MODBUS PROTOCOL

PHOENIX AC DRIVE

DX, EX, DS & ES

3 TO 3500 HP

MODBUS RTU PROTOCOL
MODBUS TCP PROTOCOL
MODBUS USD PROTOCOL

US DRIVES, INC.

Made In the U.S.A.

LIST OF FIGURES AND TABLES

1.0 INTRODUCTION

The US DRIVES PHOENIX digital AC drive has an optional **isolated** serial communication interface that enables the user to control and setup the drive using a host computer, such as an IBM compatible PC. The drive's serial interface may be configured to support either the RS-422 or RS-485 interconnect standard. In addition, schematics shown in this document show a simple hookup that permits an RS-232 interface (this hookup does not meet EIA standards, due to noise immunity levels, but will work with most host computers). The PHOENIX Modbus USD protocol is loosely based on the MODBUS RTU Protocol and features full CRC (Cyclical Redundancy Check) protection in both directions. The Phoenix drive defaults to 9600 Baud Rate, No Parity, 8-Bit Data and 2 Stop Bits.

Using an RS-485 serial interconnect, up to 32 drives can be party-lined to permit the host computer to setup and query any one of them. With using the Ethernet Communications Card, any host computer on a network can communicate to the Phoenix AC Drives.

The PHOENIX AC drive contains an onboard database of over 282 parameters that define and describe drive setups and operating conditions. In essence, a user armed with a printout of the drive's parameters can configure and operate the drive with a simple read and write command to the desired parameter.

For those users wishing to develop their own host computer drive control software, the following sections of this manual describe the Modbus USD, Modbus RTU, and Modbus TCP Protocols that are used.

END OF INTRODUCTION SECTION

2.0 HARDWARE INTERFACE

2.1 RS-485 4-Wire Operation

RS-485 and RS-422 interfaces use differential transceivers for increased noise immunity, since any noise induced in one wire will usually be induced in the other wire and thus will be canceled out at the differential receiver.

RS-485 4-Wire operation allows 32 drivers and 32 receivers to be party-lined together at distances up to 4000 feet. This allows a host computer to talk to 31 drives. The host computer has its transmitter and receiver enabled at all times. The drives always have their receivers enabled but only one drive transmitter on the party-line can be enabled at any one time.

The host computer transmits a query to a specific drive (queries have an address field that identifies the destination drive). Even though all drives on the party-line or network receive the query and decode it, only one drive will prepare and transmit a response. The destination drive enables its transmitter during its response and disables it immediately after transmission is complete. This sequencing of the drive transmitters is built into the PHOENIX drive software. The host software requires no special hand-shaking since the transmitter and receiver are enabled at all times.

There are a number of electrical supply houses that offer RS-485 interface cards for IBM-compatible PCs. US Drives has decided to only support 4-Wire RS-485 and RS-422 operation because standard MS-DOS serial device drivers may be used without modification. The proper hookup for 4-Wire RS-485 Multi-Drop between a host PC and a number of PHOENIX drives is shown in Figure 2-1.

2.2 RS-422 4-Wire Operation

RS-422 4-Wire operation is similar to RS-485 4-Wire operation except that the wiring is "point-to-point" with no other drives party-lined. This mode works at full duplex and is illustrated in Figure 2-2.

2.3 RS-232 Operation

The PHOENIX drive permits a direct RS-232 interface. It is felt, however, that the differential transmission scheme offered by the RS-422/485 standard is much more suitable for an industrial environment. Direct connection to a PC using the RS-232 scheme is not recommended for drives operating on the factory floor.

Those users that still wish to use a RS-232 interface have the following alternatives.

2.3.1 RS-232C-to-RS-422 Interface Adapter

Users who wish to use the PC's RS-232C serial port can install a RS-232C-to-RS-422 converter that is readily available from a number of electrical suppliers. It need not be isolated, as the PHOENIX user serial port is already electrically isolated from the rest of the drive. These converters typically look much like a "25 pin gender changer" plug with the conversion circuits built into the plug. Power is normally supplied to the converter by an AC-DC adapter that plugs into an 110vac duplex outlet.

2.3.2 Direct RS-232 Wiring

The following "scheme" is not recommended for permanent use. By connecting the +RXD and +TXD pins to the PC RS-232 ground, a quasi-single ended RS-232 interface can be accomplished.

Normal RS-232C signals bounce between +10 volts and –10 volts (the positive rail voltage can be between the values of +3 volts and +25 volts – likewise for the negative rail). Thus, +10 volts is interpreted as a logic "0" while -10 volts is interpreted as a logic "1". Most RS-232C receivers will detect zero volts as a logic "1" due to hysteresis effects which allows the wiring scheme shown in Figure 2-3 to work (usually!).

This direct connection generates RS-232C levels of approximately +3.7 volts to -3.7 volts. Note that the isolated RS-422/485 common on the PHOENIX control board (TB6-5) must **not** be grounded or it will short-circuit the +TXD line if the PC is grounded. This direct RS-232 wiring scheme, if it must be done, is most appropriate when using a laptop PC that is floating from earth ground.

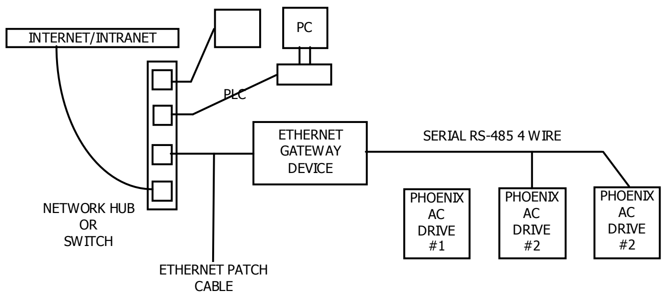
2.3.3 RS-232 Isolated Communication Interface

Using the Removable RS-232 Isolated Communication Interface, user's can use the PC's RS-232C serial port. The Communication Interface isolates the PC from the Phoenix AC Drive and no external power supply is needed. This comes with D9 female connector, 10' cable, and removable RS-232 Isolated Communication interface. Figure 2.6 shows the installation of the removable RS-232 Isolated Communication card.

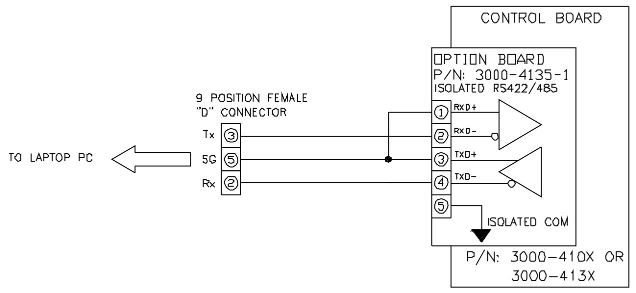
2.4 Ethernet Operation

Ethernet is a local area network technology that transmits information between devices at speeds of 10 and 100 mbps. The word Ethernet refers to hardware called out in IEEE specification 802.3 and has become an increasingly popular medium for communication in industrial environments. The protocols are implementation of TCP and UDP transport used with Ethernet hardware. This allows many different applications to run over the same network and the same cables. Thus, webservers and email run on the same physical connection courtesy of TCP/IP.

The Ethernet Gateway Device is used to interface Phoenix AC Drives to Ethernet Network. The figure below shows a typical network connection of the Phoenix AC Drives using a Ethernet Gateway Device.

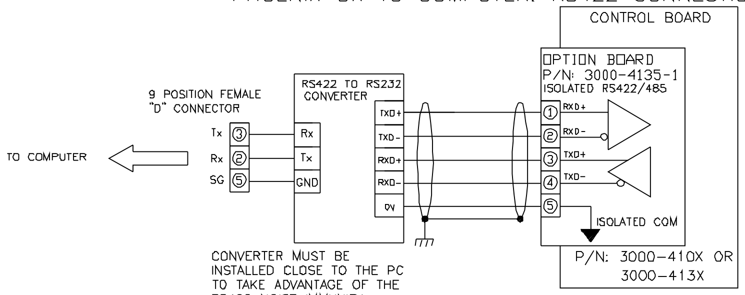


PHOENIX DX TO LAPTOP PC RS232 CONNECTION



3

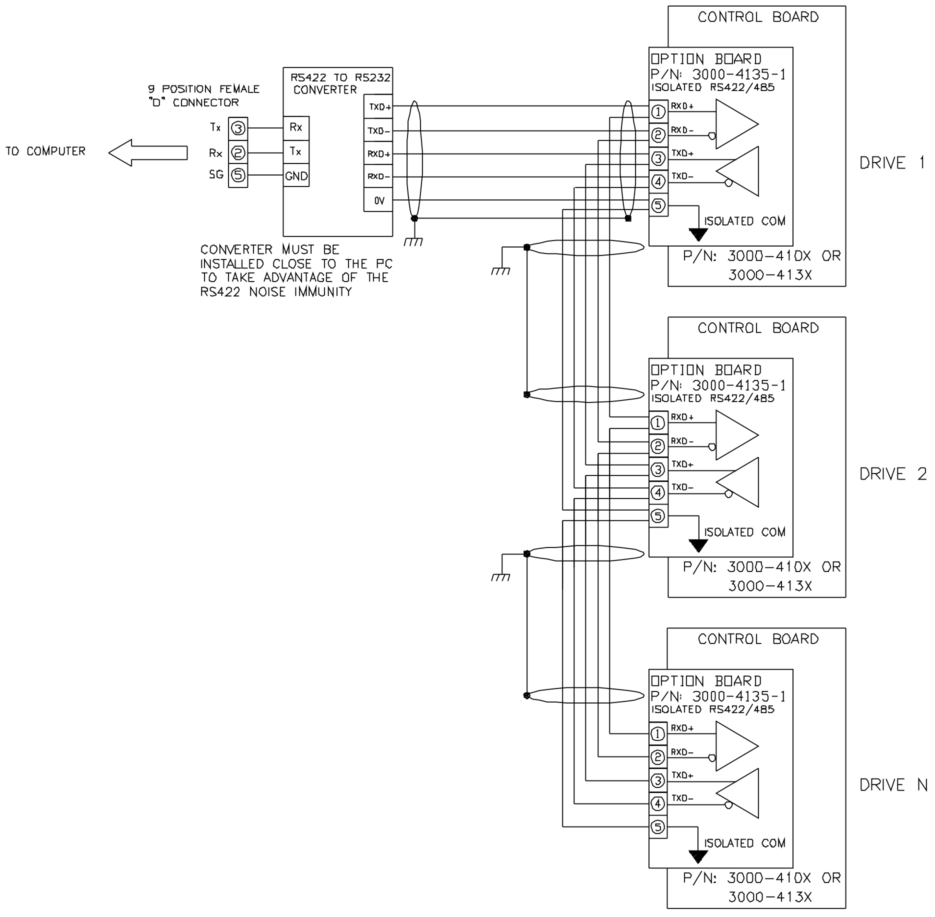
PHOENIX DX TO COMPUTER: RS422 CONNECTION



CONVERTER MUST BE INSTALLED CLOSE TO THE PC TO TAKE ADVANTAGE OF THE RS422 NOISE IMMUNITY

2

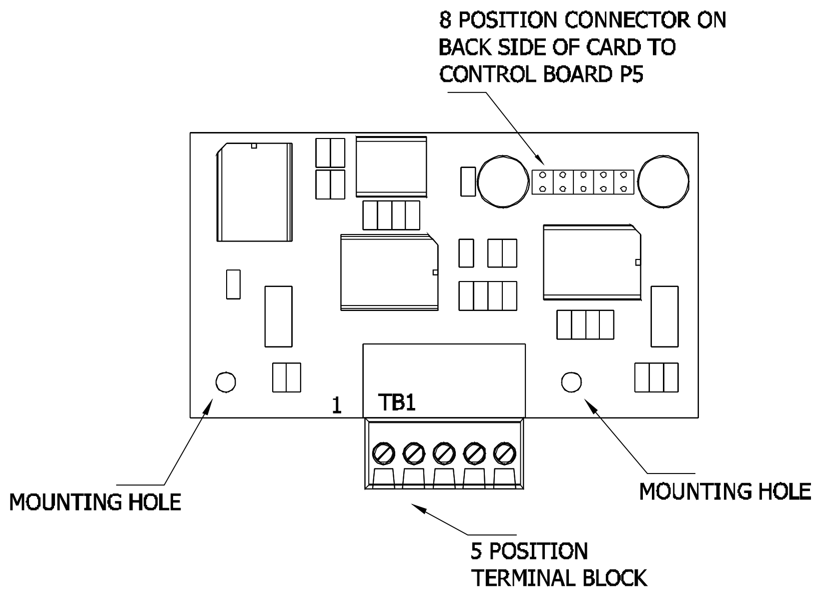
PHOENIX DX TO COMPUTER: RS485 4-WIRE PARTY-LINE CONNECTION



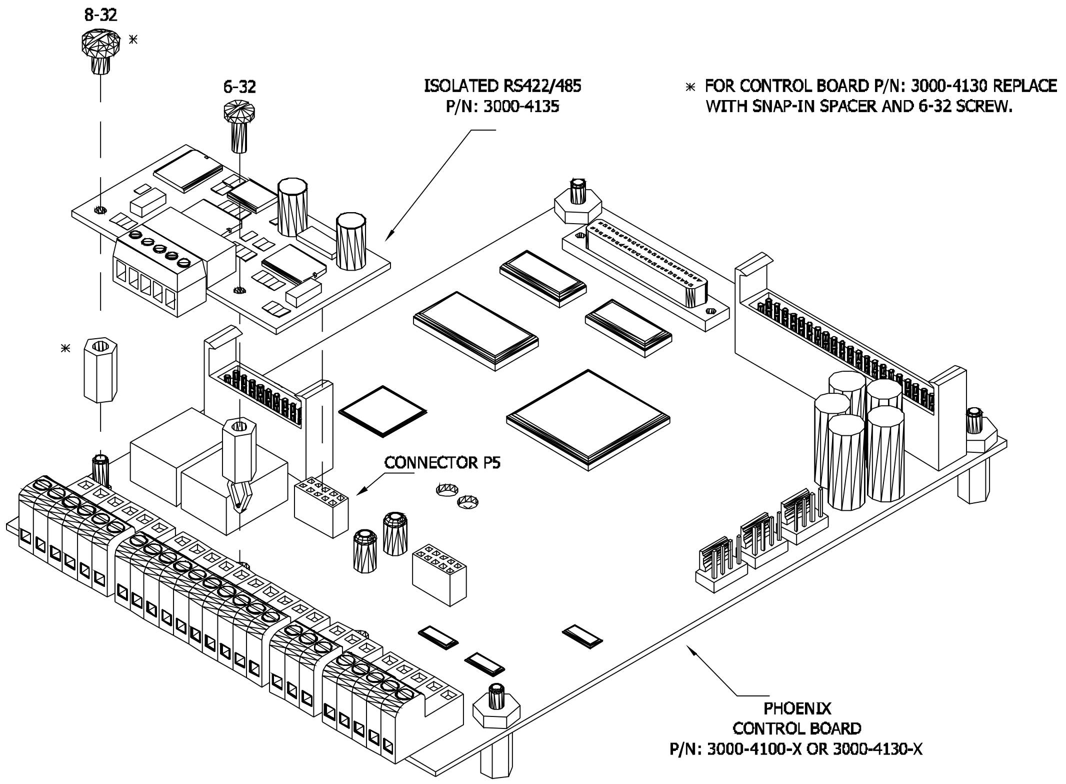
CONVERTER MUST BE INSTALLED CLOSE TO THE PC TO TAKE ADVANTAGE OF THE RS422 NOISE IMMUNITY

L

Isolated Communication Card 3000-4135-1

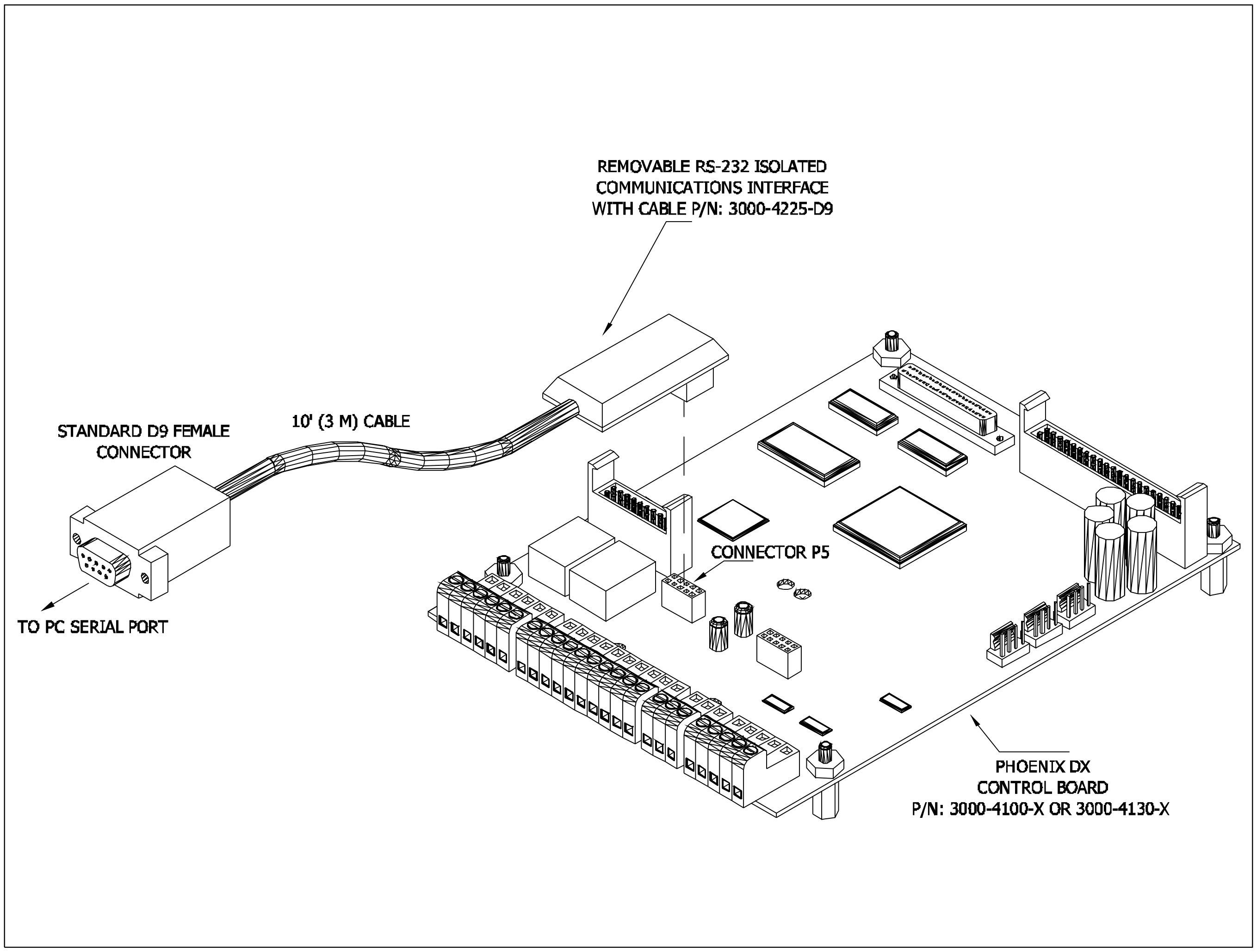

8 POSITION CONNECTOR ON BACK SIDE OF CARD TO CONTROL BOARD P5
1
TB1
MOUNTING HOLE
MOUNTING HOLE
5 POSITION TERMINAL BLOCK

TB1 CONNECTION DATA:
MAXIMUM TORQUE: 2.2 Lb-in [0.25 N-m]
WIRE SIZE: 26-16 AWG [0.14 - 1 mm]


8-32
*
6-32
ISOLATED RS422/485
P/N: 3000-4135
* FOR CONTROL BOARD P/N: 3000-4130 REPLACE WITH SNAP-IN SPACER AND 6-32 SCREW.
*
CONNECTOR P5
PHOENIX
CONTROL BOARD
P/N: 3000-4100-X OR 3000-4130-X

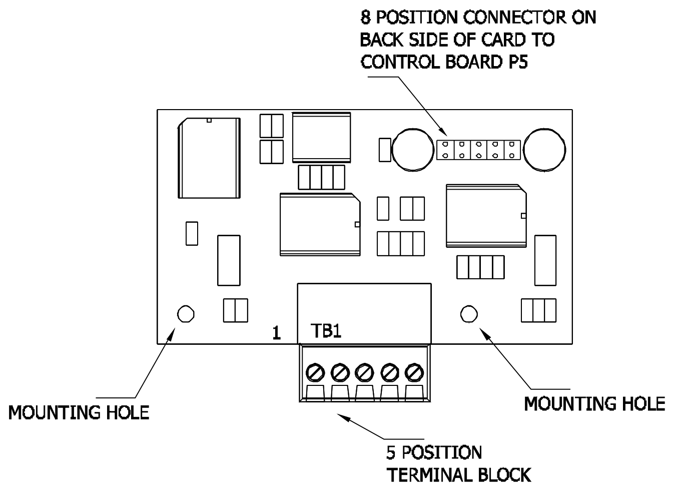
Installation of Isolated RS-422/485 Board
(with Control Board 3000-4100 & 3000-4130)
Figure 2.4

Removable USB/RS-485 Isolated Communications Interface with Cable
P/N: 3000-4226-USB

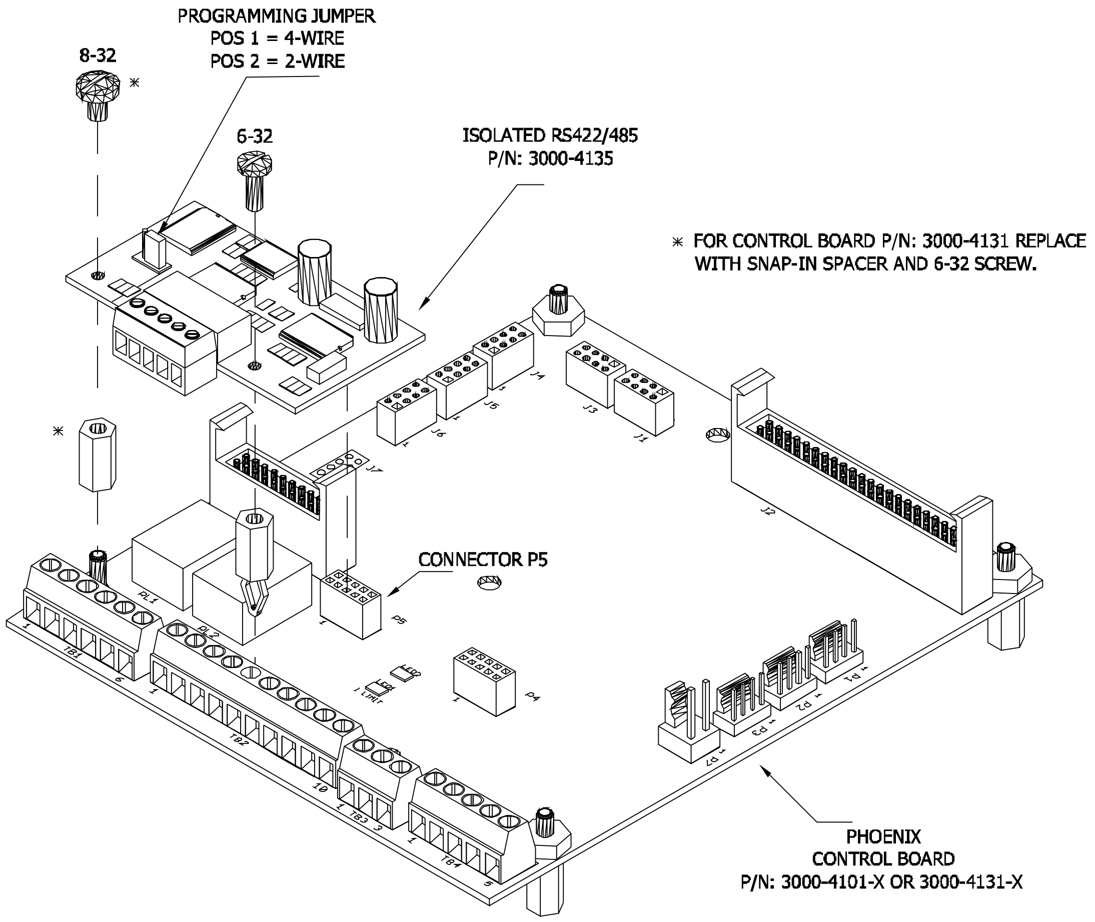


Installation of Removable USB-RS-485 Isolated Communications Interface with Cable
P/N: 3000-4226-USB (with Control Board 3000-4100 & 3000-4130)
Figure 2.5

Isolated Communication Card 3000-4135

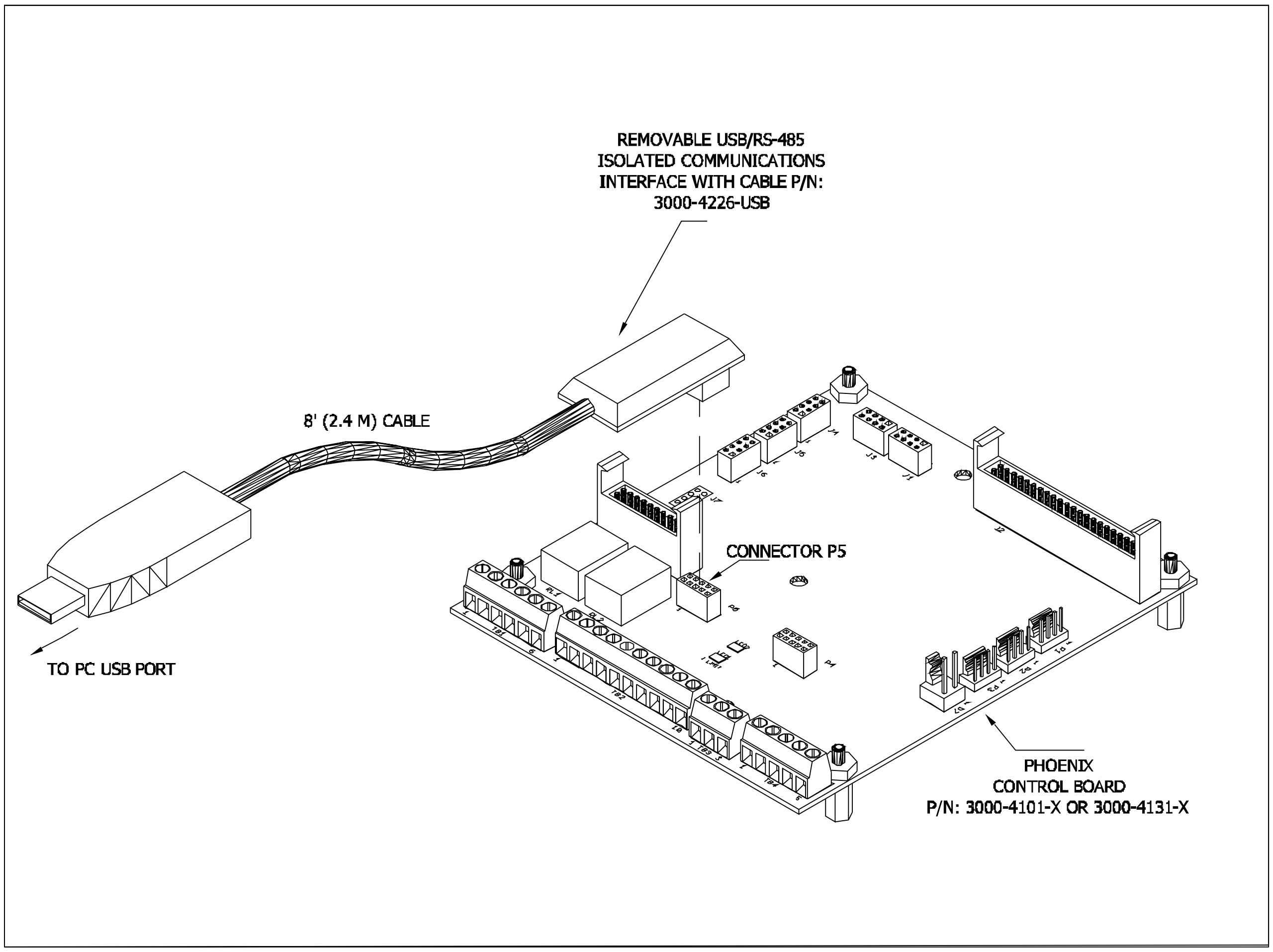


TB1 CONNECTION DATA:
MAXIMUM TORQUE: 2.2 Lb-in [0.25 N-m]
WIRE SIZE: 26-16 AWG [0.14 - 1 mm]



Installation of Isolated RS-422/485 Board
(with Control Board 3000-4101 & 3000-4131)
Figure 2.6

Removable USB/RS-485 Isolated Communications Interface with Cable
P/N: 3000-4226-USB



Installation of Removable USB/RS-485 Isolated Communications Interface with Cable
P/N: 3000-4226-USB (with Control Board 3000-4101 & 3000-4131)
Figure 2.7

END OF HARDWARE INTERFACE

3.0 MODBUS RTU PROTOCOL DESCRIPTION

3.1 Introduction Modbus Protocol

The common language used by Phoenix AC Drives is the Modbus protocol. This protocol defines a message structure that Phoenix AC Drives will recognize and use.

The Modbus protocol provides the internal standard that the Phoenix AC Drives use for parsing messages. During communications on a Modbus network, the protocol determines how each drive will know its device address, recognize a message addressed to it, determine the kind of action to be taken, and extract any data or other information contained in the message. If a reply is required, the drive will construct the reply message and send it using Modbus protocol.

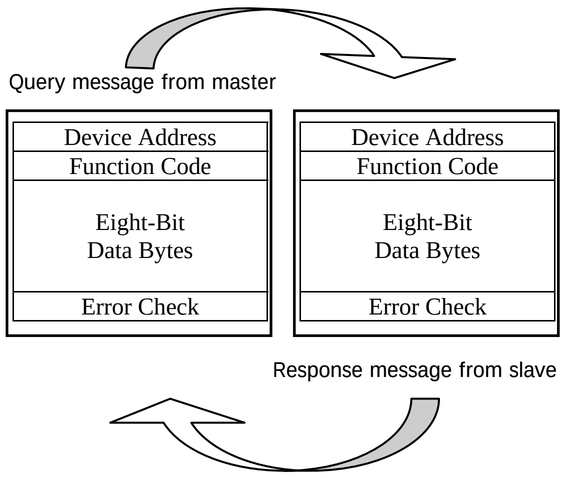
3.1.1 Transaction on Modbus Networks

Controllers communicate using a master-slave technique, in which only one device (the master) can initiate transactions (queries). The other devices (the slaves) respond by supplying the requested data to the master, or by taking the action requested in the query. Typical master devices include host processors. Typical slaves include Phoenix AC Drives.

The master can address individual slaves, or can initiate a broadcast message to all slaves. Slaves return a message (response) to queries that are addressed to them individually. Responses are not returned to broadcast queries from the master.

The Modbus protocol establishes the format for the master's query by placing into it the device (or broadcast) address, a function code defining the requested action, any data to be sent, and an error-checking field. The slave's response message is also constructed using Modbus protocol. It contains fields confirming the action taken, any data to be returned, and an error-checking field. If an error occurred in receipt of the message, or if the slave is unable to perform the requested action, the slave will construct an error message and send it as its response.

3.1.2 The Query-Response Cycle



Master-Slave Query-Response Cycle

The Query

The function code in the query tells the addressed slave device what kind of action to perform. The data bytes contain any additional information that the slave will need to perform the function. For example, function code 03 will query the slave to read holding registers and respond with their contents. The data field must contain the information telling the slave which register to start at and how many registers to read. The error check field provides a method for the slave to validate the integrity of the message contents.

The Response

If the slave makes a normal response, the function code in the response is an echo of the function code in the query. The data bytes contain the data collected by the slave, such as register values or status. If an error occurs, the function code is modified to indicate that the response is an error response, and the data bytes contain a code that describes the error. The error check field allows the master to confirm that the message contents are valid.

3.2 Two Serial Transmission Modes

Devices communicate on standard Modbus networks using either of two transmission modes: ASCII or RTU. Phoenix AC drives use RTU mode, but select the serial port communication parameters (baud rate, parity mode, etc.), during configuration of each drive. The mode and serial parameters must be the same for all devices on a Modbus network.

3.2.1 RTU Mode

The Phoenix AC drive are setup to communicate on a Modbus network using RTU (Remote Terminal Unit) mode, each eight-bit byte in a message contains two four-bit hexadecimal characters. The main advantage of this mode is that its greater character density allows better data throughput than ASCII for the same baud rate. Each message must be transmitted in a continuous stream.

Coding System

Eight-bit binary, hexadecimal 0...9, A...F

Two hexadecimal characters contained in each eight-bit field of the message.

Bits per Byte

1 start bit

8 data bits, least significant bit sent first

1 bit for even/odd parity-no bit for no parity

1 stop bit if parity is used-2 bits if no parity

Error Check Field

Cyclical Redundancy Check (CRC)

3.2.2 ASCII Mode

The Phoenix AC drive does not incorporate ASCII (American Standard Code for Information Interchange) mode. But, if a device is using ASCII mode, each eight-bit byte in a message is sent as two ASCII characters. The main advantage of this mode is that it allows time intervals of up to one second to occur between characters without causing an error.

Coding System

Hexadecimal, ASCII characters 0... 9, A...F

One hexadecimal character contained in each ASCII character of the message.

Bits per Byte

1 start bit

7 data bits, least significant bit sent first

1 bit for even/odd parity-no bit for no parity

1 stop bit if parity is used-2 bits if no parity

Error Check Field

Longitudinal Redundancy Check (LRC)

3.3 Modbus Message Framing

In either of the two serial transmission modes (ASCII or RTU), a Modbus message is placed by the transmitting device into a frame that has a known beginning and ending point. This allows receiving devices to begin at the start of the message, read the address portion and determine which device is addressed (or all devices, if the message is broadcast), and to know when the message is completed. Partial messages can be detected and errors can be set as a result.

3.3.1 RTU Framing

In RTU mode, messages start with a silent interval of at least 3.5 character times. This is most easily implemented as a multiple of character times at the baud rate that is being used on the network (shown as T1 – T2 – T3 – T4 in the figure below). The first field then transmitted is the device address.

The allowable characters transmitted for all fields are hexadecimal 0...9, A...F. Networked devices monitor the network bus continuously, including during the silent intervals. When the first field (the address field) is received, each device decodes it to find out if it is the addressed device.

Following the last transmitted character, a similar interval of at least 3.5 character times marks the end of the message. A new message can begin after this interval.

The entire message frame must be transmitted as a continuous stream. If a silent interval of more than 1.5 character times occurs before completion of the frame, the receiving device flushes the incomplete message and assumes that the next byte will be the address field of a new message.

Similarly, if a new message begins earlier that 3.5 character times following a previous message, the receiving device will consider it a continuation of the previous message. This will set an error, as the value in the final CRC field will not be valid for the combined messages. A typical message frame is shown below.

START	ADDRESS	FUNCTION	DATA	CRC CHECK	END
T1-T2-T3-T4	8 BITS	8 BITS	N X 8BITS	16 BITS	T1-T2-T3-T4

3.3.2 ASCII Framing

In ASCII mode, which is not incorporated in Phoenix AC drives, messages start with a colon (: character (ASCII 3A hex), and end with a carriage return-line feed (CRLF) pair (ASCII 0D and 0A hex).

The allowable characters transmitted for all other fields are hexadecimal 0... 9, A... F. Networked devices monitor the network bus continuously for the colon character. When one is received, each device decodes the next field (the address field) to find out if it is the addressed device.

Intervals of up to one second can elapse between characters within the message. If a greater interval occurs, the receiving device assumes an error has occurred. A typical message frame is shown below.

START	SLAVE ADDRESS	FUNCTION	DATA	LRC CHECK	END
1 CHAR :	2 CHARS	2 CHARS	N CHARS	2 CHARS	2 CHARS CRLF

3.3.3 How the Address Field is Handled

The address field of a message frame contains eight bits (RTU). Valid slave device addresses are in the range of 0...247 decimal. The individual slave devices are assigned addresses in the range of 1...247. A master addresses a slave by placing the slave address in the address field of the message. When the slave sends its response, it places its own address in this address field of the response to let the master know which slave is responding.

Address 0 is used for the broadcast address, which all slave devices recognize. When Modbus protocol is used on higher level networks, broadcasts may not be allowed or may be replaced by other methods. For example, Modbus Plus uses a shared global database that can be updated with each token rotation.

3.3.4 How the Function Field is Handled

The function code field of a message frame contains eight bits (RTU). Valid codes are in the range of 1...255 decimal. Of these, some codes are applicable to all Modicon controllers, while some codes apply only to certain models, and others are reserved for future use.

When a message is sent from a master to a slave device the function code field tells the slave what kind of action to perform. Examples are to read the ON/OFF states of a group of discrete coils or inputs; to read the data contents of a group of registers; to read the diagnostic status of the slave; to write to designated coils or registers; or to allow loading, recording, or verifying the program within the slave.

When the slave responds to the master, it uses the function code field to indicate either a normal (error-free) response or that some kind of error occurred (called an exception response). For a normal response, the slave simply echoes the original function code. For an exception response, the slave returns a code that is equivalent to the original function code with its most significant bit set to a logic 1.

For example, a message from master to slave to read a group of holding registers would have the following function code.

0000 0011 (Hexadecimal 03)

If the slave device takes the requested action without error, it returns the same code in its response. If an exception occurs, it returns:

1000 0011 (Hexadecimal 83)

In addition to its modification of the function code for an exception response, the slave places a unique code into the data field of the response message. This tells the master what kind of error occurred, or the reason for the exception.

The master device's application program has the responsibility of handling exception responses. Typical processes are to post subsequent retries of the message, to try diagnostic messages to the slave, and to notify operators.

3.3.5 Contents of the Data Field

The data field is constructed using sets of two hexadecimal digits, in the range of 00 to FF hexadecimal. These can be made from one RTU character.

The data field of messages sent from a master to slave devices contains additional information, which the slave must use to take the action defined by the function code. This can include items like discrete and register addresses, the quantity of items to be handled, and the count of actual data bytes in the field.

For example, if the master requests a slave to read a group of holding registers (function code 03), the data field specifies the starting register and how many registers are to be read. If the master writes to a group of registers in the slave (function code 10 hexadecimal), the data field specifies the starting register, how many registers to write, the count of data bytes to follow in the data field, and the data to be written into the registers.

If no error occurs, the data field of a response from a slave to a master contains the data requested. If an error occurs, the field contains and exception code that the master application can use to determine the next action to be taken.

The data field can be nonexistent (of zero length) in certain kinds of messages. For example, in a request from a master device for a slave to respond with its communications event log (function code 0B hexadecimal), the slave does not require any additional information. The function code alone specifies the action.

3.3.6 Contents of the Error Checking Field

RTU Mode

When RTU mode is used for character framing, the error checking field contains a 16-bit value implemented as two eight-bit bytes. The error check value is the result of a Cyclical Redundancy Check calculation performed on the message contents.

The CRC field is appended to message as the last field in the message. When this is done, the low-order byte of the field is appended first, followed by the high-order byte. The CRC high-order byte is the last byte to be sent in the message.

3.3.7 How Characters are Transmitted Serially

With RTU character framing, the bit sequence is:

With Parity Checking

Start	1	2	3	4	5	6	7	8	Par	Stop

Without Parity Checking

Start	1	2	3	4	5	6	7	8	Stop	Stop

Bit Order (RTU)

3.4 Error Checking Methods

Standard Modbus serial networks use two kinds of error checking. Parity checking (even or odd) can be optionally applied to each character. Frame checking (LRC or CRC) is applied to the entire message. Both the character check and message frame check are generated in the master device and applied to the message content before transmission. The slave checks each character and the entire message frame during receipt.

The master is configured by the used to wait for a predetermined timeout interval before aborting the transaction. This interval is set to be long enough for any slave to respond normally. If the slave detects a transmission error, the message will not be acted upon. The slave will not construct a response to the master. Thus the timeout will expire and allow the master's program to handle the error.

3.4.1 Parity Checking

Users can configure Phoenix AC drives for Even or Odd Parity checking, or for No Parity checking. This will determine how the parity bit will be set in each character.

If either Even or Odd Parity is specified, the quantity of 1 bit will be counted in the data portion of each character (eight for RTU). The parity bit will then be set to a 0 or 1 to result in an Even or Odd total of 1 bits. For example, these eight data bits are contained in an RTU character frame:

1100 0101

The total quantity of 1 bits in the frame is four. If Even Parity is used, the frame's parity bit will be a 0, making the total quantity of 1 bits still an even number (four). If Odd Parity is used, the parity bit will be a 1, making an odd quantity (five).

When the message is transmitted, the parity bit is calculated and applied to the frame of each character. The receiving device counts the quantity of 1 bits and sets an error if they are not the same as configured for that device (all devices on the Modbus network must be configured to use the same parity check method).

Note that Parity Checking can only detect an error if an odd number of bits are picked up or dropped in a character frame during transmission. For example, if Odd Parity checking is employed, and two 1 bits are dropped form a character containing three 1 bits, the result is still an odd count of 1 bits.

If No Parity checking is specified, no parity bit is transmitted and no parity check can be made. An additional stop bit is transmitted to fill out the character frame.

3.4.2 CRC Checking

In RTU mode, messages include an error-check field that is based on a CRC method. The CRC field checks the contents of the entire message. It is applied regardless of any parity check method used for the individual characters of the message.
The CRC field is two bytes, containing a 16-bit binary value. The CRC value is calculated by the transmitting device, which appends the CRC to the message. The receiving device recalculates a CRC during receipt of the message, and compares the calculated value to the actual value it received in the CRC field. If the two values are not equal, an error results.

The CRC is started by first preloading a 16-bit register to all 1's. Then a process begins of applying successive eight-bits bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start and stop bits, and the parity bit, do not apply to the CRC.

During generation of the CRC, each eight-bit character is exclusive ORed with the register contents. Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. The LSB is extracted and examined. If the LSB was a 1, the register is then exclusive ORed with a preset, fixed value. If the LSB was a 0, no exclusive OR takes place.

This process is repeated until eight shifts have been performed. After the last (eighth) shift, the next eight-bit byte is exclusive ORed with the register's current value, and the process repeats for eight more shifts as described above. The final contents of the register, after all the bytes of the message have been applied, is the CRC value.

When the CRC is appended to the message, the low-order byte is appended first, followed by the high-order byte.

END OF MODBUS RTU PROTOCAL DESCRIPTION

4.0 MODBUS TCP PROTOCAL DESCRIPTION

This section describes the general form of encapsulation of Modbus request and response when carried on the Ethernet network. It is important to note that the structure of the request and response body, from the slave address to the end of the data portion, have exactly the same way out and meaning as in the other Modbus variants.

4.1 Ethernet Frame

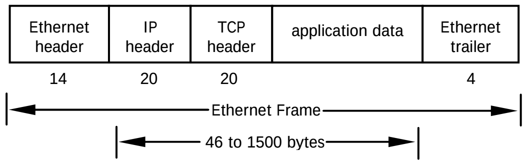


Figure 4.1 Ethernet Frame

The above figure shows an Ethernet frame. The message framing is encapsulated into the application data of the Ethernet frame.

4.2 Modbus TCP Message Framing

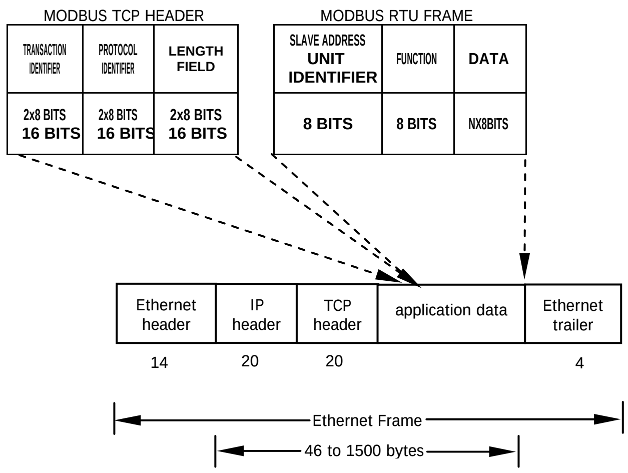


Figure 4.2 Encapsulation of Modbus RTU Frame in Ethernet Frame as Modbus TCP

The Modbus RTU frame request and responses as Modbus TCP message framing is inserted into application data location. The Modbus RTU message frame start, stop, and cRC are not included into the Modbus TCP frame. A Modbus TCP header is added and the slave address is replaced by a unit identifier. The unit identifier is considered part of the header.

4.3 Modbus TCP Header Description

The Modbus TCP Header contains the following fields:

Fields	Length	Description	Client	Server	Contents	
Transaction Identifier	2 Bytes	Identification of a MODBUS Request / Response transaction	Initialized by the client	Recopied by the server from the received request	Byte 0	Usually 0
					Byte 1	Usually 0
Protocol Identifier	2 Bytes	0 = MODBUS protocol	Initialized by the client	Recopied by the server from the received request	Byte 2	0
					Byte 3	0
Length	2 Bytes	Number of following bytes	Initialized by the client (request)	Initialized by the server (Response)	Byte 4	0
					Byte 5	Number of Bytes
Unit Identifier	1 Byte	Identification of a remote slave connected on a serial line or on other buses	Initialized by the client	Recopied by the server from the received request		

The header is 6 bytes long:

- **Transaction Identifier –** It is used for transaction pairing, the Modbus server copies in the response the transaction identifier of the request.
- **Protocol Identifier –** It is used for intra-system multiplexing. The Modbus protocol is identified by the value 0.

- **Length –** The length field is a byte count of the Unit Identifier, Function Code, and Data Fields.
- **Unit Identifier –** This field is used for intra-system routing purpose. It is typically used to communicate to a Modbus serial line slave through a gateway between an Ethernet TCP/IP network and a Modbus serial line. This field is set by the Modbus Client in the request and must be returned with the same value in the response by the server.

All Modbus/TCP Message Frames are sent via TCP on registered port 502.

4.4 Ethernet TCP Message Framing

The Modbus RTU frame request and response as Ethernet TCP message frame is inserted into the application data location. The Modbus RTU message frame start and stop is not included into the Ethernet TCP frame. The format is as shown.

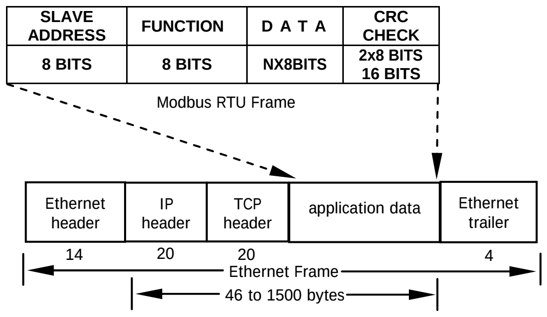


Figure 4.3 Encapsulation of Modbus RTU frame in Ethernet frame as Ethernet TCP

5.0 MODBUS FUNCTION FORMATS

5.1 Field Contents in Modbus Messages

The following tables show examples of a Modbus query and normal response. Both examples show the field contents in hexadecimal, and also show how a message could be framed in RTU mode.

Query

Field Name	Example (hex)	RTU 8-Bit Field
Header		None
Slave Address	06	0000 0110
Function	03	0000 0011
Starting Address Hi	00	0000 0000
Starting Address Lo	6B	0110 1011
No. of Registers Hi	00	0000 0000
No. of Registers Lo	03	0000 0011
Error Check		CRC (16 bits)
Trailer		None
Total Bytes		8

Response

Field Name	Example (hex)	RTU 8-Bit Field
Header		None
Slave Address	06	0000 0110
Function	03	0000 0011
Byte Count	06	0000 0110
Data Hi	02	0000 0010
Data Lo	2B	0010 1011
Data Hi	00	0000 0000
Date Lo	00	0000 0000
Data Hi	00	0000 0000
Data Lo	63	0110 0011
Error Check Trailer		CRC (16 bits) None
Total Bytes		11

The master query is a Read Holding Registers request to slave device address 06. The message requests data from three holding registers, 40108...40110.

Note: The message specifies the starting register address as 0107 (006B hex).

The slave response echoes the function code, indicating this is a normal response. The Byte Count field specifies how many eight-bit data items are being returned. It shows the count of eight-bit bytes to follow in the data.

How to Use the Byte Count Field

When you construct responses in buffers, use a Byte Count value that equals the count of eightbit bytes in your message data. The value is exclusive of all other field contents, including the Byte Count field.

5.2 Function Codes

The listing below shows some function codes supported by Modicon controllers.

Code	Name
01	Read Coil Status
02	Read Input Status
03	Read Holding Registers
04	Read Input Registers
05	Force Single Coil
06	Preset Single Register
07	Read Exception Status
08	Diagnostics

END OF MODBUS FUNCTION FORMATS

6.0 PHOENIX AC DRIVE FUNCTION FORMATS

6.1 Modbus USD Function Formats

The Phoenix AC drive supports five Modbus functions. The drive Modbus function supports only one parameter per query. Each response to the query returns multiple bytes of information. A query's a non-existing function, the response will be an error.

Code	Modbus Function	Phoenix Function
01	Read Coil Status	Read Value Strings
03	Read Holding Register	Read Parameter Value
04	Read Input Register	Read Parameter Name
05	Force Single Coil	Write Parameter Name
06	Preset Single Register	Write Parameter Value

Function Code 01–Read Parameter Value Name

This function reads the parameter value name that describes the specified parameter value on the Phoenix drive. This is for only parameters that have names for values. The value name can be 9 or 16 characters. Broadcast is not supported.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	01
Starting Address Hi	Menu Number	0B
Starting Address Lo	Parameter Number	19
Number of Points Hi	Parameter Value Hi	00
Number of Points Lo	Parameter Value Lo	01
CRC Lo	CRC Lo	2E
CRC Hi	CRC Hi	29

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	01
Byte Count	Byte Count	09
Data 1 Hi	Parameter Name Char 1	39
Data 1 Lo	Parameter Name Char 2	36
Data 2 Hi	Parameter Name Char 3	30
Data 2 Lo	Parameter Name Char 4	30
Data 3 Hi	Parameter Name Char 5	20
Data 3 Lo	Parameter Name Char 6	42
Data 4 Hi	Parameter Name Char 7	41
Data 4 Lo	Parameter Name Char 8	55
Data 5 Hi	Parameter Name Char 9	44
CRC Lo	CRC Lo	3F
CRC Hi	CRC Hi	46

The example above is reading drive 1 parameter M11P25 value name for value of 1. The response is "9600 BAUD".

Function Code 03 – Read Parameter Value

This function reads the data value and decimal point position of the specified parameter on the Phoenix drive. Broadcast is not supported.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Starting Address Hi	Menu Number	05
Starting Address Lo	Parameter Number	09
Number of Points Hi	Don't Care	00
Number of Points Lo	Don't Care	00
CRC Lo	CRC Lo	95
CRC Hi	CRC Hi	04

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Byte Count	Byte Count	03
Data 1 Hi	Parameter Value Hi	02
Data 1 Lo	Parameter Value Lo	58
Data 2 Hi	Decimal Point Location	01
CRC Lo	CRC Lo	1E
CRC Hi	CRC Hi	4E

The example above is reading drive 1 parameter M05P09 BASE MOTOR FREQ. The response is 60.0 hertz.

Decimal point is to the right 1 position:
0258 Hex $\rightarrow$ 600 Decimal $\rightarrow$ 60.0 Hertz

When the parameter has a text string for value (ex. MODBUSUSD), the data value, decimal point, and parameter name value will be return.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Starting Address Hi	Menu Number	0B
Starting Address Lo	Parameter Number	1A
Number of Points Hi	Don't Care	00
Number of Points Lo	Don't Care	00
CRC Lo	CRC Lo	66
CRC Hi	CRC Hi	29

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Byte Count	Byte Count	0C
Data 1 Hi	Parameter Value Hi	00
Data 1 Lo	Parameter Value Lo	00
Data 2 Hi	Decimal Point Location	00
Data 2 Lo	Parameter Name Char 1	4D
Data 3 Hi	Parameter Name Char 2	4F
Data 3 Lo	Parameter Name Char 3	44
Data 4 Hi	Parameter Name Char 4	42
Data 4 Lo	Parameter Name Char 5	55
Data 5 Hi	Parameter Name Char 6	53
Data 5 Lo	Parameter Name Char 7	55
Data 6 Hi	Parameter Name Char 8	53
Data 6 Lo	Parameter Name Char 9	44
CRC Lo	CRC Lo	84
CRC Hi	CRC Hi	EE

The example above is reading drive 1 parameter M11P26 SERIAL PROTOCOL. The response is MODBUSUSD.

Decimal point is to the right 0 position:
0 Hex → 0 Decimal → 0 = MODBUSUSD

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Starting Address Hi	Menu Number	01
Starting Address Lo	Parameter Number	39
Number of Points Hi	Don’t Care	00
Number of Points Lo	Don’t Care	00
CRC Lo	CRC Lo	94
CRC Hi	CRC Hi	3B

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Byte Count	Byte Count	05
Data 1 Hi	Parameter Value Hi 1	EA
Data 1 Lo	Parameter Value Lo 1	60
Data 2 Hi	Decimal Point Location	03
Data 2 Lo	Parameter Value Hi 2	00
Data 3 Hi	Parameter Value Lo 2	00
CRC Lo	CRC Lo	45
CRC Hi	CRC Hi	45

The example above is reading drive 1 parameter M01P57 PRECIS FREQ REF2. This is 32-Bit integer which is supported on DS, ES, and LS series only. The response is 60.000 hertz.

Decimal point is to the right 3 position:
0000 EA60 HEX → 60000 Decimal → 60.000 hertz.

Function Code 04 – Read Parameter Name

This function reads the parameter name that describes the specified parameter on the Phoenix drive. Broadcast is not supported.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	04
Starting Address Hi	Menu Number	05
Starting Address Lo	Parameter Number	09
Number of Points Hi	Don’t Care	00
Number of Points Lo	Don’t Care	00
CRC Lo	CRC Lo	20
CRC Hi	CRC Hi	C4

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	04
Byte Count	Byte Count	10
Data 1 Hi	Parameter Name Char 1	42
Data 1 Lo	Parameter Name Char 2	41
Data 2 Hi	Parameter Name Char 3	53
Data 2 Lo	Parameter Name Char 4	45
Data 3 Hi	Parameter Name Char 5	20
Data 3 Lo	Parameter Name Char 6	4D
Data 4 Hi	Parameter Name Char 7	4F
Data 4 Lo	Parameter Name Char 8	54
Data 5 Hi	Parameter Name Char 9	4F
Data 5 Lo	Parameter Name Char 10	52
Data 6 Hi	Parameter Name Char 11	20
Data 6 Lo	Parameter Name Char 12	46
Data 7 Hi	Parameter Name Char 13	52
Data 7 Lo	Parameter Name Char 14	45
Data 8 Hi	Parameter Name Char 15	51
Data 8 Lo	Parameter Name Char 16	20
CRC Lo	CRC Lo	52
CRC Hi	CRC Hi	D1

The example above is reading drive 1 parameter M05P09 name. The response is “BASE MOTOR FREQ”.

Function Code 05 – Write Parameter Name

This function writes a character in the specified character position for parameter name. The parameter name is only for value of 1 on parameter M11P31 or M11P32. Broadcast is not supported.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	05
Coil Address Hi	Menu Number	0B
Coil Address Lo	Parameter Number	1F
Force Data Hi	Character Number	02
Force Data Lo	Character Value	55
CRC Lo	CRC Lo	3F
CRC Hi	CRC Hi	77

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	05
Coil Address Hi	Menu Number	0B
Coil Address Lo	Parameter Number	1F
Force Data Hi	Character Number	02
Force Data Lo	Character Value	55
CRC Lo	CRC Lo	3F
CRC Hi	CRC Hi	77

This example above is writing the character “U” in position 3 on parameter M11P31 text for value 1.

Function Code 06 – Write Parameter Value

This function writes the data value to the specified parameter of the Phoenix drive. Broadcast is supported.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	06
Register Address Hi	Menu Number	05
Register Address Lo	Parameter Number	09
Preset Data Hi	Data Value Hi	02
Preset Data Lo	Data Value Lo	4E
CRC Lo	CRC Lo	D8
CRC Hi	CRC Hi	50

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	06
Register Address Hi	Menu Number	05
Register Address Lo	Parameter Number	09
Preset Data Hi	Data Value Hi	02
Preset Data Lo	Data Value Lo	4E
CRC Lo	CRC Lo	D8
CRC Hi	CRC Hi	50

This example above is writing data value 59.0 Hertz to parameter M05P09.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	06
Register Address Hi	Menu Number	01
Register Address Lo	Parameter Number	39
Preset Data Hi	Data Value Hi 1	42
Preset Data Lo	Data Value Lo 1	40
	Data Value Hi 2	00
	Data Value Lo 2	0F
CRC Lo	CRC Lo	2F
CRC Hi	CRC Hi	BB

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	06
Register Address Hi	Menu Number	01
Register Address Lo	Parameter Number	39
Preset Data Hi	Data Value Hi 1	42
Preset Data Lo	Data Value Lo 1	40
	Data Value Hi 2	00
	Data Value Lo 2	0F
CRC Lo	CRC Lo	D8
CRC Hi	CRC Hi	50

This example above is writing data value 1000.000 hertz to parameter M01P57. This is a 32-Bit integer which is supported on DS, ES and LS series only.

6.2 Modbus RTU Function Formats

The Phoenix AC drive supports three Modbus functions. The drive Modbus write function supports only one parameter per query. The drive Modbus read function supports multiple parameters per query. A query's a non-existing function, the response will be an error.

Code	Modbus Function	Phoenix Function
03	Read Multiple Register	Read Parameter Value
06	Preset Single Register	Write Parameter Value
16	Preset Multiple Register	Write Parameter Value

Function Code 03 – Read Parameter Value

This function reads the data value the specified parameters on the Phoenix drive. Up to a maximum of sixteen parameter values can be returned in the response. Broadcast is not supported.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Starting Address Hi	Menu Number	05
Starting Address Lo	Parameter Number	08
Number of Points Hi	Number of Parameters Hi	00
Number of Points Lo	Number of Parameters Lo	03
CRC Lo	CRC Lo	84
CRC Hi	CRC Hi	C5

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Byte Count	Byte Count	06
Data 1 Hi	Parameter 1 Value Hi	02
Data 1 Lo	Parameter 1 Value Lo	3F
Data 2 Hi	Parameter 2 Value Hi	02
Data 2 Lo	Parameter 2 Value Lo	58
Data 3 Hi	Parameter 3 Value Hi	00
Data 3 Lo	Parameter 3 Value Lo	01
CRC Lo	CRC Lo	75
CRC Hi	CRC Hi	39

The example above is reading drive 1 parameter M05P08 BASE MOTOR VOLT, M05P09 BASE MOTOR FREQ, M05P10 NUMBER OF POLES. The response is 575 volts, 60.0 hertz, 4 poles.

Decimal point is to the right 0 position:
023F Hex $\rightarrow$ 575 Decimal $\rightarrow$ 575 Volts

Decimal point is to the right 1 position:
0258 Hex $\rightarrow$ 600 Decimal $\rightarrow$ 60.0 Hertz

Decimal point is to the right 0 position:
0001 Hex $\rightarrow$ 1 Decimal $\rightarrow$ 4 Poles

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Starting Address Hi	Menu Number	01
Starting Address Lo	Parameter Number	39
Number of Points Hi	Number of Parameters Hi	00
Number of Points Lo	Number of Parameters Lo	02
CRC Lo	CRC Lo	15
CRC Hi	CRC Hi	FA

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	03
Byte Count	Byte Count	04
Data 1 Hi	Parameter 1 Value Hi	EA
Data 1 Lo	Parameter 1 Value Lo	60
Data 2 Hi	Parameter 2 Value Hi	00
Data 2 Lo	Parameter 2 Value Lo	00
CRC Lo	CRC Lo	CE
CRC Hi	CRC Hi	35

The example above is reading drive 1 parameter M01P57 PRECIS FREQ REF2. This is 32-Bit integer which is supported on DS, ES, and LS series only. The response is 60.000 hertz.
0000 EA60 Hex → 60000 Decimal → 60.000 hertz.

Function Code 06 – Write Parameter Value

This function writes the data value to the specified parameter of the Phoenix drive. Broadcast is supported.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	06
Register Address Hi	Menu Number	05
Register Address Lo	Parameter Number	09
Preset Data Hi	Data Value Hi	02
Preset Data Lo	Data Value Lo	4E
CRC Lo	CRC Lo	D8
CRC Hi	CRC Hi	50

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	06
Register Address Hi	Menu Number	05
Register Address Lo	Parameter Number	09
Preset Data Hi	Data Value Hi	02
Preset Data Lo	Data Value Lo	4E
CRC Lo	CRC Lo	D8
CRC Hi	CRC Hi	50

This example above is writing data value 59.0 Hertz to parameter M05P09.

Function Code 16 – Write Parameter Value

This function writes the data value to the specified parameter of the Phoenix Drive. Broadcast is supported.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	10
Starting Address Hi	Menu Number	05
Starting Address Lo	Parameter Number	09
Number of Register Hi	Number of Parameters Hi	00
Number of Register Lo	Number of Parameters Lo	01
Byte Count	Byte Count	02
Data 1 Hi	Parameter Value Hi	00
Data 1 Lo	Parameter Value Lo	64
CRC Lo	CRC Lo	F2
CRC Hi	CRC Hi	22

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	10
Starting Address Hi	Menu Number	05
Starting Address Lo	Parameter Number	09
Number of Register Hi	Don't Care	00
Number of Register Lo	Don't Care	01
CRC Lo	CRC Lo	D1
CRC Hi	CRC Hi	07

This example above is writing data value 10.0 Hertz to parameter M05P09.

Query

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	10
Starting Address Hi	Menu Number	01
Starting Address Lo	Parameter Number	39
Number of Register Hi	Number of Parameters Hi	00
Number of Register Lo	Number of Parameters Lo	02
Byte Count	Byte Count	04
Data 1 Hi	Parameter Value Hi 1	42
Data 1 Lo	Parameter Value Lo 1	40
Data 2 Hi	Parameter Value Hi 2	00
Data 2 Lo	Parameter Value Lo 2	0F
CRC Lo	CRC Lo	68
CRC Hi	CRC Hi	E9

Response

Modbus Format	Phoenix Format	Example
Slave Address	Drive Address	01
Function Code	Function Code	10
Starting Address Hi	Menu Number	01
Starting Address Lo	Parameter Number	39
Number of Register Hi	Don't Care	00
Number of Register Lo	Don't Care	02
CRC Lo	CRC Lo	90
CRC Hi	CRC Hi	37

The example above is writing data value 1000.000 hertz to parameter M01P57. This is 32-Bit integer which is supported on DS, ES, and LS series only.

6.3 Modbus USD Alternate Function Formats

When in Modbus RTU protocol mode, you can use Modbus USD function formats by adding 64 (0x40 hex) to the function code.

Code	Phoenix Function
65	Read Value Strings
67	Read Parameter Value
68	Read Parameter Name
69	Write Parameter Name
70	Write Parameter Value

Function Code 67 - Read Parameter Value

This function reads the data value and decimal point position of the specified parameter on the Phoenix drive. Broadcast is not supported.

Query

Phoenix Format	Example
Drive Address	01
Function Code	43
Menu Number	05
Parameter Number	09
Don’t Care	00
Don’t Care	00
CRC Lo	94
CRC Hi	CB

Response

Phoenix Format	Example
Drive Address	01
Function Code	43
Byte Count	03
Parameter Value Hi	02
Parameter Value Lo	58
Decimal Point Location	01
CRC Lo	1F
CRC Hi	81

The example above is reading drive 1 parameter M05P09 BASE MOTOR FREQ. The response is 60.0 hertz.

Decimal point is to the right 1 position:
0258 Hex $\rightarrow$ 600 Decimal $\rightarrow$ 60.0 Hertz

END OF PHOENIX AC DRIVE FUNCTION FORMATS

7.0 EXCEPTION RESPONSES

Except for broadcast messages, when a master device sends a query to a slave device it expects a normal response. One of four possible events can occur from the master's query:

If the slave device receives the query without a communication error, and can handle the query normally, it returns a normal response.

If the slave does not receive the query due to a communication error, no response is returned. The master program will eventually process a timeout condition for the query.

If the slave receives the query, but detects a communication error (parity or CRC), no response is returned. The master program will eventually process a timeout condition for the query.

If the slave receives the query without a communication error, but cannot handle it (for example, if the request is to read a nonexistent coil or register), the slave will return an exception response informing the master of the nature of the error.

The exception response message has two fields that differentiate it from a normal response.

Function Code Field

In a normal response, the slave echoes the function code of the original query in the function code field of the response. All function codes have a most significant bit (MSB) of 0 (their values are all below 80 hexadecimal). In an exception response, the slave sets the MSB of the function code to 1. This makes the function code value in an exception response exactly 80 hexadecimal higher than the value would be for a normal response.

With the function code's MSB set, the master's application program can recognize the exception response and can examine the data field for the exception code.

Data Field

In a normal response, the slave may return data or statistics in the data field (any information that was requested in the query). In an exception response, the slave returns an exception code in the data field. This defines the slave condition that caused the exception. Here is an example of a master query and slave exception response. The field examples are shown in hexadecimal.

Query

Byte	Contents	Example
1	Slave Address	01
2	Function	03
3	Starting Address Hi	04
4	Starting Address Lo	A1
5	Number of Points Hi	00
6	Number of Points Lo	01
7	CRC Lo	D4
8	CRC Hi	D8

Exception Response

Byte	Contents	Example
1	Slave Address	01
2	Function	83
3	Exception Code	02
4	CRC Lo	C0
5	CRC Hi	F1

In this example, the master addresses a query to slave device 01. The function code (03) is for a Read Holding Resister operation. It requests the contents of the holding resister at address 1245 (04A1 hex).

If the holding resister address is nonexistent in the slave device, the slave will return the exception response with the exception code shown (02). This specifies an illegal data address for the slave.

7.1 Exception Codes

Code	Name
01	ILLEGAL FUNCTION
02	ILLEGAL DATA ADDRESS
03	ILLEGAL DATA VALUE
04	SLAVE DEVICE FAILURE
05	ACKNOWLEDGE
06	SLAVE DEVICE BUSY
07	NEGATIVE ACKNOWLEDGE
08	MEMORY PARITY ERROR

The Phoenix AC drive exception response uses some of the existing Modbus exception codes and also has three new exception codes. The Modbus RTU Protocol only uses exception codes 01, 02 and 03.

Exception Code	Modbus Name	Phoenix Name
01	Illegal Function	Illegal Function
02	Illegal Data Address	Invalid Menu and/or Parameter Number
03	Illegal Data Value	Illegal Data Value (Invalid Data, Read Only, Value Out of Range, Drive Running, Ect.)
10	N/A	Bad CRC
11	N/A	More data needed for valid message
12	N/A	To much data needed for valid message

END OF EXCEPTION RESPONSE

8.0 CRC GENERATION

The Cyclical Redundancy Check (CRC) field is two bytes, containing a 16-bit binary value. The CRC value is calculated by the transmitting device, which appends the CRC to the message. The receiving device recalculates a CRC during receipt of the message, and compares the calculated value to the actual value it received in the CRC field. If the two values are not equal, an error results.

The CRC is started by first preloading a 16-bit register to all 1's. Then a process begins of applying successive eight-bit bytes of the message to the current contents of the register. Only the eight bits of data in each character are used for generating the CRC. Start and stop bits, and the parity bit, do not apply to the CRC.

During generation of the CRC, each eight-bit character is exclusive ORed with the register contents. The result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. The LSB is extracted and examined. If the LSB was a 1, the register is then exclusive ORed with a preset, fixed value. If the LSB was a 0, no exclusive OR takes place.

This process is repeated until eight shifts have been performed. After the last (eighth) shift, the next eight-bit character is exclusive ORed with the register's current value, and the process repeats for eight more shifts as described above. The final contents of the register, after all the characters of the message have been applied, is the CRC value.

Generating a CRC

Step 1 Load a 16-bit register with FFFF hex (all 1's). Call this the CRC register.

Step 2 Exclusive OR the first eight-bit byte of the message with the low order byte of the 16-bit CRC register, putting the result in the CRC register.

Step 3 Shift the CRC register on bit to the right (toward the LSB), zerofilling the MSB. Extract and examine the LSB.

Step 4 If the LSB is 0, repeat Step 3 (another shift). If the LSB is 1, Exclusive OR the CRC register with the polynomial value A001 hex (1010 0000 0000 0001).

Step 5 Repeat Steps 3 and 4 until eight shifts have been performed. When this is done, a complete eight-bit byte will have been processed.

Step 6 Repeat Steps 2...5 for the next eight-bit byte of the message. Continue doing this until all bytes have been processed.

Result The final contents of the CRC register is the CRC value.

Step 7 When the CRC is placed into the message, its upper and lower bytes must be swapped as described below.

Placing the CRC into the Message

When the 16-bit CRC (two eight-bit bytes) is transmitted in the message, the low order byte will be transmitted first, followed by the high order byte-e.g., if the CRC value is 1241 hex (0001 0010 0100 0001):

Addr	Func	Data Count	Data	Data	Data	Data	CRC Lo	CRC Hi
							41	12

CRC Byte Sequence

Example

An example of a C language function performing CRC generation is shown on the following pages. All of the possible CRC values are preloaded into two arrays, which are simply indexed as the function increments through the message buffer. One array contains all of the 256 possible CRC values for the high byte of the 16-bit CRC field, and the other array contains all of the values for the low byte,

Indexing the CRC in this way provides faster execution than would be achieved by calculating a new CRC value with each new character from the message buffer.

Note: This function performs the swapping of the high/low CRC bytes internally. The bytes are already swapped in the CRC value that is returned from the function. Therefore the CRC value returned from the function can be directly placed into the message for transmission.

The function takes two arguments:

```
unsigned char  puchMsg;           /* A pointer to the message buffer  */
unsigned short usDataLen;          /* The quantity of bytes in the message buffer  */
```

The function returns the CRC as a type unsigned short.

CRC Generation Function

```
unsigned short CRC16 (puchMsg, usDataLen)

unsigned char   *puchMsg;                          /* message to calculate CRC upon  */
unsigned short usDataLen;                          /* quantity of bytes in message        */
{
     unsigned char uchCRCHi = 0xFF;                /* high CRC byte initialized            */
     unsigned char uchCRCLo = 0xFF;                /* low CRC byte initialized             */
     unsigned uIndex;                                /* will index into CRC lookup          */

     while( usDataLen--)                            /* pass through message buffer        */
     {
          uIndex = uchCRCHi ^ *puchMsg++;          /* calculate the CRC                    */
          uchCRCHi = uchCRCLo ^ auchCRCHi[ uIndex};
          uchCRCLo = auchCRCLo[ uIndex];
     }
     return( uchCRCHi << 8 | uchCRCLo);
}
```

High Order Byte Table

```
/* Table of CRC values for high-order byte */

static unsigned char auchCRCHi [ ] =
{
0x00,   0xC1,   0x81,   0x40,   0x01,   0xC0,   0x80,   0x41,   0x01,   0xC0,
0x80,   0x41,   0x00,   0xC1,   0x81,   0x40,   0x01,   0xC0,   0x80,   0x41,
0x00,   0xC1,   0x81,   0x40,   0x00,   0xC1,   0x81,   0x40,   0x01,   0xC0,
0x80,   0x41,   0x01,   0xC0,   0x80,   ox41,   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
};
```

Low Order Byte Table

```
/* Table of CRC values for low-order byte */

static unsigned char auchCRCLo [ ] =
{
0x00,  0xC0,  0xC1,  0x01,  0xC3,  0x03,  0x02,  oxC2,  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
};
```

9.0 PARAMETER CONVERSION

9.1 Parameter Coding Format

The Modbus Format for a 16-bit register address is sent as two hexadecimal bytes. The register address Hi is the two digit Menu number and register address Lo is the two digit Parameter number. The example below shows M12P05 parameter coded into the Menu Number and Parameter Number for Phoenix drive. The Menu number 12 decimal is converted into a one byte (two digit representation) 0C hexadecimal. The Parameter number 5 decimal is converted into a one byte (two digit representation) 05 hexadecimal.

Modbus Format	Phoenix Format	Example
Register Address Hi	Menu Number	0C
Register Address Lo	Parameter Number	05

9.2 Parameter to Register Address Conversion

The US Drives Phoenix two digit Menu number and Parameter number sometimes needs to be converted to a register address in third party software programs. The conversion formula is:

(Menu Number * 256) + Parameter Number = Register Address

Examples:

<u>M05P06</u>

(5 * 256) + 6 = 1280 + 6 = 1286
0506 Hex → 1286 Decimal

<u>M12P05</u>

(12 * 256) + 5 = 3072 + 5 = 3077
0C05 Hex → 3077 Decimal

<u>M06P41</u>

(6 * 256) + 41 = 1536 + 41 = 1577
0629 Hex → 1577 Decimal

<u>M17P19</u>

(17 * 256) + 19 = 4352 + 19 = 4371
1113 Hex → 4371 Decimal

END OF PARAMETER CONVERSION

1.0 INTRODUCTION

1.1 Introduction

Connectivity is required either too exchange billing information or business data, or to facilitate remote process and/or machine control. For almost everyone, the TCP/IP is now the core technology for this connectivity. For many, TCP/IP is now the technology of choice for all data transfers, including both video and voice.

The TCP/IP protocol suite allows computers of all sizes, from many different computer vendors, running totally different operating systems, to communicate with each other. TCP/IP stands for Transaction Control Protocol and Internet Protocol. Actually, thought, TCP/IP is a complete protocol suite which includes these protocols, as well as others.

What started in the late 1960's as a government-financed research project into packet switching networks has, in the 1990's, turned into the most widely used form of networking between computers. It is truly an open system in that the definition of the protocol suite and many of its implementations are publicly available at little charge or no charge. It forms the basis for what is called the worldwide Internet, or the Internet, a wide area network (WAN) of more than one million computers that literally spans the globe.

This section provides an overview of the TCP/IP protocol suite.

1.2 Layering

Networking protocols are normally developed in layers, with each layer responsible for a different facet of the communications. A protocol suite, such as TCP/IP, is the combination of different protocols at various layers. TCP/IP is normally considered to be a 4-layer system.

Application	Telnet, TFP, e-mail, etc.
Transport	TCP, UDP
Network	IP, ICMP, IGMP
Link	device driver and interface card

Figure A.1 The four layers of the TCP/IP protocol suite

Each layer has a different responsibility.

1. The link layer, sometimes called the data-link layer or network interface layer, normally includes the device driver in the operating system and the corresponding network interface cad in the computer. Together they handle all the hardware details of physically interfacing with the cable (or whatever type of media is being used).
2. The network layer (sometimes called the internet layer) handles the movement of packets around the network. Routing of packets, for example, takes place here. IP (Internet Protocol), ICMP (Internet Control Message Protocol), and IGMP (Internet Group Management Protocol) provide the network layer in the TCP/IP protocol suite.
3. The transport layer provides a flow of data between two hosts, for the application layer above. In the TCP/IP protocol suite there are two vastly different transport protocols: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

   TCP provides a reliable flow of data between two hosts. It is concerned with things such as dividing the data passed to it from the application into appropriately sized chunks for the network layer below, acknowledging received packets, setting timeouts to make certain the other end acknowledges packets that are sent, and so on. Because this reliable flow of data is provided by the transport layer, the application layer can ignore all these details.

   UDP, on the other hand, provides a much simpler service to the application layer. It just sends packets of data called datagrams from one host to the other, but there is no guarantee that the datagrams reach the other end. Any desired reliability must be added by the application layer.

4. The application layer handles the details of the details of the particular application. There are many common TCP/IP applications that almost every implementation provides:

   - Telnet for remote login,
   - FTP, the File Transfer Protocol,
   - SMTP, the Simple Mail Transfer Protocol, for electronic mail,
   - SNMP, the Simple Network Management Protocol,
   - And many moreIf we have two hosts on a local area network (LAN) such as

an Ethernet, both running FTP, Figure 1.2 shows the protocols involved.

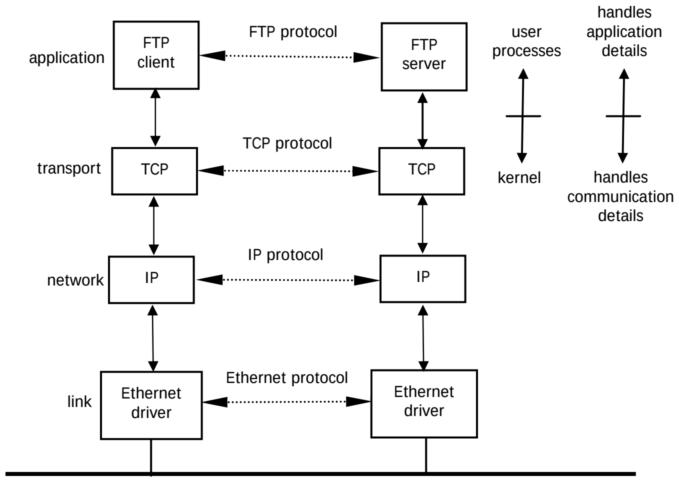


Figure A.2 Two hosts on a LAN running FTP

We have labeled one application box the FTP client and the other the FTP server. Most network applications are designed so that one end is the client and the other side the server. The server provides some type of service to clients, in this case access to files on the server host.

Each layer has one or more protocols for communicating with its peer at the same layer. One protocol, for example, allows the two TCP layers to communicate, and another protocol lets the two IP layers communicate.

On the right side of Figure 1.2 we have noted that normally the application layer is a user process while the lower three layers are usually implemented in the kernel (the operating system).

There is another critical difference between the top layer in Figure 1.2 and the lower three layers. The application layer is concerned with the details of the application and not with the movement of data across the network. The lower three layers know nothing about the application but handle all the communication details.

We show four protocols in Figure 1.2, each at a different layer. FTP is an application layer protocol, TCP is a transport layer protocol, IP is a network layer protocol, and the Ethernet protocols operate at the link layer. The TCP/IP protocol suite is a combination of many protocols. Although the commonly used name for the entire protocol suite is TCP/IP, TCP and IP are only two of the protocols. (An alternative name is the Internet Protocol Suite.)

The purpose of the network interface layer and the application layer are obvious – the former handles the details of the communication media (Ethernet, ring, etc) while the latter handles one specific user application (FTP, Telnet, etc.).

This figure shows some layers and protocols in the TCP/IP protocol suite.

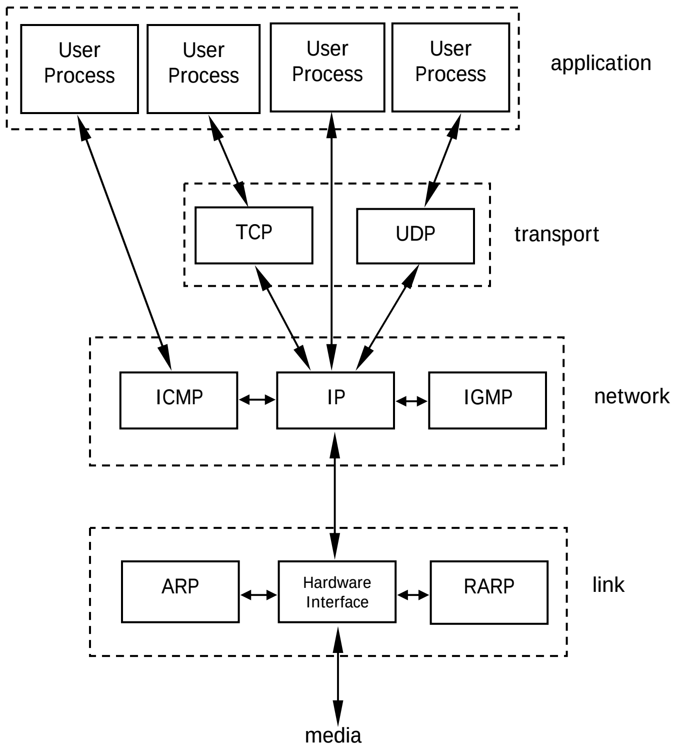


Figure A.3 Various protocols at the different layers in the TCP/IP protocol suite.

TCP and UDP are the two predominant transport layer protocols. Both use IP as the network layer.

2.0 NETWORK LAYER

2.1 IP

IP provides communication between hosts on different kinds of networks (i.e., different data-link implementations such as Ethernet and Token Ring). It is connectionless, unreliable packet delivery service. Connectionless means that there is no handshaking, each packet is independent of any other packet. It is unreliable because there is no guarantee that a packet gets delivered; higher-level protocols must deal with that.

2.2 IP Address

IP defines an addressing scheme that is independent of the underlying physical address (e.g, 48-bit MAC address). IP specifies a unique 32-bit number for each host on a network. This number is known as the Internet Protocol Address, the IP Address or the Internet Address. These terns are interchangeable. Each packet sent across the internet contains the IP address of the source of the packet and the IP address of its destination.

For routing efficiency, the IP address is considered in two parts: the prefix which identifies the physical network, and the suffix which identifies a computer on the network. A unique prefix is needed for each network in an internet. For the global Internet, network numbers are obtained from Internet Service Providers (ISPs). ISPs coordinate with a central organization called the Internet Assigned Number Authority (IANA).

2.3 IP Address Classes

The first four bits of an IP address determine the class of the network. The class specifies how many of the remaining bits belong to the prefix (aka Network ID) and to the suffix (aka Host ID). The first three classes, A, B and C, are the primary network classes.

Class	First 4 Bits	Number of Prefix Bits	Max # of Networks	Number of Suffix Bits	Max # of Hosts Per Network	Range
A	0xxx	7	128	24	16,777,216	0000 to 127.255.255.255
B	10xx	14	16,384	16	65,536	128.0.0.0 to 191.255.255.255
C	110x	21	2,097,152	8	256	192.0.0.0 to 223.255.255.255
D	1110	Multicast				224.0.0.0 to 239.255.255.255
E	1111	Reserved for future use				240.0.0.0 to 255.255.255

When interacting with mere humans, software uses dotted decimal notation; each 8 bits is treated as an unsigned binary integer separated by periods. IP reserves host address 0 to denote a network. 140.211.0.0 denotes the network that was assigned the class B prefix 140.211.

2.4 Netmasks

Netmasks are used to identify which part of the address is the Network ID and which part is the Host ID. This is done by a logical bitwise-AND of the IP address and the netmask. For class A networks the netmask is always 255.0.0.0; for class B networks it is 255.255.0.0 and for class C networks the netmask is 255.255.255.0.

2.5 Subnet Address

All hosts are required to support subnet addressing. While the IP address classes are the convention, IP addresses are typically subnetted to smaller address sets that do not match the class system. The suffix bits are divided into a subnet ID and a host ID. This makes sense for class A and B networks, since no one attaches as many hosts to these networks as is allowed. Whether to subnet and how many bits to use for the subnet ID is determined by the local network administrator of each network.

If subnetting is used, then the netmask will have to reflect this fact. On a class B network with subnetting, the netmask would not be 255.255.0.0. The bits of the Host ID that were used for the subnet would need to be set in the netmask.

2.6 Directed Broadcast Address

IP defines a directed broadcast address for each physical network as all ones in the host ID part of the address. The network ID and the subnet ID must be valid network and subnet values. When a packet is sent to a network's broadcast address, a single copy travels to the network, and then the packet is sent to every host on that network or subnetwork.

2.7 Limited Broadcast Address

If the IP address is all ones (255.255.255.255), this is a limited broadcast address; the packet is addressed to all hosts on the current (sub)network. A router will not forward this type of broadcast to other (sub) networks.

2.8 ICMP

Internet Control Message Protocol is a set of messages that communicate errors and other conditions that require attention. ICMP messages, delivered in IP datagrams, are usually acted on by either IP, TCP or UDP. Some ICMP messages are returned to application protocols.

A common use of ICMP is "pinging" a host. The Ping command (Packet INternet Groper) is a utility that determines whether a specific IP address is accessible. It sends an ICMP echo request and waits for a reply. Ping can be used to transmit a series of packets to measure average round-trip times and packet loss percentages.

3.0 LINK LAYER

3.1 ARP

The Address Resolution Protocol is used to translate virtual addresses to physical ones. The network hardware does not understand the software-maintained IP addresses. IP uses ARP to translate the 32-bit IP address to a physical address that matches the addressing scheme of the underlying hardware (for Ethernet, the 48-bit MAC address).

4.0 THE TRANSPORT LAYER

There are two primary transport layer protocols: Transmission Control Prot
ocol (TCP) and User Datagram Protocol (UDP). They provide end-to-end communication services for applications.

4.1 UDP

This is a minimal service over IP, adding only optional checksumming of data and multiplexing by port number. UDP is often used by applications that need multicast or broadcast delivery, services not offered by TCP. Like IP, UDP is connectionless and works with datagrams.

4.2 TCP

TCP is a connection-oriented transport service; it provides end-to-end reliability, resequencing, and flow control. TCP enables two hosts to establish a connection and exchange streams of data, which are treated in bytes. The delivery of data in the proper order is guaranteed.

TCP can detect errors or lost data and can trigger retransmission until the data is received, complete and without errors.

5.0 The Application Layer

There are many applications available in the TCP/IP suite of protocols. Some of the most useful ones are for sending mail (SMTP), transferring files (FTP), and displaying web pages (HTTP).

Another important application layer protocol is the Domain Name System (DNS). Domain names are significant because they guide users to where they want to go on the Internet.

5.1 DNS

The Domain Name System is a distributed database of domain name and IP address bindings. A domain name is simply an alphanumeric character string separated into segments by periods. It represents a specific and unique place in the "domain name space." DNS makes it possible for us to use identifiers such as zworld.com to refer to an IP address on the Internet. Name servers contain information on some segment of the DNS and make that information available to clients who are called resolvers.

6.0 ETHERNET FRAME

6.1 Ethernet Frame

The term Ethernet generally refers to a standard published in 1982 by Digital Equipment Corp., Intel Corp., and Xerox Corp. It is the predominant form of local area network technology used with TCP/IP today. It uses an access method called CSMA/CD, which stands for Carrier Sense, Multiple Access with Collision Detection.

The Ethernet frame length ranges from 64 bytes to 1518 bytes. The Ethernet frame uses 48-bit (6-byte) destination and source addresses. These are what we call hardware addresses.

The next 2 bytes is the Ethernet type field which identifies the type of data that follows. In the Ethernet frame the data immediately follows the type field. The data length ranges from 46 bytes to 1500 bytes. The minimum data frames for Ethernet is 46 bytes.

The CRC field is a cyclic redundancy check (a checksum) that detects errors in the rest of the frame. (This is also called the FCS or frame check sequence.) The CRC field is 4 bytes.

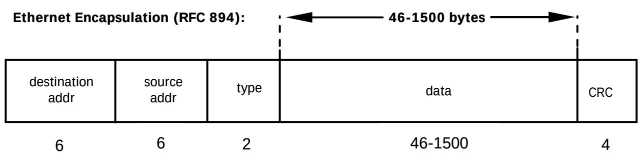


Figure A.4 Ethernet Frame (RFC 894)

6.2 Encapsulation

When an application sends data using TCP, the data is sent down the protocol stack, through each layer, until it is sent as a stream of bits across the network. Each layer adds information to the data by prepending headers (and sometimes adding trailer information) to the data that it receives. Figure 4.2 shows this process. The unit of data that TCP sends to IP is called a TCP segment. The unit of data that IP sends to the network interface is called an IP datagram. The stream of bits that flows across the Ethernet is called a frame.

The numbers at the bottom of the headers and trailer of the Ethernet frame in Figure 4.2 are the typical sizes of the headers in bytes.

A physical property of an Ethernet frame is that the size of its data must be between 46 and 1500 bytes.

> To be completely accurate in Figure 4.2 we should say that the unit of data passed between IP and the network interface is a packet. This packet can be either an IP datagram or a fragment of an IP datagram.

We could draw a nearly identical picture for UDP data. The only changes are that the unit of information that UDP passes to IP is called a UDP datagram, and the size of the UDP header is 8 bytes.

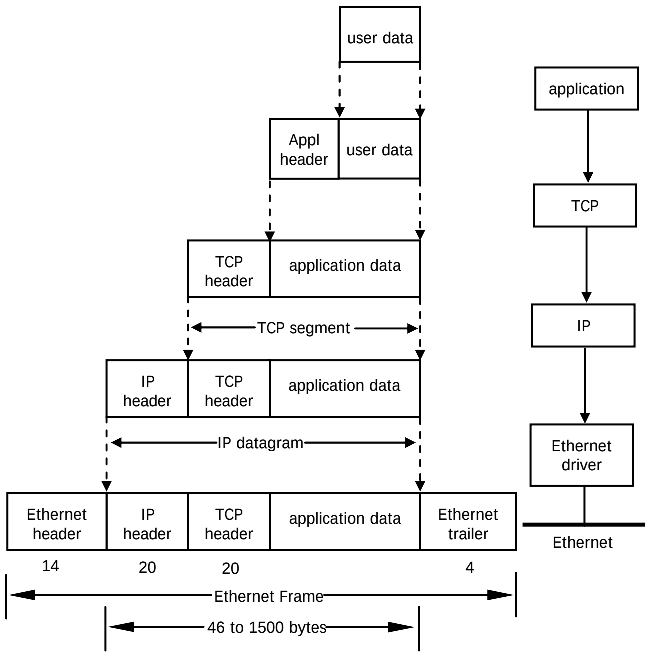


Figure A.5 Encapsulation of data as it goes down the protocol stack

Recall from Figure 1.3 (p. 2) that TCP, UDP, ICMP, and IGMP all send data to IP. IP must add some type of identifier to the IP header that it generates, to indicate the layer to which the data belongs. IP handles this by storing an 8-bit value in its header called the protocol field. A value of 1 is for ICMP, 2 is for IGMP, 6 indicates TCP, and 17 is for UDP.

Similarly, many different applications, can be using TCP or UDP at any one time. The transport layer protocols store an identifier in the headers they generate to identify the application. Both TCP and UDP use 16-bit port numbers to identify applications. TCP and UDP store the source port number and the destination port number in their respective headers.

The network interface sends and receives frames on behalf of IP, ARP, and RARP. There must be some form of identification in the Ethernet header indicating which network layer protocol generated the data. To handle this there is a 16-bit frame type field in the Ethernet header.

"THE HIGH HORSEPOWER DESIGN EXPERTS"

US Drives Inc.
2221 Niagara Falls Boulevard
P.O. Box 281
Niagara Falls, NY 14304-0281
Tel: (716) 731-1606 Fax: (716) 731-1524
Visit us at www.usdrivesinc.com

Products Designed And Manufactured
In The United States Of America

MODP0917 Printed in U.S.A.