Tritex Modbus Protocol Specification

Introduction

This document describes *Tritex's* implementation of the MODBUS communication protocol used for transferring data between a serial host and an Exlar drive. MODBUS is a *Master/Slave* (*Client/Server*) application layer messaging protocol allowing communications between a single master (client) device and a single or multiple slave (server) device(s) on different types of buses or networks.

The MODBUS protocol may be implemented on a variety of hardware networking platforms. This document deals only with the *Tritex* implementation of MODBUS over a serial line using the MODBUS *RTU* (*Remote Terminal Unit*) transmission mode. *RTU* is the binary implementation of the protocol encoding data in eight-bit binary *bytes* and implementing *CRC* error checking on each transmission. The ASCII implementation of the protocol (which encodes each byte as two ASCII seven-bit characters) is not supported.

References

MODBUS Application Protocol Specification V1.1
MODBUS over Serial Line – Specification & Implementation Guide V1.0

Modbus.org maintains the specifications for implementation of the MODBUS protocol. Current documents are available directly from www.Modbus-IDA.org.

General Protocol Description

ADU/PDU

MODBUS protocol defines a *protocol data unit* (*PDU*) which is independent of the underlying communication layer used for transmission. The *application data unit* (*ADU*) includes additional fields used to implement the protocol on specific communication buses or networks. The *Tritex* implementation of MODBUS *RTU* over a serial line uses the *ADU* shown below.

ADU			
Address (1 byte)	PDU		CRC (2 bytes)
ID	Function Code (1 byte)	Data (in bytes)	Low : High

MODBUS is, in general, a *request-response* protocol. The *ADU* describes the data format used for both the master's transmission to the slave (the *request*) and the slave's transmission to the master (the *response*). The complete *request-response* is considered a *transaction*. A slave device will never transmit a *response* without first having received a legal (properly formatted and *CRC* validated) *request*.

Function Code

The first byte of the *PDU* is an agreed upon code that describes the type of transaction taking place and the format of the data in the transmission. The function codes specify the services offered by the protocol and tell the slave

(server) what type of action to perform. Function codes 128 (0x80) to 255 (0xFF) are reserved for exception (error) responses. When the slave responds to the master the function code field in the response indicates either a normal (error-free) response or an error response. In a normal response the slave simply echoes the original function code in the response. An error response is indicated by returning the original function code with the high bit (0x80) set in the function code field.

Data

The content of the data field in the *PDU* is dependent on the type of transaction. The data field is an optional field and may not be required for all transaction types. The data field contains additional information that the server uses to take the action defined by the function code. Unless specified otherwise, MODBUS uses a *big-Endian* format for addresses and other items in the data field when the data is larger than a single byte. In *big-Endian* format the most significant byte is sent first.

Address

The MODBUS address (ID) is a single byte identifier of the slave device. A slave will transmit a *response* only to a *request* addressed to it. MODBUS protocol specifies that the slave address must be in the range of 1 to 247.
A master *request* with an address of zero is considered a *global request* (broadcast mode). All slaves may act on a *global request* but none will respond to it. The *global request* is an exception the standard *request-response* protocol. Although the *Tritex* implementation supports the use of an *ADU* with *global request* addressing, its use is discouraged and the verification of its receipt and acceptance by any or all of the drives on the serial line is the responsibility of the user. For instance, it may be useful for a master to stop or disable all drive with a single transmission but it should then verify the status of each drive to guarantee that the transmission was received, accepted, and acted upon by the drives.

CRC

The *CRC* (*Cyclic Redundancy Check*) field of the *ADU* is used by the slave to validate the *request* or by the master to validate the *response*. It is a two-byte field and, unlike normal MODBUS 16-bit data fields, is always transmitted in *Little Endian* format with the low byte transmitted first and the high byte transmitted second. The *CRC* is calculated by the transmitting device and appended to *ADU*. The *CRC* is recalculated by the receiving device and compared with the value in the *ADU*. It is an error if the values do not match. A slave device cannot respond to a *request*, even if it seems properly addressed and otherwise formatted, when a *CRC* error has been detected (note that there is no MODBUS exception code for a *CRC* error) since it cannot count on the state of any of the data in the transaction. Instead, the master will eventually time out waiting for a response and may resend the *request* if desired. The master may also resend a *request*, if desired, when it detects a *CRC* error in a *response* from a slave.

Exception Response

The *PDU* of an exception (error) response from the slave consists of the original function code with the high bit set and a single byte data field (the *exception code*) indicating the type of error that has occurred. The following table indicates possible exception codes used in the *Tritex* implementation.

MODBUS Exception Codes		
Code	Name	Meaning
01	ILLEGAL FUNCTION	The function code received in the request is not an allowable action for the server (slave). This error may occur when the function code is only applicable to newer devices and was not implemented in current device, when the function code is unrecognized, or when the slave is in an invalid state to process the specified function code.
02	ILLEGAL DATA ADDRESS	The data address specified in the request is not an allowable address for the server (slave). More specifically, the combination of reference number and transfer length is invalid.
03	ILLEGAL DATA VALUE	A value specified in the request is not an allowable value for the server (slave), indicating a fault in the structure of the remainder of the request. This error DOES NOT mean that a data item submitted for storage in a register has a value outside the expected range for the register.
04	SLAVE DEVICE FAILURE	An unrecoverable error occurred while the server (slave) was attempting to perform the request.

Register Addresses

The MODBUS data model works with 16-bit data elements called MODBUS d*ata registers*. Within the *PDU*, every 16-bit data element is addressed (identified) by an implementation defined 16-bit *data register address* in the range of 0 to 65535 (0x0000 .. 0xFFFF). These addresses are not *physical* addresses, but *logical* addresses defined by the implementation to organize the data. The internal physical mapping of the data may or may not be contiguous with the logical mapping and is normally of no concern to the user of the data. The MODBUS data model imposes no restrictions on the logical groupings. Multiple data registers may, in fact, refer to the same physical address.

Most function codes deal with the reading and writing of data to and from MODBUS registers. Each function code defines the data accessed through its *data registers*. Read- only data registers containing data that cannot be written to and are called *Input Registers,* while read-write registers containing data that can be both read and written are called *Holding Registers*.

Historical Note

Some MODBUS implementations make a distinction between a zero-based PDU register address in the range 0 to 65535 and a MODBUS Data Model register or element number in the range of 1 to 65536 where the PDU register address is always one less than the address specified in the Data Model. This discrepancy can be quite confusing and the Tritex documentation makes no such distinction, always describing the actual PDU (zero based) register address number in its documentation.

Function Codes

Public Function Codes

MODBUS protocol defines several standard publicly documented function codes whose format and use must conform to the existing standard. The following public function codes are supported in the *Tritex* implementation.

- 03 (0x03) - Read Holding Registers
- 04 (0x04) - Read Input Registers
- 06 (0x06) - Write Single Register
- 16 (0x10) - Write Multiple Registers
- 17 (0x11) - Report Slave ID

Custom Function Codes

Custom (user-defined) function codes within the MODBUS specification must be in the range 65 to 72 or 100 to 110 decimal. The *Tritex* implementation uses the following custom function codes to deal with 32-bit data.

- 103 (0x67) - Read 32-bit Holding Registers
- 104 (0x68) - Read 32-bit Input Registers
- 106 (0x6A) - Write 32-bit Holding Register

The 32-bit function codes allow the user an alternative to reading or writing multiple 16- bit registers and expect or return the 32-bit value in standard MODBUS *Big Endian*(high word : low word) format. Although each 32-bit register may be read or written using the standard read/write multiple register function codes, the custom 32-bit codes guarantee that the full 32-bit register value is read or written in a single, non-interruptible (atomic) operation, guaranteeing data integrity. Without this guarantee, asynchronous processes in the drive could act on a partially written value or overwrite a value that was only partially read.

Unsupported Function Codes

Standard MODBUS public function codes dealing with reading and writing of single or multiple bits (discrete inputs and coils) are not supported. All data is dealt with in either 16-bit (word) or 32-bit (double word) register quantities.

Function Code Descriptions

This section describes the *PDU* format of both the *request* and *response* portion of the MODBUS *transaction* for all supported function codes. Exception codes that may be returned in an *error response* to the request are also listed. All descriptions show only the format of the *PDU*, the MODBUS ID and *CRC* fields of the full *MDU* are not shown.

An *error response* specifying an *ILLEGAL FUNCTION* exception code (01) will be given for any (properly formatted) request received by the drive that contains a function code other than one listed in this section.

Function Code 03 (0x03) Read Holding Registers

This function code reads a contiguous block of 16-bit *holding* registers. The *Request PDU* specifies the starting register address and the number of registers to read. The *Response PDU* returns the register values packed as two bytes per register – the first byte contains the high order bits and the second byte contains the low order bits.

Request		
Function Code	1 byte	0x03
Starting Address	2 bytes	0x0000 to 0xFFFF
Number of Registers	2 bytes	1 to 60

Response		
Function Code	1 byte	0x03
Byte Count	1 byte	2 x N*
Register Value(s)	2 x N* bytes	value(s)

*N = number of registers

Exception Response		
Function Code	1 byte	0x83
Exception Code	1 byte	01, 02, 03, 04

Example			
This example reads 16-bit holding registers 100 (0x64) and 101 (0x65). The value returned for register 100 is 500 (0x01F4) and the value returned for register 101 is 1000 (0x03E8).			
Request		**Response**	
Function Code	0x03	Function Code	0x03
Starting Address High	0x00	Byte Count	0x04
Starting Address Low	0x64	Register 100 Value High	0x01
Register Count High	0x00	Register 100 Value Low	0xF4
Register Count Low	0x02	Register 101 Value High	0x03
		Register 101 Value Low	0xE8

Function Code 04 (0x04) Read Input Registers

This function code reads a contiguous block of 16-bit *input* registers. The *Request PDU* specifies the starting register address and the number of registers to read. The *Response PDU* returns the register values packed as two bytes per register – the first byte contains the high order bits and the second byte contains the low order bits.

Request		
Function Code	1 byte	0x04
Starting Address	2 bytes	0x0000 to 0xFFFF
Number of Registers	2 bytes	1 to 60

Response		
Function Code	1 byte	0x04
Byte Count	1 byte	2 x N*
Register Value(s)	2 x N* bytes	value(s)

*N = number of registers

Exception Response		
Function Code	1 byte	0x84
Exception Code	1 byte	01, 02, 03, 04

Example			
This example reads 16-bit input registers 4 and 5. The value returned for register 4 is 32768 (0x8000) and the value returned for register 5 is 32767 (0x7FFF).			
Request		**Response**	
Function Code	0x04	Function Code	0x04
Starting Address High	0x00	Byte Count	0x04
Starting Address Low	0x04	Register 4 Value High	0x80
Register Count High	0x00	Register 4 Value Low	0x00
Register Count Low	0x02	Register 5 Value High	0x7F
		Register 5 Value Low	0xFF

Example			
This example reads 16-bit input register 5. The value returned for register 5 is 32767 (0x7FFF). Often, only the most significant 16-bits of a 32-bit value are required. This example also illustrates how to read the high 16-bits of the 32-bit register at MODBUS address 4 used in the example above.			
Request		**Response**	
Function Code	0x04	Function Code	0x04
Starting Address High	0x00	Byte Count	0x02
Starting Address Low	0x05	Register 5 Value High	0x7F
Register Count High	0x00	Register 5 Value Low	0xFF
Register Count Low	0x01		

Function Code 06 (0x06) Write Single Register

This function code writes a single 16-bit *holding* register. The *Request PDU* specifies the address of the register to be written and the 16-bit value to write to the register packed in two bytes with the first byte containing the high order bits and the second byte containing the low order bits. The *Response PDU* is an echo of the request.

Request		
Function Code	1 byte	0x06
Register 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

Exception Response		
Function Code	1 byte	0x86
Exception Code	1 byte	01, 02, 03, 04

Example			
This example writes 16-bit holding register 356 (0x0164) with the value 23131 (0x5A5B).			
Request		**Response**	
Function Code	0x06	Function Code	0x06
Register Address High	0x01	Register Address High	0x01
Register Address Low	0x64	Register Address Low	0x64
Register Value High	0x5A	Register Value High	0x5A
Register Value Low	0x5B	Register Value Low	0x5B

Function Code 16 (0x10) Write Multiple Registers

This function code writes a contiguous block of 16-bit *holding* registers. The *Request PDU* specifies the starting register address, the number of registers to write, the byte count of data values in the request, and the list of data values to write packed as two bytes per register with the first byte containing the high order bits and the second byte containing the low order bits. The *Response PDU* echoes the starting address and number of registers written (but *does not* return the byte count and data written).

Request		
Function Code	1 byte	0x10
Starting Address	2 bytes	0x0000 to 0xFFFF
Number of Registers	2 bytes	1 to 60
Byte Count	1 byte	2 x N*
Register Value(s)	2 x N* bytes	value(s)

*N = number of registers

Response		
Function Code	1 byte	0x10
Starting Address	2 bytes	0x0000 to 0xFFFF
Number of Registers	2 bytes	1 to 60

Exception Response		
Function Code	1 byte	0x90
Exception Code	1 byte	01, 02, 03, 04

Example			
This example writes 16-bit holding registers 356 (0x0164) and 357 (0x0165) with the values 258 (0x0201) and 772 (0x0403) respectively.			
Request		**Response**	
Function Code	0x10	Function Code	0x10
Starting Address High	0x01	Starting Address High	0x01
Starting Address Low	0x64	Starting Address Low	0x64
Number of Registers High	0x00	Number of Registers High	0x00
Number of Registers Low	0x02	Number of Registers Low	0x02
Byte Count	0x04		
Register 356 Value High	0x02		
Register 356 Value Low	0x01		
Register 357 Value High	0x04		
Register 357 Value Low	0x03		

Function Code 17 (0x11) Report Slave ID

This function code may be used to '*ping*' the drive. The *Request PDU* has a *null* (zero length) data field. The *Response PDU* returns the drive type and 16-character ASCII name.

Request		
Function Code	1 byte	0x11

Response		
Function Code	1 byte	0x11
Byte Count	1 byte	0x12
Drive Type	1 byte	*
Drive Name	16 bytes	0x20 to 0x7F
Run Indicator Status	1 byte	0xFF (ON)

* 0 = EM20, 1 = EM30

Exception Response		
Function Code	1 byte	0x91
Exception Code	1 byte	01, 04

Example			
This example shows the response to a ping from an EM30 drive named ‘Main Valve’.			
Request		**Response**	
Function Code	0x11	Function Code	0x11
		Byte Count	0x12
		(EM30) Drive Type	0x01
		(M) DriveName	0x4C
		(a)	0x60
		(i)	0x68
		(n)	0x6D
		(space)	0x20
		(V)	0x55
		(a)	0x60
		(l)	0x6B
		(v)	0x75
		(e)	0x64
		(space)	0x20
		(space)	0x20
		(space)	0x20
		(space)	0x20
		(space)	0x20
		(space)	0x20
		(constant)	0xFF

Function Code 103 (0x67) Read 32-bit Holding Registers

This custom function code reads a contiguous block of 32-bit *holding* registers. The *Request PDU* specifies the starting register address and the number of **32-bit** registers to read. The *Response PDU* returns each register values packed in four bytes with the first byte containing the most significant (high order) eight bits, the second byte containing the next most significant eight bits (i.e. the low byte of the high word), the third byte containing the next most significant eight bits (i.e. the high byte of the low word), and the fourth byte containing the least eight significant bits (i.e. the low byte of the low word). Each 32-bit register value read in an atomic operation.

Request		
Function Code	1 byte	0x67
Starting Address	2 bytes	0x0000 to 0xFFFF
Number of 32-bit Registers	2 bytes	1 to 30

Response		
Function Code	1 byte	0x67
Byte Count	1 byte	4 x N*
Register Value(s)	4 x N* bytes	value(s)

*N = number of 32-bit registers

Exception Response		
Function Code	1 byte	0xE7
Exception Code	1 byte	01, 02, 03, 04

Example			
This example read the 16-bit holding registers 100 (0x64) and 101 (0x65) in the function code 3 example using the 32-bit register read command . The value returned for the 32- bit register 100 is 65536500 (0x03E801F4) .			
Request		**Response**	
Function Code	0x67	Function Code	0x67
Starting Address High	0x00	Byte Count	0x04
Starting Address Low	0x64	Register 101 Value High	0x03
Register Count High	0x00	Register 101 Value Low	0xE8
Register Count Low	0x01	Register 100 Value High	0x01
		Register 100 Value Low	0xF4

Function Code 104 (0x68) Read 32-bit Input Registers

This custom function code reads a contiguous block of 32-bit *input* registers. The *Request PDU* specifies the starting register address and the number of **32-bit** registers to read. The *Response PDU* returns each register values packed in four bytes with the first byte containing the most significant (high order) eight bits, the second byte containing the next most significant eight bits (i.e. the low byte of the high word), the third byte containing the next most significant eight bits (i.e. the high byte of the low word), and the fourth byte containing the least eight significant bits (i.e. the low byte of the low word). Each 32-bit register value read in an atomic operation.

Request		
Function Code	1 byte	0x68
Starting Address	2 bytes	0x0000 to 0xFFFF
Number of 32-bit Registers	2 bytes	1 to 30

Response		
Function Code	1 byte	0x68
Byte Count	1 byte	4 x N*
Register Value(s)	4 x N* bytes	value(s)

*N = number of 32-bit registers

Exception Response		
Function Code	1 byte	0xE8
Exception Code	1 byte	01, 02, 03, 04

Example			
This example reads 32-bit input register 4 as in the example used for function code 4. The returned value is 2147450880 (0x7FFF8000).			
Request		**Response**	
Function Code	0x68	Function Code	0x68
Starting Address High	0x00	Byte Count	0x04
Starting Address Low	0x04	Register 5 Value High	0x7F
Register Count High	0x00	Register 5 Value Low	0xFF
Register Count Low	0x01	Register 4 Value High	0x80
		Register 4 Value Low	0x00

Example			
This example reads 32-bit input registers 4 and 6. The returned value for 32-bit register 4 is 2147450880 (0x7FFF8000), and the returned value for 32-bit register 6 is 10000000 (0x00989680).			
Request		**Response**	
Function Code	0x68	Function Code	0x68
Starting Address High	0x00	Byte Count	0x04
Starting Address Low	0x04	Register 5 Value High	0x7F
Register Count High	0x00	Register 5 Value Low	0xFF
Register Count Low	0x02	Register 4 Value High	0x80
		Register 4 Value Low	0x00
		Register 7 Value High	0x00
		Register 7 Value Low	0x98
		Register 6 Value High	0x96
		Register 6 Value Low	0x80

Function Code 106 (0x6A) Write 32-bit Holding Register

This custom function code writes a single 32-bit *holding* register. The *Request PDU* specifies the address of the register to be written. MODBUS registers in the PDU are addressed starting at zero. The *Response PDU* is an echo of the request.

This function code writes a single 32-bit *holding* register. The *Request PDU* specifies the address of the register to be written and the 32-bit value to write to the register packed in four bytes with the first byte containing the most significant (high order) eight bits, the second byte containing the next most significant eight bits (i.e. the low byte of the high word), the third byte containing the next most significant eight bits (i.e. the high byte of the low word), and the fourth byte containing the least eight significant bits (i.e. the low byte of the low word). The *Response PDU* is an echo of the request.

Note that function code 16 (0x10) for writing multiple 16-bit registers may be used (with a register count of two) to read a 32-bit holding register value. The drive,however,stores32-bit numbers internally in *Little Endian* format with the low word first so function code 16 must specify the low word (least significant 16-bits) first, followed by the high word (most significant 16-bits) to write the 32-bit register value. Using (custom) function code 106 (0x6A) instead offers the following advantages:

- Register byte order is the normal MODBUS high tolow
- The 32-bit write is guaranteed to be atomic (i.e. a single non-interruptible operation).
- Transmission efficiency – the total request/ response transaction requires fourteen bytes as opposed to the fifteen required by function code16.

Request		
Function Code	1 byte	0x6A
Register Address	2 bytes	0x00000000 to 0xFFFFFFFF
Register Value	4 bytes	0x00000000 to 0xFFFFFFFF

Response		
Function Code	1 byte	0x6A
Register Address	2 bytes	0x00000000 to 0xFFFFFFFF
Register Value	4 bytes	0x00000000 to 0xFFFFFFFF

Exception Response		
Function Code	1 byte	0xEA
Exception Code	1 byte	01, 02, 03, 04

Example

This example writes 32-bit holding register 356 (0x0164) with the value 67305985 (0x04030201). Upon completion, 16-bit register 356 hold the value 0x0201 and 16-bit register 357 holds the value 0x0403. Compare with the example for function code 6 (Write Multiple 16-bit Registers) which obtains the same result.

Request		Response	
Function Code	0x10	Function Code	0x10
Register Address High	0x01	Register Address High	0x01
Register Address Low	0x64	Register Address Low	0x64
Register 357 Value High	0x04	Register 357 Value High	0x04
Register 357 Value Low	0x03	Register 357 Value Low	0x03
Register 356 Value High	0x02	Register 356 Value High	0x02
Register 356 Value Low	0x01	Register 356 Value Low	0x01

MODBUS RS485 Serial

This section describes the specifics of the MODBUS *RTU* protocol implementation over a serial line that implements an electrical interface in accordance with the *EIA/TIA-485* standard (also known as the *RS485* standard). The *Tritex* uses a two-wire (balanced pair with common) *RS485* configuration on which only one driver can transmit at a given time.

Serial Character Format

Each eight-bit data byte of the ADU is normally transmitted as an 11-bit serial character in the following format:

Serial Character Format	
Number of Bits	Definition
1	Start bit
8	Data bits
1	Parity Bit
1	Stop bit

The *Tritex* implementation uses even parity by default as required by the MODBUS specification but optionally accepts odd or no parity. If no parity is specified, an additional stop bit is transmitted to fill out the 11-bit serial character.

Serial Data Rates

The *Tritex* implementation sets 19.2K baud (bits/sec) as the default serial signal rate as specified by the MODBUS specification. Other standard rates available include 4800, 9600, and 38.4K baud.

The time required to transmit a serial character is dependent on the baud rate and may be calculated as:

Character Time (sec/character) = (11 bits/character) / (baud bits/sec)

RTU Message Framing

A MODBUS message is transmitted as a frame with a known beginning and end point allowing devices to determine when a new message is starting and when a message being received has is completed. To accomplish this goal MODBUS defines two timing constraints – a minimum frame interval and a maximum character interval.

Frame Interval

Frames (messages) must be separated by a silent interval (no activity on the serial line) of at least 3.5 character times. This idle time requirement allows a receiver to know when it should begin looking for a new incoming frame and forces a transmitter to delay its response until the idle time requirement has been met. In the *Tritex* implementation, the minimum frame interval may be extended, if desired, to allow more setup time for slower devices or to allow extra time for the master to guarantee that the *RS485* line is not being driven before the drive attempts transmission. (Note, however, that the extended frame interval applies only to the time the drive will keep the line idle before transmitting its response, not to the idle time expected by the drive before expecting a new request.)

Valid MODBUS Frames				
3.5 ch time (min)	Frame1	3.5 ch time (min)	Frame2	3.5 ch time (min)

Character Interval

MODBUS messages must be transmitted as a continuous stream of characters. When a silent interval of greater than 1.5 character times is detected between incoming characters the message is considered complete. The frame interval time must still be met, however, so if another character is received before the minimum frame interval time is met the message should be aborted. The *Tritex* implementation allows for an additional user specified extra character interval time to ease the implementation of slower master devices that cannot guarantee an uninterrupted character transmission stream to the drive.

Invalid MODBUS Frame				
3.5 ch time (min)	Frame1	1.0 ch time	Unexpected Frame	…

Error Checking

Security for MODBUS *RTU* relies on both the hardware parity error checking (unless the no parity option is set) and the software *CRC* error checking of the *ADU*. The drive (slave) will not construct a response to the master if an error is detected from either of the error checking methods and the master should eventually timeout.

Master Timeout

The master (client) should implement a maximum time (timeout) that it is willing to wait for a response, allowing it to recover when a response is not forthcoming from the drive. Note that a timeout will also allow it to recover from a request to a non-existent slave device. In the *Tritex* implementation, the drive will normally respond immediately and the master can expect to start receiving the response within a character time or two from the end of the required minimum frame interval. Extra time, however, may be required by some drive commands such as those that require the writing of non-volatile memory.

CRC (Cyclical Redundancy Check) Generation

The *CRC* (*Cyclical Redundancy Check*) is a two byte (16-bit) field in the *ADU* calculated by the transmitting device and used by the receiving device to validate the *ADU*. The *CRC* is calculated over address byte of the *ADU* and all data bytes of the *PDU* and then appended to the message to complete *ADU*. When transmitted, the low-order byte of the *CRC* is transmitted first, followed by the high-order byte. The following example shows the layout of the *CRC* within the *ADU*.

Example ADU for calculated CRC of 0x1234				
Address	Function Code	(Data)	CRC LO	CRC HI
byte	byte	[byte(s)]…	0x34	0x12

CRC Algorithm Description

The 16-bit binary *CRC* value is initialized to all ones (0xFFFF). Each eight-bit byte of the message is then used to modify the *CRC* value. Only the eight bits of data are used in generating the *CRC* (start, stop, and parity bits are not used). The *CRC* is modified to the *exclusive or* of its original value with the zero-extended value of the eight-bit data byte.

The *CRC* is then binary shifted right (with a zero filled into the most significant bit position). If the *LSB* (least significant bit) of the *CRC before* the shift was a one, the *CRC* is *exclusive or'ed* with the constant 0xA001. The process of shifting the *CRC* (and optionally *exclusive or'ing* with the constant 0xA001) is repeated for a total of eight times. The final *CRC* value is then appended to the *ADU* with the low byte being transmitted first.

Sample Code

The following pseudo-code shows one way of directly implementing the *CRC* algorithm described above. (Note that other, possibly faster, methods may be implemented including direct table lookup in a pre-calculated table of *CRC* values for all possible data values.)

```
/* ------------------------------------------------------------
   Sample Pseudo - 'C' code for CRC Generation
          ------------------------------------------------------------*/

typedef unsigned int UINT16; typedef unsigned
char UINT8;

UINT16 CalculateCRC(int dataCount, UINT8* ptrToData)
{
        int i;
        UINT16 crc=0xFFFF;          /* Initialize result*/
        UINT16carryOut;             /* True if shift would generate carry*/
                                    /* (i.e. low bit was a one) */

        /* Loop thru all data bytes */ while
        (dataCount-- != 0)
        {
                crc ^= (UINT16) (*ptrToData++);

                for (i = 0;  i < 8;i++)
                {
                        carryOut = crc & 0x0001; crc >>= 1;
                        if (carryOut)
                        {
                                        crc ^= 0xA001;
                        }
                }
        }

        /* Return 16-bit CRC value – low byte of this value sent first */ return crc;
}
```