_FUNC_REPORT_SLAVEID:
    print_modbus_reply_report_slaveid(buf, buflen);
    break;

  default:
    break;
  }
}

void print_modbus_reply_read_reg(uint8_t *buf, int buflen) {
  uint8_t byte_cnt;
  int c;
  uint32_t crc_temp;
  modbus_reply_read_reg* reply_msg = (modbus_reply_read_reg*) buf;

  fprintf(stderr, "Response received:\n");
  fprintf(stderr, "  Modbus addr: %d\n", reply_msg->modbus_addr);
  fprintf(stderr, "  Modbus function: %d\n", reply_msg->modbus_func);
  fprintf(stderr, "  Modbus value bytes: %d\n", reply_msg->modbus_val_bytes);

  byte_cnt = reply_msg->modbus_val_bytes;

  /* Display registers */
  fprintf(stderr, "  registers (hex): \n");
  for (c = 0; c < byte_cnt / 2; c++) {
    fprintf(stderr, "%04X ", ntohs(reply_msg->modbus_reg_val[c]));
  }
  fprintf(stderr, "\n");

  fprintf(stderr, "  registers (unsigned dec): \n");
  for (c = 0; c < byte_cnt / 2; c++) {
    fprintf(stderr, "%u ", ntohs(reply_msg->modbus_reg_val[c]));
  }
  fprintf(stderr, "\n");

  fprintf(stderr, "  registers (signed dec): \n");
  for (c = 0; c < byte_cnt / 2; c++) {
    fprintf(stderr, "%d ", (short) ntohs(reply_msg->modbus_reg_val[c]));
  }
  fprintf(stderr, "\n");
  /* Check the CRC in the packet */
  crc_temp = read_crc16((uint8_t*) buf,
                        sizeof(modbus_reply_read_reg) +
                        reply_msg->modbus_val_bytes);
  fprintf(stderr, "  CRC (hex): %02X\n", crc_temp);

  for (c = 0; c < byte_cnt / 2; c++) {
    printf("%d ", (short) ntohs(reply_msg->modbus_reg_val[c]));
  }
  printf("\n");
}

void print_modbus_reply_write_reg(uint8_t *buf, int buflen) {
  uint32_t crc_temp;
  modbus_reply_write_reg* reply_msg = (modbus_reply_write_reg*) buf;

  fprintf(stderr, "Response received:\n");
  fprintf(stderr, "  Modbus addr: %d\n", reply_msg->modbus_addr);
  fprintf(stderr, "  Modbus function: %d\n", reply_msg->modbus_func);
  fprintf(stderr, "  Modbus register address: %d\n", ntohs(reply_msg->modbus_reg_addr));
  fprintf(stderr, "  Modbus register value (hex): %04X\n",
          ntohs(reply_msg->modbus_reg_val));
  fprintf(stderr, "  Modbus register value (unsigned dec): %u\n",
          ntohs(reply_msg->modbus_reg_val));
  fprintf(stderr, "  Modbus register value (signed dec): %d\n",
          (short) ntohs(reply_msg->modbus_reg_val));

  /* Check the CRC in the packet */
  crc_temp = read_crc16((uint8_t*) buf, sizeof(modbus_reply_write_reg));
  fprintf(stderr, "  CRC (hex): %02X\n", crc_temp);
}

void print_modbus_reply_write_multireg(uint8_t *buf, int buflen) {
  uint32_t crc_temp;
  modbus_reply_write_multireg* reply_msg = (modbus_reply_write_multireg*) buf;

  fprintf(stderr, "Response received:\n");
  fprintf(stderr, "  Modbus addr: %d\n", reply_msg->modbus_addr);
  fprintf(stderr, "  Modbus function: %d\n", reply_msg->modbus_func);
  fprintf(stderr, "  Modbus register address: %d\n", ntohs(reply_msg->modbus_reg_addr));
  fprintf(stderr, "  Modbus register quantity: %d\n", ntohs(reply_msg->modbus_reg_qty));

  /* Check the CRC in the packet */
  crc_temp = read_crc16((uint8_t*) buf, sizeof(modbus_reply_write_multireg));
  fprintf(stderr, "  CRC (hex): %02X\n", crc_temp);
}


void print_modbus_reply_report_slaveid(uint8_t *buf, int buflen) {
  uint32_t crc_temp;
  uint8_t additionalData[80]; /* Buffer for additional data */

  modbus_reply_report_slaveid* reply_msg = (modbus_reply_report_slaveid*) buf;

  fprintf(stderr, "Response received:\n");
  fprintf(stderr, "  Modbus addr: %d\n", reply_msg->modbus_addr);
  fprintf(stderr, "  Modbus function: %d\n", reply_msg->modbus_func);
  fprintf(stderr, "  byte count: %d\n", reply_msg->modbus_val_bytes);
  fprintf(stderr, "  slave ID (hex): %02X\n", reply_msg->modbus_slaveid);
  fprintf(stderr, "  run indicator (0x00 - OFF, 0xFF - ON): %02X\n",
          reply_msg->modbus_run_indicator);
  strncpy((char*)additionalData, (char*) reply_msg->modbus_additional,
          reply_msg->modbus_val_bytes - 2);
  fprintf(stderr, "  additional data: %s\n", additionalData);

  /* Check the CRC in the packet */
  crc_temp = read_crc16((uint8_t*) buf,
                        sizeof(modbus_reply_report_slaveid) +
                        reply_msg->modbus_val_bytes - 2);
  fprintf(stderr, "  CRC (hex): %02X\n", crc_temp);
```

```
}
```

crc16.c

```
#include <stdint.h>

/* Table of CRC values for highorder byte */
static uint8_t 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, 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
} ;

/* Table of CRC values for loworder byte */
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, 0xD3,
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
};

/* The function returns the CRC as a uint16_t type */
uint16_t calc_crc16(uint8_t* puchMsg, uint16_t usDataLen)
{
  uint8_t uchCRCHi = 0xFF ; /* high byte of CRC initialized */
  uint8_t uchCRCLo = 0xFF ; /* low byte of CRC initialized */
  uint32_t uIndex ;         /* will index into CRC lookup table */

  while (usDataLen--) /* pass through message buffer */
  {
    uIndex = uchCRCLo ^ *puchMsg++ ; /* calculate the CRC */
    uchCRCLo = uchCRCHi ^ auchCRCHi[uIndex] ;
    uchCRCHi = auchCRCLo[uIndex] ;
  }

  return (uchCRCHi << 8 | uchCRCLo) ;
}
```

```
/* The function reads 16-bit CRC from the byte array */
uint16_t read_crc16(uint8_t* byteArr, uint16_t byteOffset)
{
  uint16_t crc_temp = (byteArr[byteOffset+1] << 8) | byteArr[byteOffset];
  return crc_temp;
}
```

DieWithError.c

```
#include <stdio.h>  /* for perror() */
#include <stdlib.h> /* for exit() */

void DieWithError(char *errorMessage)
{
    perror(errorMessage);
    exit(1);
}
```

TCPModbusServer.c

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), and connect() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXPENDING 5    /* Maximum outstanding connection requests */

void DieWithError(char *errorMessage);  /* Error handling function */
void HandleTCPClient(int clntSocket, unsigned char modbusAddr);   /* TCP client handling function */

int main(int argc, char *argv[])
{
    int servSock;                    /* Socket descriptor for server */
    int clntSock;                    /* Socket descriptor for client */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned short echoServPort;     /* Server port */
    unsigned char  modbusAddr;       /* 8-bit modbus addr */
    unsigned int clntLen;            /* Length of client address data structure */

    if (argc != 3)     /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage:  %s <Server Port> <Modbus Addr>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);  /* First arg:  local port */
    modbusAddr   = (unsigned char) atoi(argv[2]);  /* Second arg: modbus addr */

    /* Create socket for incoming connections */
    if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
    echoServAddr.sin_port = htons(echoServPort);      /* Local port */

    /* Bind to the local address */
    if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
```

```
        DieWithError("bind() failed");

    /* Mark the socket so it will listen for incoming connections */
    if (listen(servSock, MAXPENDING) < 0)
        DieWithError("listen() failed");

    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        clntLen = sizeof(echoClntAddr);

        /* Wait for a client to connect */
        if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
                               &clntLen)) < 0)
            DieWithError("accept() failed");

        /* clntSock is connected to a client! */

        printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

        HandleTCPClient(clntSock, modbusAddr);
    }
    /* NOT REACHED */
}
```

HandleModbusTCPClient.c

```
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for recv() and send() */
#include <unistd.h>     /* for close() */
#include <string.h>
#include <stdint.h>
#include <netinet/in.h>
#include "E30ModbusMsg.h"

#define RCVBUFSIZE 1024   /* Size of receive buffer */
#define SLAVEID "Veris Model E30A Branch Circuit Monitor, S/N=0x12345678, Location=\"NOT_ASSIGNED\""


void HandleTCPClient(int clntSocket, uint8_t modbus_addr)
{
    char rxBuf[RCVBUFSIZE];    /* Buffer for echo string */
    int recvMsgSize = 0;        /* Size of received message */
    char txBuf[RCVBUFSIZE];  /* Buffer for reply string */
    int replyMsgSize = 0;       /* Size of reply message */

    /* Receive message from client */
    if ((recvMsgSize = recv(clntSocket, rxBuf, RCVBUFSIZE, 0)) < 0)
        DieWithError("recv() failed");

    printf("Message of %d bytes received.\n", recvMsgSize);

    /* Send received string and receive again until end of transmission */
    while (recvMsgSize > 0)      /* zero indicates end of transmission */
    {
        /* Display the received message as hex arrays */
        int c;
        for (c = 0; c < recvMsgSize; c++) {
          printf("%02X ", (uint8_t)*(rxBuf + c));
        }
        printf("\n");

        /* Check the modbus server addr */
        if (rxBuf[BYTEPOS_MODBUS_ADDR] == modbus_addr) {
          uint32_t crc_in_packet = 0;
          uint32_t crc_calculated = 0;

          /* Check the CRC in the packet */
          crc_in_packet = ((uint8_t) rxBuf[recvMsgSize - 1]) << 8 |
                          ((uint8_t) rxBuf[recvMsgSize - 2]);

          /* Calculate the CRC */
          crc_calculated = calc_crc16((uint8_t*)rxBuf, recvMsgSize - 2) & 0x0ffff;

          if (crc_in_packet == crc_calculated) {

            /* Check the available function codes */
            if (rxBuf[BYTEPOS_MODBUS_FUNC] == MODBUS_FUNC_REPORT_SLAVEID) {
              uint32_t crc_temp = 0;
              uint32_t crc_offset;

              modbus_reply_report_slaveid* replyMsg =
                (modbus_reply_report_slaveid*) txBuf;

              replyMsg->modbus_addr = modbus_addr;
              replyMsg->modbus_func = MODBUS_FUNC_REPORT_SLAVEID;
              replyMsg->modbus_val_bytes = strlen(SLAVEID) + 2;
              replyMsg->modbus_slaveid = 0xff;
              replyMsg->modbus_run_indicator = 0xff;
              strcpy((char*)replyMsg->modbus_additional, SLAVEID);

              crc_offset = sizeof(modbus_reply_report_slaveid) +
                            strlen(SLAVEID);
              crc_temp = calc_crc16((uint8_t*) txBuf, crc_offset) & 0x0ffff;
              txBuf[crc_offset] = (uint8_t) (crc_temp & 0x0ff);
              txBuf[crc_offset + 1] = (uint8_t) (crc_temp >> 8 & 0x0ff);
              txBuf[crc_offset + 2] = 0;
              replyMsgSize = crc_offset + 2;

              if (send(clntSocket, txBuf, replyMsgSize, 0) != replyMsgSize)
                DieWithError("send() failed");
            }
            else if (rxBuf[BYTEPOS_MODBUS_FUNC] == MODBUS_FUNC_READ_REG) {
              uint32_t crc_temp;
              uint32_t crc_offset;
              uint16_t reg_qty;
              uint16_t cnt;

              modbus_req_read_reg* reqMsg = (modbus_req_read_reg*) rxBuf;
              modbus_reply_read_reg* replyMsg = (modbus_reply_read_reg*) txBuf;

              reg_qty = ntohs(reqMsg->modbus_reg_qty);

              replyMsg->modbus_addr = modbus_addr;
              replyMsg->modbus_func = MODBUS_FUNC_READ_REG;
              replyMsg->modbus_val_bytes = 2 * reg_qty;
              for (cnt = 0; cnt < reg_qty; cnt++) {
                replyMsg->modbus_reg_val[cnt] = htons(cnt);
              }

              crc_offset = sizeof(modbus_reply_read_reg) + 2 * reg_qty;
              crc_temp = calc_crc16((uint8_t*) txBuf, crc_offset) & 0x0ffff;
              txBuf[crc_offset] = (uint8_t) (crc_temp & 0x0ff);
              txBuf[crc_offset + 1] = (uint8_t) (crc_temp >> 8 & 0x0ff);
              txBuf[crc_offset + 2] = 0;
              replyMsgSize = crc_offset + 2;

              if (send(clntSocket, txBuf, replyMsgSize, 0) != replyMsgSize)
                DieWithError("send() failed");
            }
            else {
              printf("Modbus function code %02x not supported!\n",
                rxBuf[BYTEPOS_MODBUS_FUNC]);
            }
          }
          else {
            printf("CRC does not match!\n");
            printf("CRC in the packet: %02x\n", crc_in_packet & 0x0ffff);
            printf("CRC calculated: %02x\n", crc_calculated & 0x0ffff);
```

```
          }
        }
        else {
          printf("Modbus server address does not match!\n");
          printf("address in the packet: %d\n", rxBuf[BYTEPOS_MODBUS_ADDR]);
          printf("address of this server: %d\n", modbus_addr);
        }

        /* See if there is more data to receive */
        if ((recvMsgSize = recv(clntSocket, rxBuf, RCVBUFSIZE, 0)) < 0)
            DieWithError("recv() failed");
    }

    close(clntSocket);    /* Close client socket */
}
```