Appendix D
Table of Contents

1.0 How to Use this Addenda 4

2.0 Summary of Westnet II Data Highway Commands 5
2.1 Set Highway Address 5
2.2 Determine Remote Address that PCI is addressed to 5
2.3 Read Highway Status 5
2.4 Open Session 5
2.5 Close Session 5
2.6 Read Session Status 5
2.7 Read Local PCI Address Setting 6

3.0 PC1000 Local Area Network Message Structure 8
3.1 Set Unit Address 8
3.2 Read Block 9
3.3 Write Block 9
3.4 Erase Memory 9
3.5 Retest PLC 10
3.5 PC 1000 LAN Timing Diagram 11
3.6 PC 1000 LAN RS-485 Computer Wiring Diagram 12
3.7 PC 1000 LAN RS-232 Computer Wiring Diagram 13

4.0 Modbus™ 14
4.1 Supported CPUs 14
4.2 Message Types 15
4.2.1 Address 15
4.2.2 Function Code 15
4.2.3 Data 16
4.2.4 Error Check 16
4.2.5 Exception Response 16
4.3 Message Structure 17
4.3.1 Read Coil (Output) Status (Function Code 1) 17
4.3.2 Read Input Status (Function Code 2) 17
4.3.3 Read Holding Registers (Function Code 3) 17
4.3.4 Read Input Registers (Function Code 4) 18
4.3.5 Write Single Coil (Function Code 5) 18
4.3.6 Write Single Holding Register (Function Code 6) 18
4.3.7 Read Exception Status (Function Code 7) 18
4.3.8 Loopback Test (Function Code 8) 18
4.3.9 Preset Multiple Registers (Function Code 16 decimal) 18
4.4 Modbus RS-232 Computer Wiring Diagram 19
4.5 Sample Modbus Master program in C 20

5.0 PC500/PC2000 Loader Protocol 47
5.1 Request SYSPAR 47
5.2 Request Data Block Pointers 48
5.4 Request User Memory Starting Addresses 49
5.5 Read Byte (Immediate) 49
5.6 Write Byte (Immediate) 52
5.7 Cabling 52

6.0 HPPC Program Loader Protocol 53

6.1 Cabling 53
6.2 Configuring HPPC 53
6.3 HPPC Memory Map 55
6.3.1 HPPC Data/Operand Field 55
6.3.2 Holding Registers 56
6.4 Asynchronous Protocol 56
6.4.1 Message Structure 57
6.4.2 Address Fields 57
6.4.3 Opcodes 57
6.4.3.1 Read Parameter Table (Opcode 0Dh) 59
6.4.3.2 Read Memory (Opcode 0Ch) 60
6.4.3.3 Data Write (Opcode 0Bh) 60
6.4.3.4 Bit Write (Opcode 0A) 61
6.4.3.5 Monitor (Opcode 09h) 62
6.5 Frame Check Sequence 63
6.6 Key Parameter Table Locations 65
6.7 Error Response 66

7.0 More Information 67

1.0 How to Use this Addenda

The manual describes the protocols used by various Westinghouse programmable controllers. Once the protocol is known, a computer program can be written to transfer data to and from the programmable controller. The purpose of this manual is to assist in a programmer in writing just such a computer program.

The NLAM-B58 manual is now divided into two sections, the main section and the addenda. This addenda includes information on new protocols, modem wiring diagrams and new products.

For additional information refer to Section 8, page 67.

2.0 Summary of Westnet II Data Highway Commands

To address a drop on the Westnet II data Highway:

- Send the "Set Highway Address" to the Programmable Controller Interface (PCI).
- Once the PCI acknowledges this message, send the "Read Session Status" message. The response from this message will tell you if (1) a PLC exists at the address specified in the "Set Highway Address" message and (2) if another computer has opened a "session" with that PLC, thus preventing you from communicating with that PLC.
- You are now free to use conventional memory read and write (00 - 08 hex) opcodes. The PCI has, in effect, opened a virtual link with that addressed PLC.

2.1 Set Highway Address

```
Sent to PCI  09 00 HW DR SB CK
Response     09 00 HW DR SB CK
```

2.2 Determine Remote Address that PCI is addressed to

```
Sent to PCI  09 01 00 00 00 CK
Response     09 01 HW DR SB CK
```

2.3 Read Highway Status

```
Sent to PCI  09 02 00 00 00 CK
Response     09 02 EC 00 00 CK
```

2.4 Open Session

```
Sent to PCI  09 03 00 00 00 CK
Response     09 03 00 00 00 CK
```

2.5 Close Session

```
Sent to PCI  09 04 00 00 00 CK
Response     09 04 00 00 00 CK
```

2.6 Read Session Status

```
Sent to PCI  09 05 00 00 00 CK
Response     09 05 SS DR SB CK
```

2.7 Read Local PCI Address Setting

```
Sent to PCI 09 06 00 00 00 CK
Response    09 06 HW DR SB CK
```

HW - Highway number (0 is only currently supported value)
DR - Drop Number (1 - 254)
SB - Subdrop Number (0 or 1)
CK - Checksum of previous five bytes
SS - Session Status
- 80 - No drop in session with the PLC
- xx - Highway number of drop in session with this PLC

EC - Error Code
- 0 - No error found
- 1 - Message buffer full
- 2 - Invalid addressed drop (255 invalid, for example)
- 4 - Message size invalid (32 words is maximum allowed)
- 12 - PLC in session with another computer. You are locked out.

Description of Westnet Command Codes

Set Highway Address

This code is used to set up a transparent channel from one drop of the highway to another. Once the "Set Highway Address" command is acknowledged by the highway, conventional "6 Byte" commands may be sent to the programmable controller. Communications proceed as if the computer were connected directly to the programmable controller.

Read Remote Address

This function returns the address of the drop addressed by this PCI.

Read Highway Status

If a highway command is acknowledged with a 2 Byte error response, execute this command to determine the cause of the highway error.

Open Session

If the programmer wishes to lock out other stations during the time that the computer is communicating with the addressed drop, a "session" may be opened.

Close Session

A session must be closed with a drop to allow other locations the ability to communicate with that drop.

Read Session Status

This command returns the address of the location in session with the addressed drop.

Read Local Address

Returns the address of the drop your computer is plugged into.

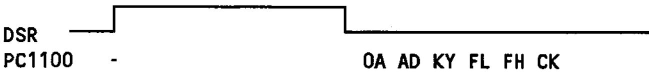
3.0 PC1000 Local Area Network Message Structure

To address a drop on the PC LAN network:

- Generate a "sync" signal at the slave by either raising Port B DSR, or sending a positive signal to the SYNC A line (as referenced to SYNC B). Refer to the timing diagram with the "Set Unit Address" for more information on this sync signal.
- Send a "Set Unit Address" message to the slave. The correctly addressed slave will answer, indicating a successful session has been opened with that site.
- Send Read Block and Write Block messages as necessary.

3.1 Set Unit Address

Computer - 0A AD 00 00 00 CK



Where:

AD	-	address of remote PLC desired
KY	-	status word returned with four least significant bits represent the PLC keyswitch position and fault status: Bit 0 - Stop/Program Mode Bit 1 - Run/Program Protect Mode Bit 2 - Run/Modify Mode Bit 3 - Fault Mode
FL	-	status word that represents the lower order byte of a 16 bit fault register in the PLC that assists in troubleshooting the PLC
FH	-	status word that represent the upper order byte of a 16 bit fault register in the PLC that assists in the troubleshooting the PLC
CK	-	checksum of previous 5 bytes

The Set Unit Address message, is used to request a particular slave on a multi-drop system to respond. All other stations drop off-line and remain off-line until another Set Unit Address message is transmitted over the network. It is important to note that a slave, once addressed, will remain connected to the network until another Set Unit Address message is heard. During this time, normal READ BLOCK and WRITE BLOCK messages are sent to that particular slave.

Notice that the DSR lead (pin 6 of the PC1100) must be raised high just before the message is received, but dropped low after the last byte has been transmitted.

3.2 Read Block

```
Computer - 05 LA HA NU 00 CK
PC1100   -                      05 LA HA NU 00 CK LD1 HD1 LD2 HD2..DK
```

Where:

LA - low byte of address in PLC to read
HA - high byte of address in PLC to read
NU - number of bytes to read minus 1 (i.e. 0 = 1 byte to read)
CK - checksum of previous 5 bytes
LDn - Low byte of word 'n' retrieved from PLC (number indicates which data word)
HDn - High byte of word 'n' retrieved from PLC (number indicates which data word)
DK - data checksum of all data bytes returned (does not include the first 6 bytes of command response

Notice that the DSR lead (pin 6 of the PC1100) must be raised high just before the message is received, but dropped low after the last byte has been transmitted.

3.3 Write Block

```
Computer - 07 LA HA NU 00 CK LD1 HD1 LD2..DK
PC1100   -                                 07 LA HA NU 00 CK
```

Where:

LA - low byte of address in PLC to write
HA - high byte of address in PLC to write
NU - number of bytes to write minus 1 (i.e. 0 = 1 byte to write)
CK - checksum of previous 5 bytes
LD - Low byte of word 'n' transmitted to PLC

DK - data checksum of all data bytes sent (does not include the first 6 bytes of command

Notice that the DSR lead (pin 6 of the PC1100) must be raised high just before the message is received, but dropped low after the last byte has been transmitted.

3.4 Erase Memory

```
Computer -     0F 01 00 00 00 10 C5 3A A3 5C EE 2A E2 AE 2A EE AE E2 C5 3A A3 5C
PC1200   -                                                                      0F 01 00 00 00 10
```

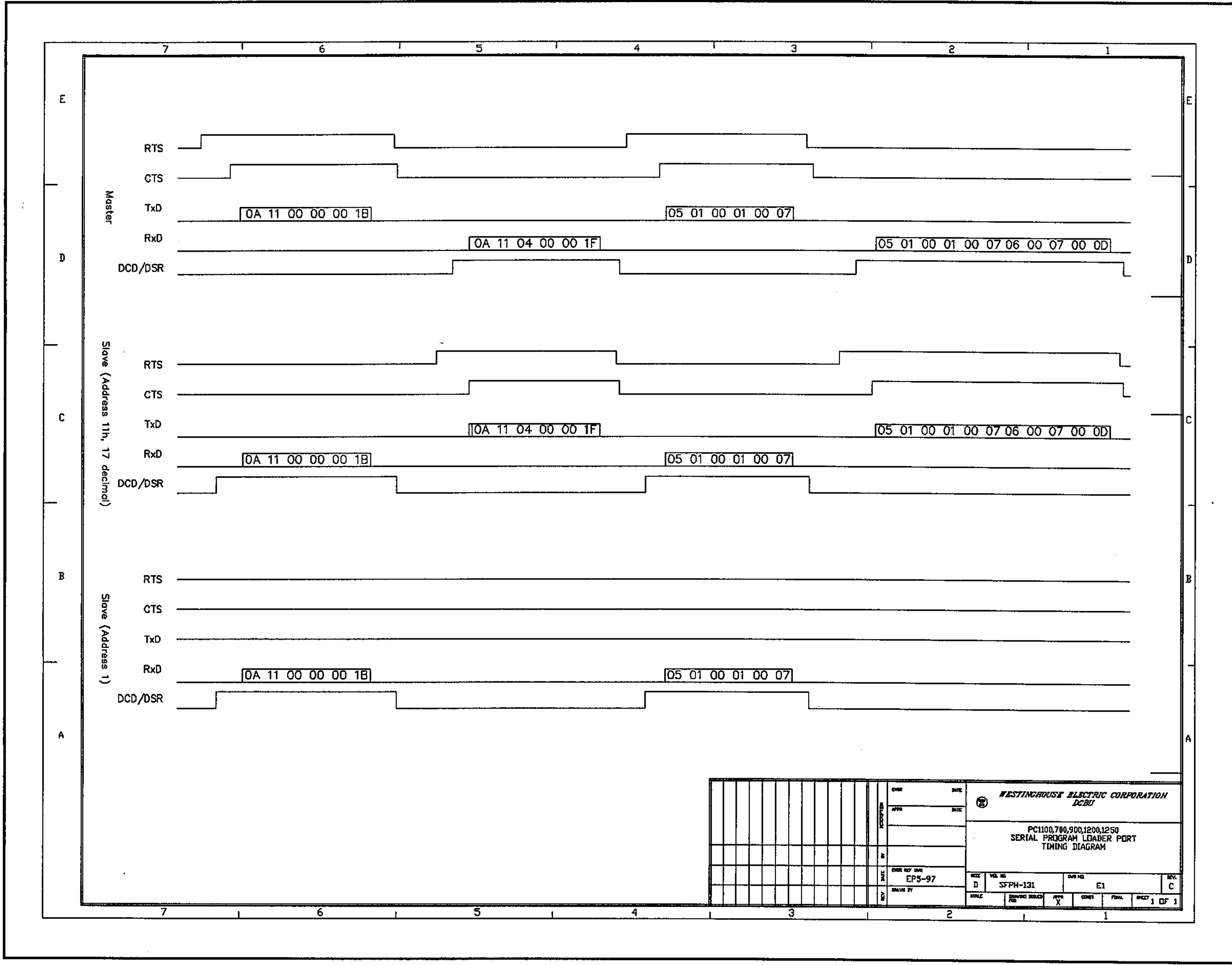
This message is only recognized by a PC1200 or PC1250. The keyswitch must be in stop mode (the key cannot be overridden to stop). All Ladder, Output Register and Holding Register memory is erased. The fault register is not erased, however executing a retest PC instruction immediately following this Erase Memory message will clear any old fault bits and cause the PLC to check for any new faults.

3.5 Retest PLC

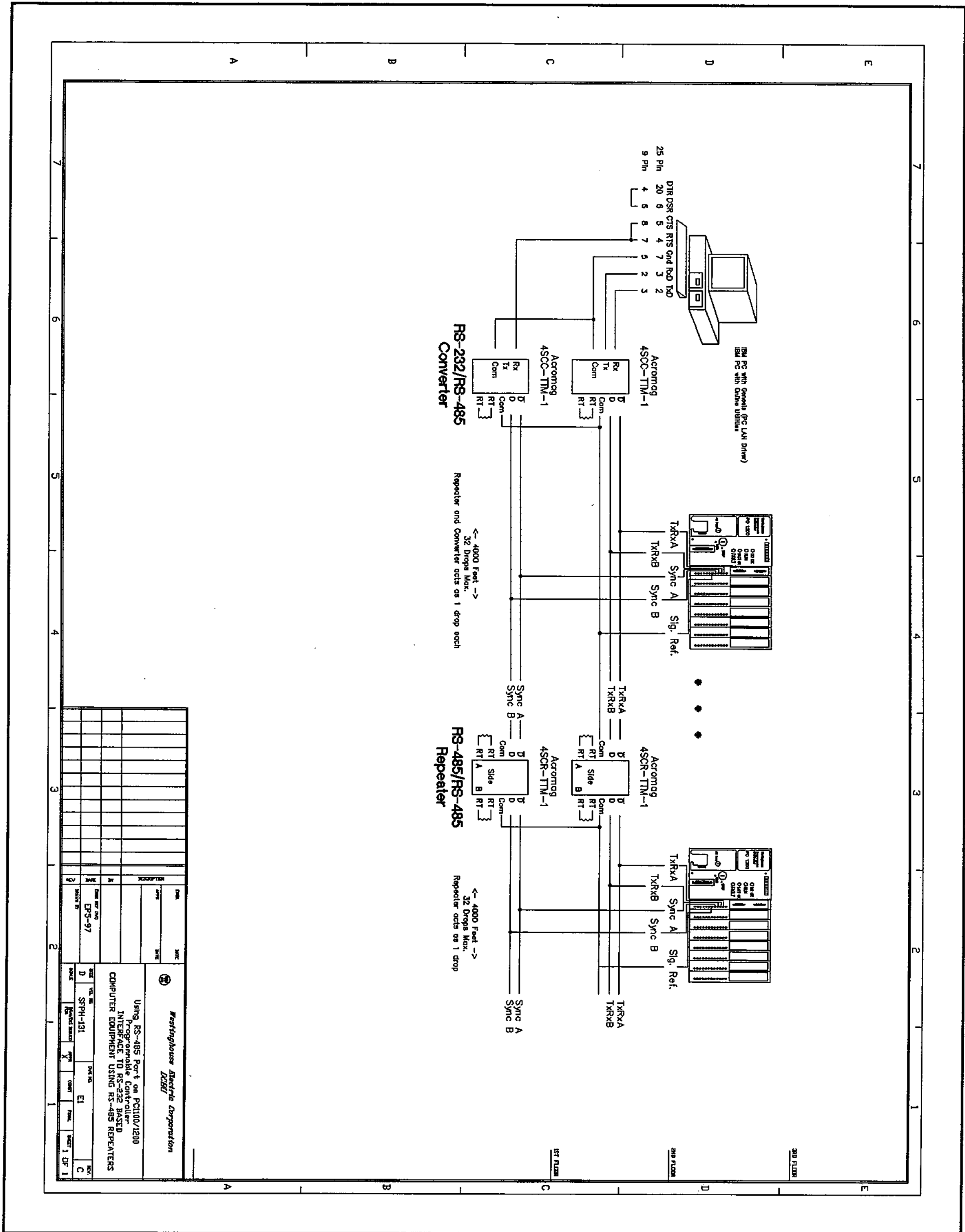
```
Computer -     03 05 82 11 00 9B
PLC      -                      03 05 82 11 00 9B
```

This message is simply a Bit Write to memory address 8205h. When the second bit of that word is set (and the PLC is in the stop mode), the PLC will clear the fault register and complete another set of diagnostics. If no faults are found the fault register remains clear. If a lingering fault is detected, the fault register is updated with this new fault data.

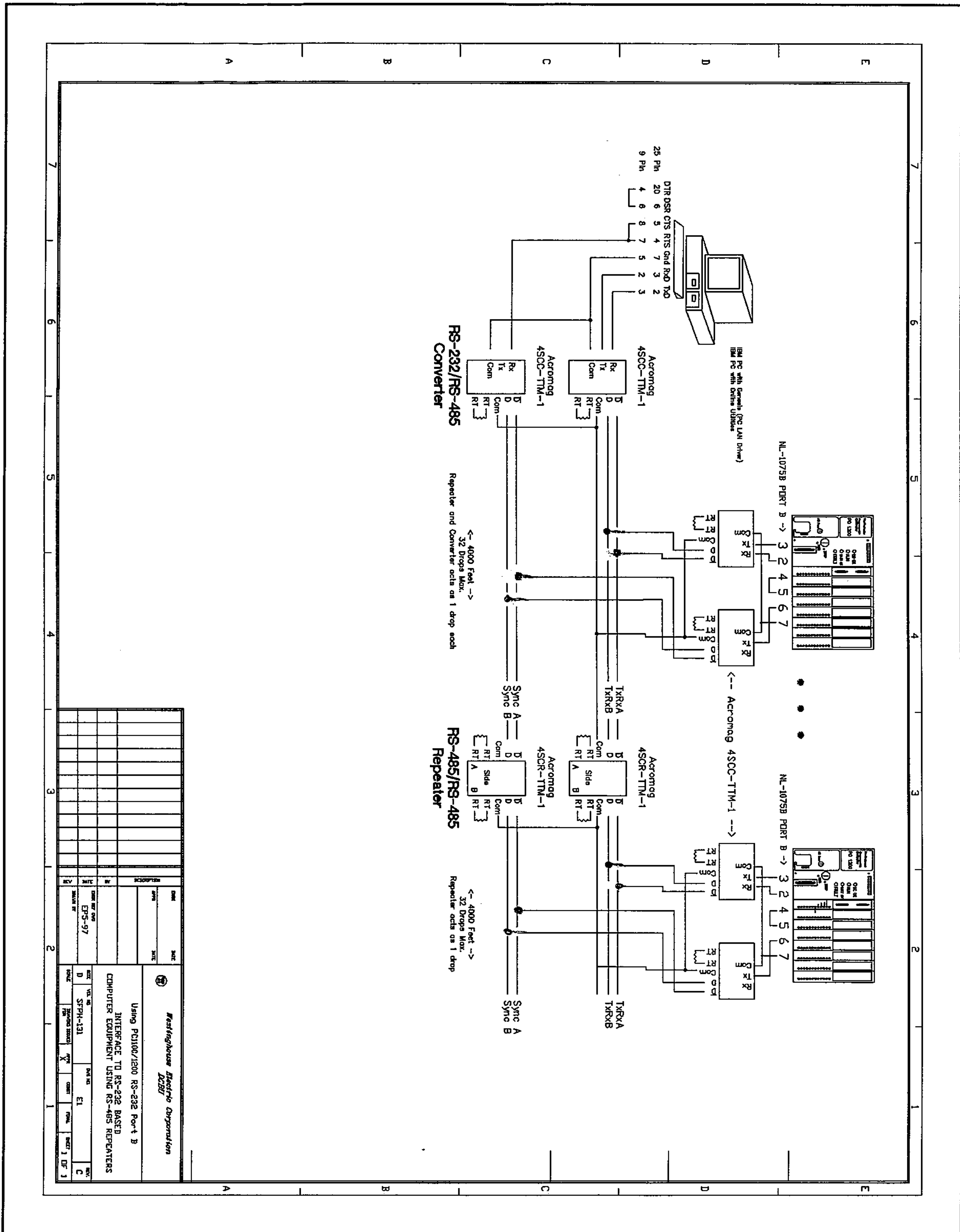
3.5 PC 1000 LAN Timing Diagram



3.6 PC 1000 LAN RS-485 Computer Wiring Diagram



3.7 PC 1000 LAN RS-232 Computer Wiring Diagram



4.0 Modbus™

Westinghouse programmable controllers offer the Modbus™ communications protocol built in to the entire family of PLCs.

4.1 Supported CPUs

Westinghouse PLC	Modbus Activated by:
PC503	NL-580 module with Modbus function block
PC1200	firmware (Configure Port ladder function)
PC1250	firmware (Configure Port ladder function)
PC2000	NL-2524 or NL-2525 module
PC1100	NCMZ-1799 module with SP1799-910 firmware
PC700	NCMZ-799 module with SP799-910 firmware
PC900	NCMZ-799 module with SP799-910 firmware
HPPC1500	NLFP-1520
HPPC1700	NLFP-1520

Westinghouse PLCs support each of these Modbus function codes:

01 - Read Output (coil) status
02 - Read Input status
03 - Read Output (and Holding) Registers
04 - Read Input Registers (not avail on PC503)
05 - Force Single coil
06 - Preset Single register
08 - Loop back diagnostic
16 - Preset Multiple registers

07 - Read exception status (PC1200, PC1250, NLFP-1520)
15 - Preset multiple coils (PC503, PC1200, PC1250, NLFP-1520)
17 - Report slave ID (PC1200, PC1250)
20 - Read extended memory (PC1200, PC1250)
21 - Write extended memory (PC1200, PC1250)

All Westinghouse PLCs support the Modbus RTU mode. Additionally, these units support Modbus ASCII mode:

HPPC1500
HPPC1700

Each function code is supported up to the maximum count value permitted by the Modbus specification for that function. Each Modbus slave supports exception codes 01, 02, 03, 04 and 06.

4.2 Message Types

The Modbus communications network was designed to provide a simple means of connecting slave controllers to a master controller for the purpose of distributed control and data acquisition. The simplicity of the protocol has made it a popular choice among many control and instrumentation vendors seeking a "common" language.

The Modbus protocol consists of a slave address, function code, data, and error checking. The master controller issues a command to which the appropriate slave responds. The following describes the message parameters:

4.2.1 Address

The address field contains the address of a specific slave (1-255) as assigned by the user. Each slave must be assigned a unique address. Only the addressed slave will respond to a message. The response message contains the address of the slave which issued it. If designing a system that includes non-Westinghouse Modbus devices, restrict the addresses to 1-247. Certain non-Westinghouse devices interpret addresses 248-255 in a different way.

A broadcast message uses a slave address of "0". All slaves on the network interpret the instruction and take action, but do not issue a response message. Only function codes 5, 6 and 16 can be used with this feature.

4.2.2 Function Code

This field defines the action to be performed. Implemented Functions Codes are 1, 2, 3, 4, 5, 6, 7, 8 and 16. The following table lists the function codes supported by the NCMZ-1799 module and how they are interpreted. Refer to the specific literature supplied with the device to determine the exact function codes supported.

01 Read Coil Status

Obtains the current status (On/Off) of a group of logic coils. Un-requested coils in the response message are zeroed.

02 Read Input Status

Obtains the current status (On/Off) of a group of discrete inputs.

03 Read Holding Registers

Obtains the current value in one or more Holding Registers. A maximum of 125 registers is allowed per request.

04 Read Input Registers

Obtains current value in one or more input registers.

05 Modify Coil Status

Sets logic coil to a state of On or Off. Note the change is made to the output status table and NOT THE FORCE TABLE in effect overriding the force table. Although this function will write to the coil regardless of the status of the force bit, the coil will be immediately overridden by the ladder program. If you wish to actually override the force state, you will need to locate the address of the force bit in the PLC memory (refer to the Westinghouse PLC Communications Manual).

06 Modify Register Contents

Write information to one register. The Module does not limit writes to below HRRU, therefore the user should use extreme caution when writing to any memory location other than Holding Registers.

07 Read Exception Status

NCMZ-1799 : The low order byte of HR6 is read. The contents of HR6 may be set by the Ladder Program.

PC1200: Coils CR1017-CR1024 are returned

HPPC : Not supported

08 Loop Back Diagnostic

Diagnostic test message which can be used to test communications. The response message is identical to received message. The test message must be the full length of a normal command (8 bytes)

16 Preset Multiple Registers

Information can be written to Multiple Holding Registers (up to 125 at a time) with one command.

4.2.3 Data

The data field contains information needed to perform the specific function. This may be values, addresses, or limits. For example, the function code Read Coils (01), requires data of starting coil number and quantity.

4.2.4 Error Check

This field contains the error checksum which is used to determine if a transmission has been received correctly. The Cyclical Redundancy Check (CRC) is used for the RTU format used in the 1.

4.2.5 Exception Response

Exception error codes are generated by the slave if the master requests an invalid point, quantity or function. In the response message, the most significant bit of the function code is set. The data field contains a one byte code, as described below:

code	**description**
01	Invalid function code
02	Invalid address
03	Invalid quantity requested
04	Communication error between module and host PLC

4.3 Message Structure

SA - Slave Address (valid 1 - 247)
AH - Address (high byte of reference number)
AL - Address (low byte of reference number)
BC - Byte count (number of bytes in data portion of message)
CH - Coil number (high byte of number)
CL - Coil number (low byte of number)
DH - Data Sent (High byte)
DL - Data Sent (Low byte)
EH - Error Check (High byte of CRC)
EH - Error Check (Low byte of CRC)
Dx - Data bytes sent or received
HH - High byte of number of Holding Registers
HL - Low byte of number of Holding Registers
IH - High byte of number of inputs
IL - Low byte of number of inputs
RH - High byte of number of Input Registers
RL - Low byte of number of Input Registers
ST - Coil Status (low byte of HR0006 is read with this exception status message)
VH - Value High (FF- ON, 00- OFF)
VL - Value Low (always set to 00)
xx - Any value (00-FF)

4.3.1 Read Coil (Output) Status (Function Code 1)

Sent-> SA 01 AH AL CH CL EH EL
Rvd <- SA 01 BC D1 D2... Dn EH EL

D1 - First byte of coil status's returned. Low order bit is first coil, etc.

4.3.2 Read Input Status (Function Code 2)

Sent-> SA 02 AH AL IH IL EH EL
Rvd <- SA 02 BC D1 D2... Dn EH EL

D1 - First byte of input status returned. Low order bit is first input, etc.

4.3.3 Read Holding Registers (Function Code 3)

Sent-> SA 03 AH AL HH HL EH EL
Rvd <- SA 03 BC D1 D2... Dn EH EL

D1 - High byte of first register returned
D2 - Low byte of first register returned
D3 - High byte of second register returned ....
Dn - Low byte of last register returned

4.3.4 Read Input Registers (Function Code 4)

Sent-> SA 04 AH AL RH RL EH EL
Rvd <- SA 04 BC D1 D2... Dn EH EL

- D1 - High byte of first register returned
- D2 - Low byte of first register returned
- D3 - High byte of second register returned ....
- Dn - Low byte of last register returned

4.3.5 Write Single Coil (Function Code 5)

Sent-> SA 05 AH AL VH VL EH EL
Rvd <- SA 04 AH AL VH VL EH EL

4.3.6 Write Single Holding Register (Function Code 6)

Sent-> SA 06 AH AL DH DL EH EL
Rvd <- SA 06 AH AL DH DL EH EL

4.3.7 Read Exception Status (Function Code 7)

Sent-> SA 07 EH EL
Rvd <- SA 07 ST EH EL

4.3.8 Loopback Test (Function Code 8)

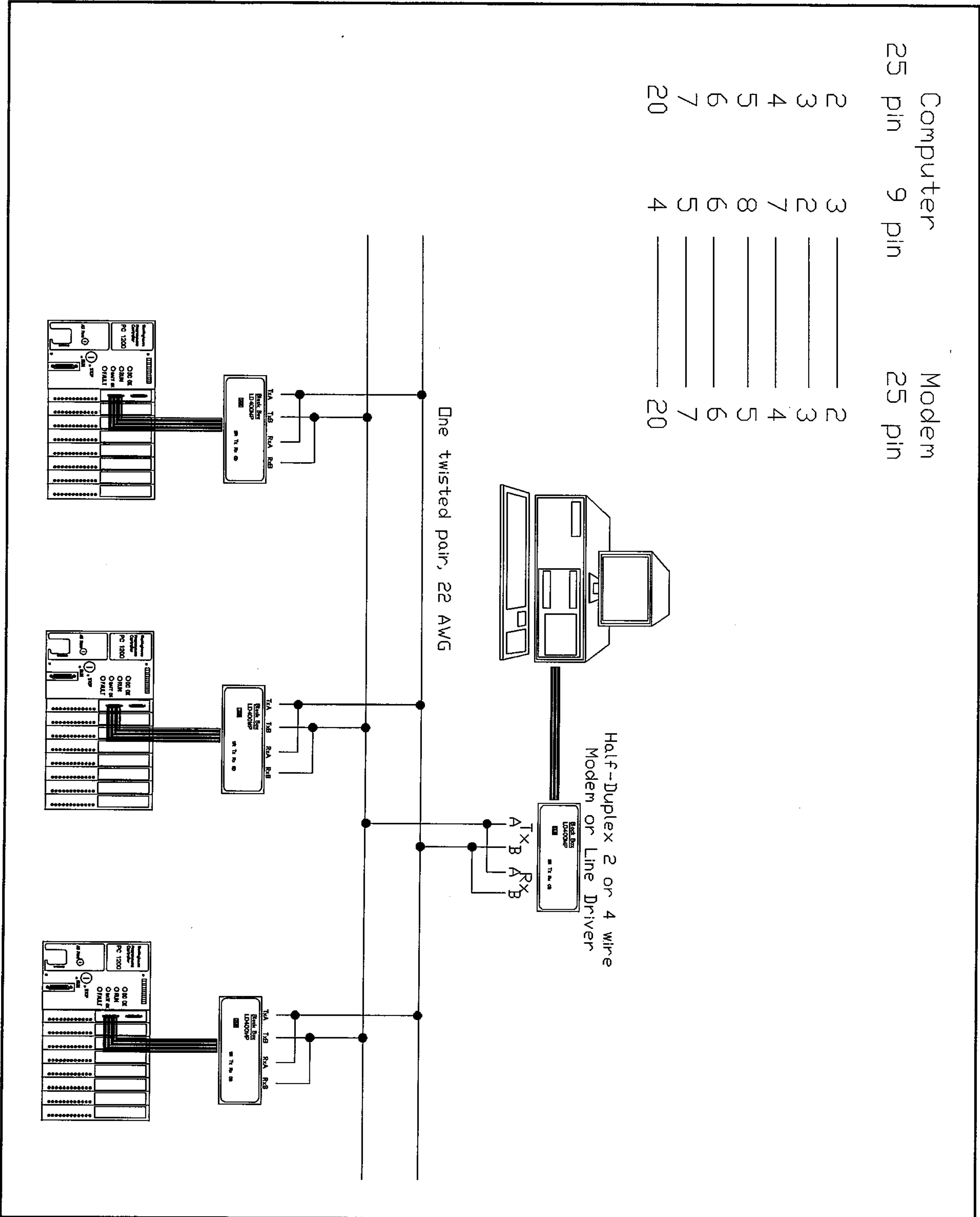
Sent-> SA 08 xx xx xx xx EH EL
Rvd <- SA 06 xx xx xx xx EH EL

4.3.9 Preset Multiple Registers (Function Code 16 decimal)

Sent-> SA 10 AH AL HH HL D1 D2 ... Dn EH EL
Rvd <- SA 10 AH AL HH HL EH EL

- D1 - High byte of first Holding Register written to remote
- D2 - Low byte of first Holding Register written to remote
- Dn - Low byte of last Holding Register written to remote

4.4 Modbus RS-232 Computer Wiring Diagram



4.5 Sample Modbus Master program in C

```
/*                    INIT.C
*
*  This is the main routine for the Modbus test program.
*  It displays the menu and calls the routines from MOD.C.
*
*/

    #include <\c\mod\mdos.h>
    #include <conio.h>
    #include <\c\mod\modbus.h>       /* uxi support routines */

    #define STOP '\032'    /* control z */

     int loopbk(char);
     int read_ex(char);

    void get_slave_id();
    void init_test();
    void port_config();
    int loopbak();
    int junk1(int);

/*********************************************************************************/

main ()     /* main routine */
{
        /* define variables */

    int x, y, i, n;         /* general purpose integers */
    char type, c;


   setmode(3);            /* set up display mode */
   clear();               /* clear the screen */
   home();                /* move the cursor to the home position */

   init_test();  /* initialize */

   menu();       /* display menu */

    /* Get the selection from the user. */

    while ( type != STOP )
    {
      printf("\n>>");       /* display prompt */
      type = getch();       /* read character */
      putch(type);          /* echo character to screen */

      switch (toupper(type)) {      /* execute code */

      case  '1':            /* read coil status */
             printf(" Read coil status \n");
             read_coils();
             break;

      case  '2':            /* read input status */
             printf(" Read input status \n");
             read_inputs();
             break;

      case  '3':            /* read holding registers */
             printf(" Read Holding Registers \n");
```

```
        read_registers();
        break;

case  '4':          /* read input registers */
        printf(" Read Input Registers \n");
        read_input_regs();
        break;

case  '5':          /* force single coil */
        printf(" Force Single Coil \n");
        force_coil();
        break;

case  '6':          /* preset single register */
        printf(" Preset Single Register \n");
        preset_reg();
        break;

case  '7':          /* read exception status */
        printf(" Read Exception Status \n");
        read_exception();
        break;

case  '8':          /* loopback test */
        printf(" Loopback Test \n");
        loopback();
        break;

case  'M':          /* display menu */
        menu();
        break;

case  'P':          /*  */
        printf(" Preset Multiple Registers \n");
        preset_regs();
        break;

case  'F':          /*  */
        printf(" Force Multiple Coils \n");
        force_coils();
        break;

case  'R':          /*  */
        printf(" Report Slave ID \n");
        report_slave();
        break;

case  'C':          /*  */
        printf(" Configure Port \n");
        port_config();
        break;

case  'H':          /*  */
        printf(" Help Screen \n");
        help();    /*  */
        break;

case  'S':          /*  */
        get_slave_id();
        break;

case  'Z':          /*  */
        printf(" General Read \n");
        gen_read();
```

```
            break;

      case  'A':          /*  */
            printf(" General Write \n");
            gen_write();
            break;


      case  'L':          /*  */
           /* Continuous loop test */
            printf(" Continuous loopbacks \n");
           while ( type != STOP )
           {
         loopbak();

         if (keyhit() == TRUE) type = getch();

           }

            type = 0;
            break;

      case  'Q':          /* quit */
            type = STOP;
            break;

       }    /* end of switch */

     } /* end while */

   }   /* end of main */

/*****************************************************************/

 menu()
 {

   printf("\n\n              MODBUS Test Routine \n");
   printf("\n 1  Read Output Status       F  Force Multiple Coils");
   printf("\n 2  Read Input Status        P  Preset Multiple Registers");
   printf("\n 3  Read Output Registers    R  Report Slave ID");
   printf("\n 4  Read Input Registers     M  Menu");
   printf("\n 5  Force Single Coil        C  Configure Port");
   printf("\n 6  Preset Single Register   H  Help");
   printf("\n 7  Read Exception Status    S  Change Slave Address");
   printf("\n Z  General Read             A  General Write   ");
   printf("\n 8  Loopback Test            Q  Quit Routine \n");

  }
/*****************************************************************/

 help()
 {

   printf("\n This program will allow an IBM PC to act as a Modbus master.");
   printf("\n A null modem cable should be connected the COM1 port of the ");
   printf("\n PC and the serial port of the Modbus slave.   ");
   printf("\n Default values are: 9600 baud, 1 stop, no parity, 8 data.");
   printf("\n Default slave address = 1. ");

  }
```

```
/*
*                          DOS.C
*     This module contains the dos handling routines.
*     These routines are used for screen manipulation and for
*     timekeeping.
*
*/

   #include <dos.h>
   #include <\c\mod\mdos.h>
   #define  TRUE      1
   #define  FALSE     0


  /**********************************************************************/

  /*   setmode.c -- set the video mode  */
  void setmode( m )
  unsigned char m;

   {
   union REGS rin, rout;

   rin.h.ah = 0;
   rin.h.al = m;
   int86(0x10, &rin, &rout);
   }

  /**********************************************************************/
  /*   returns the current page number
  */

  unsigned char getpage()

  {
   union REGS rin, rout;

   rin.h.ah = 15;
   int86(0x10, &rin, &rout);
   return rout.h.bh;
   }

  /**********************************************************************/

  /*   setborder.c -- set border color
  */

  void setborder(color)
   unsigned char color;

   {
   union REGS rin, rout;

   rin.h.ah = 11;
   rin.h.bh = 0;
   rin.h.bl = color;

   int86(0x10, &rin, &rout);
   }
  /**********************************************************************/

/*  moves cursor to indicated position */

 void setcurs(row,col,page)
```

```
unsigned char row, col, page;

  {
   union REGS rin, rout;

   rin.h.ah = 2;
   rin.h.dh = row;
   rin.h.dl = col;
   rin.h.bh = page;
   int86(0x10, &rin, &rout);
  }
  /*********************************************************************/

/*  get current cursor position */

 void getcurs(pr,pc,page)
 unsigned char *pr, *pc, page;

  {
   union REGS rin, rout;

   rin.h.ah = 3;
   rin.h.bh = page;
   int86(0x10, &rin, &rout);
   *pr = rout.h.dh;
   *pc = rout.h.dl;
  }

  /*********************************************************************/

 void home()

  {
   setcurs(0,0,getpage());
  }

  /*********************************************************************/

 void clear()

  {
   union REGS rin, rout;

   rin.h.ah = 6;
   rin.h.al = 0;
   rin.h.ch = 0;
   rin.h.cl = 0;
   rin.h.dh = 24;
   rin.h.dl = 79;
   rin.h.bh = 7;
   int86(0x10, &rin, &rout);
  }

  /*********************************************************************/

  /*  Checks to see if there is a character
      available at the keyboard.    */

 int keyhit()

  {
   union REGS rin, rout;

   rin.h.ah = 0x0b;
```

```
 intdos(&rin, &rout);

  if (rout.h.al == 0xff)
  return(TRUE);
  else return(FALSE);
 }
/*********************************************************************/

/*   get status -- returns state of COM1 port. */

int get_status()

{
 union REGS rin, rout;

 rin.h.ah = 3;    /* get port status */
 rin.x.dx = 0;    /* com 1 port */

 int86(0x14, &rin, &rout);

  return(rout.x.ax);
 }
/*********************************************************************/

/*   xmit_char -- sends character out com1. */

int xmit_char( unsigned char ch)

{
 union REGS rin, rout;

 rin.h.ah = 1;    /* send character */
 rin.x.dx = 0;    /* com 1 port */
 rin.h.al = ch;

 int86(0x14, &rin, &rout);

  return(rout.x.ax);

 }

/*********************************************************************/

/*   rec_char -- receives character from com1. */

int rec_char(unsigned char *ch)

{
 union REGS rin, rout;

 rin.h.ah = 2;    /* get character */
 rin.x.dx = 0;    /* com 1 port */

 int86(0x14, &rin, &rout);

 *ch = rout.h.al;

  if ((rout.h.ah & 0x80) == 0)
  return(TRUE);
  else return(FALSE);

 }
```

```
/**********************************************************************/

/*   init_port -- initializes com1 port. */

int init_port(unsigned char ch)

{
 union REGS rin, rout;

 rin.h.ah = 0;    /* initialize port */
 rin.x.dx = 0;    /* com 1 port */
 rin.h.al = ch;

 int86(0x14, &rin, &rout);

  return(rout.x.ax);  /* return status */
 }

/**********************************************************************/

/*   get_char -- receives character from com1. */

int get_char(unsigned char *ch)
{
 #define RS232_BASE_PTR (( int far *)( 0x0040L << 16) ) /* 0040:0000 */
 #define DTR_ON 0X01
 #define DSR_ON 0X20
 #define RX_PORT 0X03F8
 #define MODEM_CON 0X03FC
 #define MODEM_STAT 0X03FE
 #define LINE_STAT 0X03FD
 #define LINE_CON 0X03FB
 #define DLY 8000

 union REGS rin, rout;
 int count;
 int rs232_port;
 int far *base;


 /* DTR ON */
 outp(MODEM_CON,DTR_ON);  /* turn DTR on, RTS off */

 /* wait for DSR */

 count = 0;
 while ((inp(MODEM_STAT) & DSR_ON) == 0 && count < DLY) {count++;}

 if (count < DLY)  /* if no timeout */
 {
    /* wait for data ready flag */
    count = 0;
    while ((inp(LINE_STAT) & 1) == 0 && count < DLY) {count++;}

    if (count < DLY)  /* get character */
    {
     *ch = inp(RX_PORT);
     return(TRUE);
    }
    else
    {
     return(FALSE);
    }
```

```
   }
   else
   {
      return(FALSE);
   }
  }
  /**********************************************************************/


/*  returns the number of seconds
    that have elapsed since midnight.   */

 double get_secs()

  {
   double hrs, mins, secs, hundredths;
   union REGS rin, rout;

   rin.h.ah = 0x2c;

   intdos(&rin, &rout);

   hrs = (double) rout.h.ch;
   mins = (double) rout.h.cl;
   secs = (double) rout.h.dh;
   hundredths = (double) rout.h.dl;

   return (hrs * 3600.0 + mins * 60.0 + secs + hundredths / 100.0);
   }

  /**********************************************************************/

  /* return the elapse time between start and finish in seconds */


   double how_long (double start, double finish )
   {
   #define FULL_DAY 86400.0  /* seconds in a day */

   if (start > finish)
     return ( FULL_DAY - start + finish );
     else return ( finish - start );
   }
  /**********************************************************************/

 /*  delay (t) number of seconds */
 void delay (double t)
 {
  double start, current;
  int i;

   current = 0;
   start = get_secs();   /* get seconds since midnight */
   while ( current < t )
   {
      current = how_long( start, get_secs() );
   }
  }
```

```
/*                    MOD.C
*   This file contains the routines that send and receive
*   commands to the Modbus slave.
*
*/

    #include <conio.h>
    #include <\c\mod\mdos.h>
    #include <\c\mod\modbus.h>       /* support definations */

    struct byte  {
       unsigned char b0;
       unsigned char b1;} ;

    union xxx {
       unsigned int w;
       struct byte  b;
            };

    union xxx temp;

    byte config;

    int get_char(unsigned char *);
    void get_slave_id();
    void port_config();
    void init_test();
    int xmit_char(unsigned char);
    int rec_char(unsigned char *);
    int init_port(unsigned char);
    int get_status();
    void disp_buf(unsigned char *, int);
    int swap_bytes(int);
    int get_start();
    int get_number();
    int junk(int);
     int loopbk(char);
     int read_ex(char);
    byte get_byte();
    int get_word();

    void send_request(char *, int);
    int get_response();

    union mod_messages loc_buf;

    unsigned char slave_id;   /* slave modbus address */
    double rts_delay;         /* 3 character time */

/**********************************************************************/
void get_slave_id()
     {
     int n;

       n = 300;
       while (n > 255)
       {

       printf("\n  Enter the slave's address > ");
       scanf("%d",&n);
       if (n > 255 )
             printf("\n ERROR - value too large");
       }
       slave_id = n;
```

```
     }

void port_config()
     {
     int n;

       /* get baud rate */
       printf("\n  Configure COM1 port \n");

       rts_delay = 0.004;  /* 4 mS delay */
       printf("\n  Baud rates:  9600, 4800, 2400, 1200, 600, 300");
       printf("\n  Enter baud rate >> ");
       scanf("%d",&n);
       config = 0xE0; /* assume 9600 */
       if ( n == 4800 )
       {
          config = 0xC0;
          rts_delay = 0.008;
       }
       if ( n == 2400 )
       {
          config = 0xA0;
          rts_delay = 0.016;
       }
       if ( n == 1200 )
       {
          config = 0x80;
          rts_delay = 0.026;
       }
       if ( n == 600 )
       {
          config = 0x60;
          rts_delay = 0.050;
       }
       if ( n == 300 )
       {
          config = 0x40;
          rts_delay = 0.100;
       }

       printf("\n  Enter Stop bits (1 or 2) >> ");
       scanf("%d",&n);
       if ( n == 2 ) config = config | 0x04;

       printf("\n  Parity:  0 = none, 1 = odd, 2 = even");
       printf("\n  Enter Parity >> ");
       scanf("%d",&n);
       if ( n == 1 ) config = config | 0x08;
       if ( n == 2 ) config = config | 0x18;

       printf("\n  Enter Data bits (7 or 8) >> ");
       scanf("%d",&n);
       if ( n == 7 ) config = config | 0x02;
       else config = config | 0x03;

       init_port(config); /* initialize the port */
     }

void init_test()
     {
     int n;
      /* initialize variables */

      slave_id = 1;
```

```
     init_port(0XEF); /* initialize the port */
     /* 8 date, 1 stop, 9600 baud, no parity */
     config = 0xE3;   /* e3 */

    }

int get_start()
    {
    int n;

      printf("\n Enter starting address >");
      scanf("%d",&n);
      return(n);
    }

int get_value()
    {
    int n;

      printf("\n Enter register value >");
      scanf("%d",&n);
      return(n);
    }

int get_hr_val()
    {
    int n;

      printf("\n Enter holding register value >");
      scanf("%d",&n);
      n = swap_bytes(n);
      return(n);
    }

int get_number()
    {
    int n;

      printf("\n Enter total number >");
      scanf("%d",&n);
      return(n);
    }

int get_state()
    {
    int n;

      printf("\n Enter coil state (1 or 0) >");
      scanf("%d",&n);
      if ( n != 0 ) n = 0X00FF;
      return(n);
    }

byte get_coil_stat()
    {
    byte n;

      printf("\n Enter byte of coil data (HEX) >");
      scanf("%2x",&n);
      return(n);
    }

byte get_byte()
```

```
    {
    byte n;

      printf("\n Enter byte value (HEX) >");
      scanf("%2x",&n);
      return(n);
    }

int get_word()
    {
    int n;

      printf("\n Enter word value (HEX) >");
      scanf("%4x",&n);
      return(n);
    }

/***************************************************************/
/*
*       Read coils request
*/

int read_coils()

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = slave_id;             /* Modbus target drop */
    fpnt->fc = 1;                     /* transaction code */
    fpnt->start_adrs = swap_bytes(get_start());
    fpnt->number = swap_bytes(get_number());
    fpnt->crc = crc(loc_buf.c,6);     /* checksum */

    printf("\n Read Coils request");
    disp_buf(loc_buf.c,8);

    send_request(loc_buf.c,8); /* send message to slave */
    get_response();

    return(8);
}


/***************************************************************/
/*
*       Read inputs request
*/

int read_inputs()

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = slave_id;             /* Modbus target drop */
    fpnt->fc = 2;                     /* transaction code */
    fpnt->start_adrs = swap_bytes(get_start());
    fpnt->number = swap_bytes(get_number());
    fpnt->crc = crc(loc_buf.c,6);     /* checksum */
```

```
    printf("\n Read Inputs request");
    disp_buf(loc_buf.c,8);

    send_request(loc_buf.c,8); /* send message to slave */
    get_response();

    return(8);
}


/*************************************************************/
/*
*       Read output registers request
*/

int read_registers()

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = slave_id;           /* Modbus target drop */
    fpnt->fc = 3;                   /* transaction code */
    fpnt->start_adrs = swap_bytes(get_start());
    fpnt->number = swap_bytes(get_number());
    fpnt->crc = crc(loc_buf.c,6);     /* checksum */

    printf("\n Read Holding Registers request");
    disp_buf(loc_buf.c,8);

    send_request(loc_buf.c,8); /* send message to slave */
    get_response();

    return(8);
}

/*************************************************************/
/*
*       Loopback test
*/

int loopbk(char s)

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = s;            /* Modbus target drop */
    fpnt->fc = 8;                   /* transaction code */
    fpnt->start_adrs = 0;
    fpnt->number = 0x1234;
    fpnt->crc = crc(loc_buf.c,6);     /* checksum */

    disp_buf(loc_buf.c,8);

    send_request(loc_buf.c,8);        /* send message to slave */
    i = get_response();

    return(i);
}
```

```
/****************************************************************/
/*
*       Loopback test
*/

int loopback()

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = slave_id;             /* Modbus target drop */
    fpnt->fc = 8;                     /* transaction code */
    fpnt->start_adrs = 0;
    fpnt->number = 0x1234;
    fpnt->crc = crc(loc_buf.c,6);     /* checksum */

  /*    printf("\n Loopback Test");    /****/
    disp_buf(loc_buf.c,8);

    send_request(loc_buf.c,8);        /* send message to slave */
    i = get_response();

    return(i);
}

/****************************************************************/
/*
*       Loopback test
*/

int loopbak()

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = slave_id;             /* Modbus target drop */
    fpnt->fc = 8;                     /* transaction code */
    fpnt->start_adrs = 0;
    fpnt->number = 0x1234;
    fpnt->crc = crc(loc_buf.c,6);     /* checksum */


    send_request(loc_buf.c,8);        /* send message to slave */
    disp_buf(loc_buf.c,8);

    return(2);
}


/****************************************************************/
/*
*       junk1 test
*/

int junk1(int ccc)

{
int i;
struct read_ *fpnt;
```

```
    fpnt = &loc_buf.reads_;
    fpnt->sid = 222;            /* Modbus target drop */
    fpnt->fc = 8;              /* transaction code */
    fpnt->start_adrs = 2;
    fpnt->number = 0x1;
    fpnt->crc = 104;     /* checksum */

    send_request(loc_buf.c,ccc);        /* send message to slave */
    /*  disp_buf(loc_buf.c,ccc);   /****/

    return(8);
}

/**************************************************************/
/*
*       Read input registers request
*/

int read_input_regs()

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = slave_id;            /* Modbus target drop */
    fpnt->fc = 4;                    /* transaction code */
    fpnt->start_adrs = swap_bytes(get_start());
    fpnt->number = swap_bytes(get_number());
    fpnt->crc = crc(loc_buf.c,6);     /* checksum */

    printf("\n Read Input Registers request");
    disp_buf(loc_buf.c,8);

    send_request(loc_buf.c,8); /* send message to slave */
    get_response();

    return(8);
}


/**************************************************************/
/*
*       Force single coil request
*/

int force_coil()

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = slave_id;            /* Modbus target drop */
    fpnt->fc = 5;                    /* transaction code */
    fpnt->start_adrs = swap_bytes(get_start());
    fpnt->number = get_state();
    fpnt->crc = crc(loc_buf.c,6);     /* checksum */

    printf("\n Force single coil request");
    disp_buf(loc_buf.c,8);

    send_request(loc_buf.c,8); /* send message to slave */
```

```
    get_response();

    return(8);
}

/***************************************************************/
/*
*       Preset single register request
*/

int preset_reg()

{
int i;
struct read_ *fpnt;

    fpnt = &loc_buf.reads_;
    fpnt->sid = slave_id;            /* Modbus target drop */
    fpnt->fc = 6;                    /* transaction code */
    fpnt->start_adrs = swap_bytes(get_start());
    fpnt->number = swap_bytes(get_value());
    fpnt->crc = crc(loc_buf.c,6);    /* checksum */

    printf("\n Preset single register request");
    disp_buf(loc_buf.c,8);

    send_request(loc_buf.c,8); /* send message to slave */
    get_response();

    return(8);
}


/***************************************************************/
/*
*       Read exception status request
*/

int read_ex(char s)

{
int i;
struct read_ex *fpnt;

    fpnt = &loc_buf.read_except;
    fpnt->sid = s;              /* Modbus target drop */
    fpnt->fc = 7;                    /* transaction code */
    fpnt->crc = crc(loc_buf.c,2);    /* checksum */

    disp_buf(loc_buf.c,4);

    send_request(loc_buf.c,4); /* send message to slave */
    i = get_response();

    return(i);
}

/***************************************************************/
/*
*       Read exception status request
*/

int read_exception()
```

```
{
int i;
struct read_ex *fpnt;

    fpnt = &loc_buf.read_except;
    fpnt->sid = slave_id;           /* Modbus target drop */
    fpnt->fc = 7;                   /* transaction code */
    fpnt->crc = crc(loc_buf.c,2);   /* checksum */

    printf("\n Read exception status request");
    disp_buf(loc_buf.c,4);

    send_request(loc_buf.c,4); /* send message to slave */
    get_response();

    return(4);
}


/**************************************************************/
/*
*       Report slave ID request
*/

int report_slave()

{
int i;
struct read_ex *fpnt;

    fpnt = &loc_buf.read_except;
    fpnt->sid = slave_id;           /* Modbus target drop */
    fpnt->fc = 17;                   /* transaction code */
    fpnt->crc = crc(loc_buf.c,2);   /* checksum */

   /*   printf("\n Report Slave ID request");   /****/
    disp_buf(loc_buf.c,4);

    send_request(loc_buf.c,4); /* send message to slave */
    get_response();

    return(4);
}


/**************************************************************/
/*
*        Force multiple coils request
*/

int force_coils()

{
int i, n, x, len;
byte cnt;
struct force_cr *fpnt;

    fpnt = &loc_buf.force_crs;
    fpnt->sid = slave_id;           /* Modbus target drop */
    fpnt->fc = 15;                  /* transaction code */
    fpnt->start_adrs = swap_bytes(get_start());
    i = get_number();    /* get number of crs */
    fpnt->number = swap_bytes(i);
```

```
    n = i;
    n = n & 0x0007;    /* get remainder */
    x = i >> 3;        /* divide by 8 */
    if ( n != 0 ) x++; /* increment byte count if remainder */
    cnt = x;
    fpnt->byte_cnt = cnt;   /* byte count */
    len = x + 9;

    /* get coil statuses */
    for (i = 0; i < x; i++)
       fpnt->cr_stat[i] = get_coil_stat();

    temp.w = crc(loc_buf.c,len-2);     /* checksum */

    fpnt->cr_stat[i++] = temp.b.b0;
    fpnt->cr_stat[i] = temp.b.b1;

    printf("\n Force multiple coils request");
    disp_buf(loc_buf.c,len);

    send_request(loc_buf.c,len); /* send message to slave */
    get_response();

    return(len);
}

/***************************************************************/
/*
*       Preset multiple registers request
*/

int preset_regs()
{
int i, n, x, len;
byte cnt;
struct force_hr *fpnt;

    fpnt = &loc_buf.force_hrs;
    fpnt->sid = slave_id;              /* Modbus target drop */
    fpnt->fc = 16;                     /* transaction code */
    fpnt->start_adrs = swap_bytes(get_start());
    i = get_number();    /* get number of registers */
    fpnt->number = swap_bytes(i);
    cnt = i*2;
    fpnt->byte_cnt = cnt;   /* byte count */

    len = i*2 + 9;

    /* get register values */
    /* This is done in a strange way because of trouble
       with the word boundary */

    x = 0;
    for (n = 0; n < i; n++)
    {
      temp.w = get_hr_val();
      fpnt->hr_val[x++] = temp.b.b0;
      fpnt->hr_val[x++] = temp.b.b1;
    }

    temp.w = crc(loc_buf.c,len-2);     /* checksum */
    fpnt->hr_val[x++] = temp.b.b0;
    fpnt->hr_val[x] = temp.b.b1;
```

```
    printf("\n Preset multiple registers request");
    disp_buf(loc_buf.c,len);

    send_request(loc_buf.c,len); /* send message to slave */
    get_response();

    return(len);
}

/***************************************************************/
/*
*       General Write
*/

int gen_write()
{
int i, n, x, len;
byte cnt;
struct gen_wr *fpnt;

    fpnt = &loc_buf.general_wr;
    fpnt->sid = slave_id;              /* Modbus target drop */
    fpnt->fc = 21;                     /* transaction code */
    fpnt->ref_typ = 0x33;              /* reference type */
    printf("\n segment");
    fpnt->seg_adrs = swap_bytes(get_word());
    printf("\n offset");
    fpnt->off_adrs = swap_bytes(get_word());
    i = get_number();    /* get number of registers */
    fpnt->number = swap_bytes(i);
    cnt = i;
    fpnt->byte_cnt = cnt + 7;   /* byte count */

    len = i + 12;

    /* get register values */
    /* This is done in a strange way because of trouble
       with the word boundary */

    x = 0;
    for (n = 0; n < i; n++)
    {
      fpnt->bt_val[x++] = get_byte();
    }

    temp.w = crc(loc_buf.c,len-2);     /* checksum */
    fpnt->bt_val[x++] = temp.b.b0;
    fpnt->bt_val[x] = temp.b.b1;

    printf("\n General write request");
    disp_buf(loc_buf.c,len);

    send_request(loc_buf.c,len); /* send message to slave */
    get_response();

    return(len);
}

/***************************************************************/
/*
*       General Read
*/
```

```
int gen_read()
{
int i, n, x, len;
byte cnt;
struct gen_rd *fpnt;

    fpnt = &loc_buf.general_rd;
    fpnt->sid = slave_id;            /* Modbus target drop */
    fpnt->fc = 20;                   /* transaction code */
    fpnt->ref_typ = 0x33;            /* reference type */
    printf("\n segment");
    fpnt->seg_adrs = swap_bytes(get_word());
    printf("\n offset");
    fpnt->off_adrs = swap_bytes(get_word());
    i = get_number();    /* get number of registers */
    fpnt->number = swap_bytes(i);
    fpnt->byte_cnt = 7;    /* byte count */

    len = 12;
    fpnt->crc = crc(loc_buf.c,10);     /* checksum */

    printf("\n General read request");
    disp_buf(loc_buf.c,len);

    send_request(loc_buf.c,len); /* send message to slave */
    get_response();

    return(len);
}

/*****************************************************************/
/*
*       send request
*
*/

void send_request(char *from_ptr, int len)

 {
  int i;
  unsigned char c;


    /* clear the receive buffer */
    while ((get_status() & 0x0100) != 0) rec_char(&c);

    /* send the request */

    for (i = 0; i < len; i++) xmit_char(*from_ptr++);

    /* wait until the shift and transfer registers are empty. */
    while ((get_status() & 0x6000) != 0x6000) ;

 }

/*****************************************************************/
/*
*       get response
*
*/

int get_response()

{
```

```
int i, len, stat;
unsigned char *from_ptr;    /*  */
int timeout, counter, char_in;
int good_resp;

good_resp = TRUE;
from_ptr = &loc_buf.c[0];    /* byte pointer to local buffer */
char_in = FALSE;
len = 0;
counter = 0;

/* wait for first character to arrive */

while ( !char_in && (counter < 25) )
{
  counter++;
  /* see if a character is available */
  if ( get_char(from_ptr) )
  {
     char_in = TRUE;
     *from_ptr++;
     len++;
  }
}

/* if a character was received ... */
if (char_in == TRUE)
{
 /* get the rest of the characters */

 while ( get_char(from_ptr) )
  {
     *from_ptr++;
     len++;
  }

 /*  printf("\n Response from slave");   /*   */

 disp_buf(loc_buf.c,len);   /* display message in local buffer */

 if ( (loc_buf.c[1] & 0x80) != 0 ) good_resp = FALSE;

}

else
 {
 printf("\n ERROR - Communication Timeout"); /****/
 good_resp = FALSE;
 }

      return(good_resp);
}

/***************************************************************/
/*
*        Display Buffer
*  Display the buffer in hex.
*/

void disp_buf(unsigned char *pt, int len)
 {

 int i, n;
```

```
    n = 0;
    i = 0;
    printf("\n ");

    while ( i < len )
     {
      printf("%2x ", *pt);  /* print integer */
      pt++;
      i += 1;       /* add one to byte count */
      if (n >= 15)
       {  /* then start a new line */
          printf("\n ");
          n = 0;
       }
      else n++;

     } /* end while */
}
```

```
/*      Modbus CRC routine
*/

   #include <stdlib.h>
   #include <conio.h>


   struct byte  {
      unsigned char b0;
      unsigned char b1;} ;

   union xxx {
      unsigned int w;
      struct byte  b;
           };

 int swap_bytes(union xxx);
 unsigned int crc(unsigned char *, int);

int CRC_TABLE[256] = {0, -16191, -15999, 320, -15615, 960,  640, -15807,
    -14847,    1728,    1920,  -14527,    1280,  -14911,  -15231,    1088,
    -13311,    3264,    3456,  -12991,    3840,  -12351,  -12671,    3648,
      2560,  -13631,  -13439,    2880,  -14079,    2496,    2176,  -14271,
    -10239,    6336,    6528,   -9919,    6912,   -9279,   -9599,    6720,
      7680,   -8511,   -8319,    8000,   -8959,    7616,    7296,   -9151,
      5120,  -11071,  -10879,    5440,  -10495,    6080,    5760,  -10687,
    -11775,    4800,    4992,  -11455,    4352,  -11839,  -12159,    4160,
     -4095,   12480,   12672,   -3775,   13056,   -3135,   -3455,   12864,
     13824,   -2367,   -2175,   14144,   -2815,   13760,   13440,   -3007,
     15360,    -831,    -639,   15680,    -255,   16320,   16000,    -447,
     -1535,   15040,   15232,   -1215,   14592,   -1599,   -1919,   14400,
     10240,   -5951,   -5759,   10560,   -5375,   11200,   10880,   -5567,
     -4607,   11968,   12160,   -4287,   11520,   -4671,   -4991,   11328,
     -7167,    9408,    9600,   -6847,    9984,   -6207,   -6527,    9792,
      8704,   -7487,   -7295,    9024,   -7935,    8640,    8320,   -8127,
    -24575,   24768,   24960,  -24255,   25344,  -23615,  -23935,   25152,
     26112,  -22847,  -22655,   26432,  -23295,   26048,   25728,  -23487,
     27648,  -21311,  -21119,   27968,  -20735,   28608,   28288,  -20927,
    -22015,   27328,   27520,  -21695,   26880,  -22079,  -22399,   26688,
     30720,  -18239,  -18047,   31040,  -17663,   31680,   31360,  -17855,
    -16895,   32448,   32640,  -16575,   32000,  -16959,  -17279,   31808,
    -19455,   29888,   30080,  -19135,   30464,  -18495,  -18815,   30272,
     29184,  -19775,  -19583,   29504,  -20223,   29120,   28800,  -20415,
     20480,  -28479,  -28287,   20800,  -27903,   21440,   21120,  -28095,
    -27135,   22208,   22400,  -26815,   21760,  -27199,  -27519,   21568,
    -25599,   23744,   23936,  -25279,   24320,  -24639,  -24959,   24128,
     23040,  -25919,  -25727,   23360,  -26367,   22976,   22656,  -26559,
    -30719,   18624,   18816,  -30399,   19200,  -29759,  -30079,   19008,
     19968,  -28991,  -28799,   20288,  -29439,   19904,   19584,  -29631,
     17408,  -31551,  -31359,   17728,  -30975,   18368,   18048,  -31167,
   -32255, 17088,  17280, -31935, 16640,  -32319, -32639,  16448 };


/*********************************************************/

unsigned int crc(unsigned char *data_ptr, int len)

  {
   int i;
/*    unsigned char t1;*/
   union xxx xsum, temp;


         xsum.w = 0XFFFF;
```

```
         for (i = 0; i < len; i++, data_ptr++)
            {

        temp.b.b0 = *data_ptr;
        temp.b.b0 = temp.b.b0 ^ xsum.b.b0;
        temp.b.b1 = 0;
        xsum.b.b0 = xsum.b.b1;
        xsum.b.b1 = 0;

        xsum.w = xsum.w ^ CRC_TABLE[temp.w];

            }

    return(xsum.w);

   }           /* end crc */

/*********************************************************************/

int swap_bytes(union xxx n)

  {
    union xxx temp;

    temp.b.b0 = n.b.b1;
    temp.b.b1 = n.b.b0;

  return(temp.w);

  }
```

```
/******************************************************************************
*   mdos.h - definitions/declarations for dos routines.
*
*   Purpose:
*   This is used to declare the routines defined in dos.c.
*
******************************************************************************/


    void home();
    void clear();
    void setmode(unsigned char);
    void setborder(unsigned char);
    unsigned char getpage();
    int  keyhit();
    int get_char(unsigned char *);
    int init_port(unsigned char);
    int rec_char(unsigned char *);
    int get_status();
    int xmit_char(unsigned char);
```

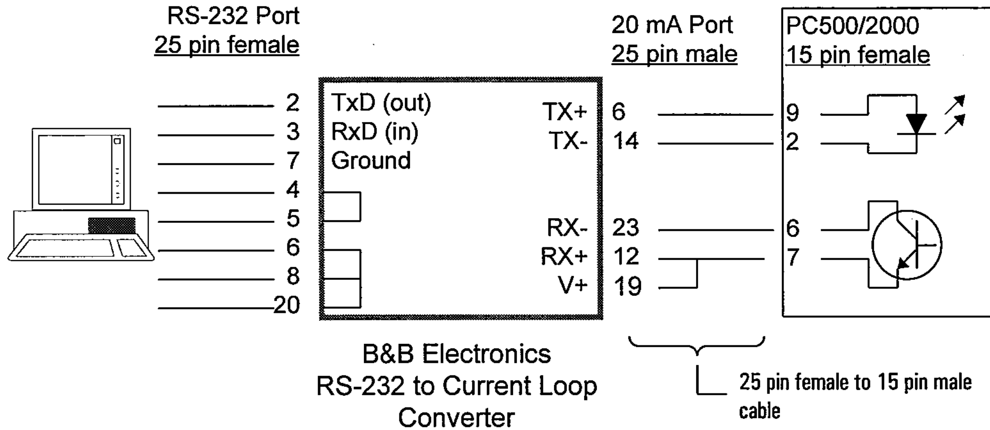
```
/******************************************************************************
*   modbus.h - definitions/declarations for modbus test routines
*
*   Purpose:
*   This file has declarations of modbus test routines and defines
*   the structure of the commands.
*
******************************************************************************/

/* define the data types used by routines */

typedef unsigned char byte;


    #define OK       1      /* flag */
    #define TRUE     1      /* true flag */
    #define FALSE    0      /* false flag */
    #define BUF_LEN  400    /* length in bytes of local buffer */

/* define structures for modbus commands */

struct read_ {          /* structure for read requests */
    byte sid;           /* slave ID */
    byte fc;            /* function code */
    int start_adrs;     /* starting address */
    int number;         /* number to read */
    int crc;            /* CRC */
    };

struct read_ex {        /* read exception status */
    byte sid;           /* slave id */
    byte fc;            /* function code */
    int crc;            /* checksum */
    };

struct force_cr  {      /* force multiple coils */
    byte sid;           /* slave id */
    byte fc;            /* function code */
    int start_adrs;     /* starting cr */
    int number;         /* number to force */
    byte byte_cnt;      /* byte count */
    byte cr_stat[100];  /* coil statuses */
    };

struct gen_rd  {        /* force multiple registers */
    byte sid;           /* slave address */
    byte fc;            /* function code */
    byte byte_cnt;      /* byte count */
    byte ref_typ;       /* byte count */
    int seg_adrs;       /* starting hr */
    int off_adrs;       /* starting hr */
    int number;         /* number to write */
    int crc;            /* checksum */
    };

struct gen_wr  {        /* force multiple registers */
    byte sid;           /* slave address */
    byte fc;            /* function code */
    byte byte_cnt;      /* byte count */
    byte ref_typ;       /* byte count */
    int seg_adrs;       /* starting hr */
    int off_adrs;       /* starting hr */
    int number;         /* number to write */
    byte bt_val[100];   /* register values */
```

```
    };

struct force_hr  {      /* force multiple registers */
    byte sid;           /* slave address */
    byte fc;            /* function code */
    int start_adrs;     /* starting hr */
    int number;         /* number to write */
    byte byte_cnt;      /* byte count */
    byte hr_val[200];   /* register values */
    };

struct report_s_id  {   /* report slave id response */
    byte sid;           /* slave id */
    byte fc;            /* function code */
    byte byte_cnt;      /* byte count */
    byte slv_id;        /* slave id */
    byte run_lit;       /* run light */
    byte mode_reg;      /* mode register */
    byte version;       /* version number */
    int err_reg;        /* error register */
    int mode_chg;       /* mode change */
    int crc;            /* checksum */
    };

    union mod_messages {
       int    i[200];
       unsigned char  c[400];
       struct   read_     reads_;
       struct   gen_wr    general_wr;
       struct   gen_rd    general_rd;
       struct   read_ex   read_except;
       struct   force_cr  force_crs;
       struct   force_hr  force_hrs;
       struct   report_s_id report_id;
       };


    int loopback();
    int preset_regs();
    int report_slave();
    int force_coils();
    int preset_reg();
    int read_exception();
    int force_coil();
    int read_coils();
    int read_inputs();
    int read_registers();
    int read_input_regs();
    int gen_read();
    int gen_write();
```

PC500 / PC2000 Loader Port Protocol

Physical Interface

Female 15 pin D shell connector
Four wire interface, 20 mA current loop. Transmitter: passive. Receiver: passive.



Data Format

Bit Rate:	9600
Data Bits:	8
Parity:	Even
Stop Bits:	1

Memory Types

Process Input Image	Ixxx.y, IBxxx, IWxxx	Scanned data from discrete and analog inputs
Process Output Image	Qxxx.y, QBxxx, QWxxx	Data written from PLC table to discrete and analog outputs
Flag Memory	Fxxx.y, FYxxx, FWxxx	Internal "logic coil data"
Timers	Tzzz	Pointers to timer memory
Counter	Czzz	Pointers to counter memory
System Memory	RSwww	Internal status words

Blocks

- Organization	OBaaa	Blocks of program code that are scanned under PLC operating system control
- Program	PBaaa	Blocks of program code that is scanned under user program control
- Function	FBaaa	User written special functions
- Data	DBaaa	Dynamically allocated memory for storage of recipes, messages, etc.

www	Word Number (0-255 typical)
xxx	Byte Number (0-127 typical for I/O, 0-255 for internal memory)
y	Bit Number (0-7)
zzz	Timer or Counter number (0-32 or 0-127)
aaa	Block Number (1-255 typical)

Rules

1. The Siemens loader protocol has little inherent error checking. There is no CRC, LRC or other checksum on any message. The only error checking is a parity check on individual bytes. (The lack of error checking is likely a reason that Siemens does not wish to use this protocol for inter-device communications.)

2. As of the date of this document, no method has been found for writing individual bits. This can cause a problem when trying to set or reset bits within the PLC. As a result, a byte must be allocated for each controlled bit -- with the resulting inefficiency.

3. The protocol uses the Data Link Escape (DLE). As with the Allen-Bradley PLC2 protocol, it may be required to "stuff" two DLE (hex 10) characters whenever one appears embedded in a message. As of the date of this document, the Write Memory command has not been extensively tested and as a result, we have little experience with this potential problem. It may be necessary to stuff an extra 10 hex character if that character appears in the data or address portion.

4. When a computer requests a block of data from the PLC, the PLC does not seem to stuff extra DLE characters when it transmits this block of data back to the computer. This would seem to indicate that the protocol does not require that an extra DLE character be stuffed whenever one appears in the data stream (as the AB PLC2 protocol requires). This would make sense since the number of bytes expected from the PLC for a read memory request is not ambiguous. The computer requests a particular block size to be read, therefore the number of bytes to be returned is known. As a result a definite end-of-message character is not required. Since no end-of-message character is needed to delimit the final byte of the message, "spurious" 10_{16} (DLE) characters appearing in the data stream would not be expected to cause problems.

5. The PLC interrupts the program scan to respond to a serial report request. Within the resolution limits of the protocol analyzer, the latency (time for PLC to respond) was measured at less than 500 μs. No testing was done to see if the data returned was from mid scan or if it was last updated at the end of the previous scan.

6. It was heard during this time that the Siemens protocol included different opcodes for reading memory at end of scan and immediately (mid-scan). No testing was done to see whether the Read Memory opcode discovered (04h) performed a mid-scan read, or read the value from the last scan.

7. Siemens addresses memory on byte boundaries. The PC500 (S5-10xU) and the PC2000 (S5-115U) have instruction that can manipulate 16 bit "words", but the bit number is different from a PLC that addresses memory on 16 bit word boundaries.

8. Cross-reference between Siemens and Westinghouse:

Siemens	Westinghouse
AG-90U	PC50
AG-95U	PC55
S5-100U	PC500
S5-102U	PC502
S5-103U	PC503
S5-115U	PC2000

9. While we have identified two additional opcodes that are not documented (08 -- Upload Program to PLC; 1A -- Retrieve Starting Program Address), many additional opcodes would seem to waiting to be discovered (00-02, 05-07, 09-17, 19, 1C-??).

Write Memory (03h)		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	From Computer	02			**03**		10	06				10	06	aa	bb	cc	dd	x1	x2	xN	10	04				10
	From PLC		10	06		02			16	10	03												10	06	02	
		26	27	28	29	30	31																			
	From Computer	06				10	06																			
	From PLC		12	10	03																					
		aabb: starting address in PLC ccdd: ending address in PLC x1: first byte of data to write x2: second byte of data to write xN: Nth byte to write																								

Read Memory (04h)		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	From Computer	02			**04**		10	06				10	06	aa	bb	cc	dd	10	04				10	06		
	From PLC		10	06		02			16	10	03									10	06	02				
		26	27	28	29	30	31	32	33	34	35	36	37	38	39	40										
	From Computer						10	06		10	06				10	06										
	From PLC	d1	d2	d3	d4	dN			02			12	10	03												
		aabb:starting address of memory block ccdd:ending address of memory block d1:value found at aabb, d2:value found at aabb+1, dN:value at ccdd																								

Read Pointer Table (18h)		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	From Computer	02			**18**		10	06				10	06	10	04				10	06						
	From PLC		10	06		02			16	10	03					10	06	02			00	00	00	00	00	I1
		26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	67	68	69	70	71	72	73	74	75	76
	From Computer																10	06		10	06				10	06
	From PLC	I2	Q1	Q2	F1	F2	T1	T2	C1	C2	R1	R2	X1	X2	P1	P2			02			12	10	03		
		I1I2: PII, Q1Q2: PIQ , F1F2: Flag memory, T1T2: Timer, C1C2: Counters, R1R2: System memory, X1X2: reserved, P1P2: Program end address																								

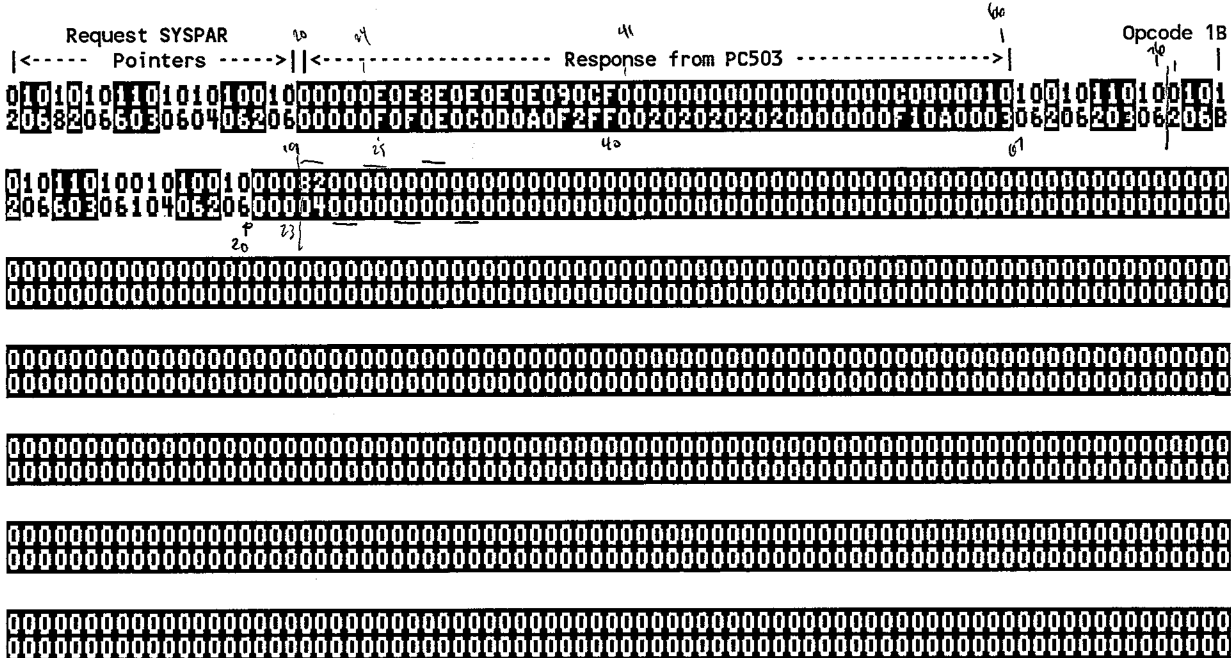
Read DB Pointers (1Bh)		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	From Computer	02			**1B**		10	06				10	06	01	10	04				10	06					
	From PLC		10	06		02			16	10	03						10	06	02			00	00	00	D1	D2
		26	27	28	29	30	...	534	535	536	537	538	539	540	541	542	543	544	545							
	From Computer									10	06		10	06				10	06							
	From PLC	D2	D3	D4	D5	D6	...	10	03			02			12	10	03									
		D1D2: address of DB001, D2D3: address of DB002																								

The PC500/2000 uses a "poll-respond" type of protocol that uses a method of retrieving data that is called "session building". What this means is that to read data from or write data to, the PC500/2000, a session must first be established. Once the PC500/2000 understands what you are requesting, you then issue a generic "read byte", "write byte" or "read system memory". It is important to realize that this final "read" or "write" message does not contain information telling you what address is to be read from or written to. That information is handled by an earlier message, dubbed "session creator".

5.1 Request SYSPAR

The block of data shown below was transmitted to the PC503. Among other things, the purpose of the transaction was to request that the PC503 return the starting addresses to each of the "SYSPAR" memory locations. Refer to your *PC-500 Systems Manual*, page 9-15 for more information on this function. For the PC503, those SYSPAR's are:

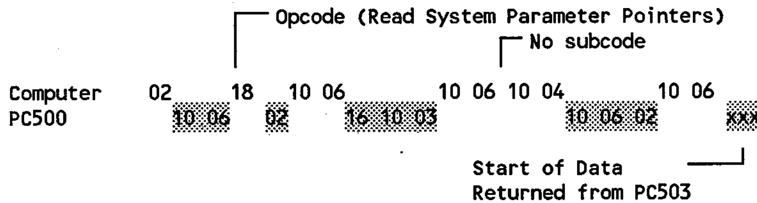
Process Input Image (PII)	EF00
Process Output Image (PIQ)	EF80
Flag Memory Area	EE00
Timer Memory Area	EC00
Counter Memory Area	ED00
System Data Area in PROM	EA00
PC Software Release	2804
Program End Address	CFFF



- Opcode 18:Reading Pointers to IB, QB, FB, DB, etc. (Protocol Analyzer Display) Opcode 1B: Read DB Pointers

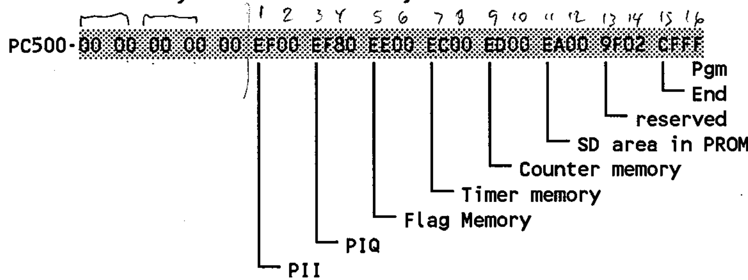
The inverse video is a response from the PC503 while normal video represents data sent from the computer (or smart device).

Note the contents of the first line of the request from the computer:



Each hexadecimal character (with the exception of the data and some control bytes) is actually an ASCII control character. If we replace the hexadecimal value with the equivalent ASCII name, the display would look something like this:

The PC503 responds to the above "give and take" by returning a block of data bytes that contains the data that opcode 18 asked for, namely *send the starting address of the PII, PIQ, flag word, timer, counter and system data memory.* Here is what the data break down as:

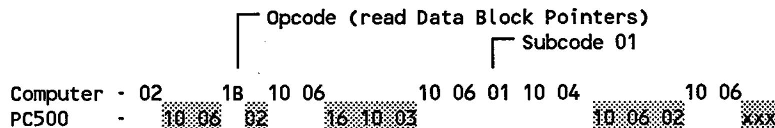


The bytes that represent the starting addresses have been merged together for clarity. Following this response from the PC500, an "epilogue" is sent to the PC500. That sequence is:

Notice that the last byte sent from the computer is not acknowledged by the PC503. This assures the computer that the session has been closed with the PC500 and that a new session could now be opened.

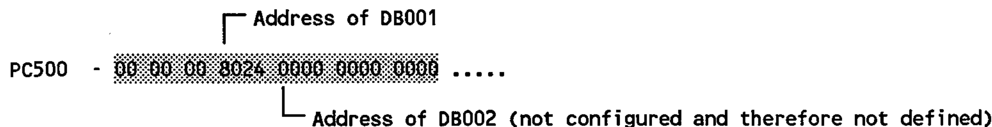
5.2 Request Data Block Pointers

The next session is indeed opened next. It is a request to read the "Data Block Pointer Addresses". It uses the 1B opcode, but this time a "subcode" is added to the message. The message looks like this:



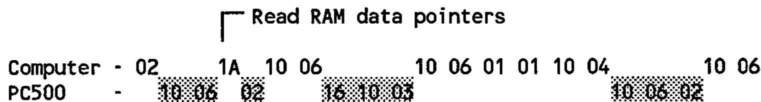
As before the "xxx" represents the data block sent back from the PC500 that contains the real information. The bytes returned up to this point only create the session.

In this particular PC503, only one Data Block (DB001) was configured in the PC503 memory. DB000 is always created by the system. This block can be read, but shouldn't be written to. DB001 through DB0255 (for the PC503) are available to written to, or read from. To find the location of this data block, examine the data returned:

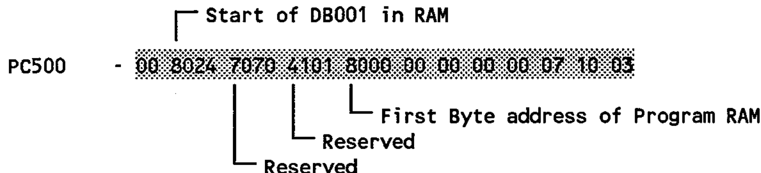


5.4 Request User Memory Starting Addresses

Another opcode that may be used is 1A. This opcode also returns the address of the first data block in memory.

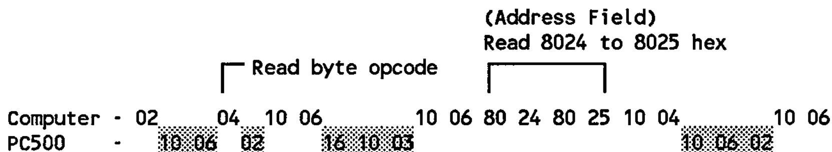


Following this session building, the PC503 responds with:



5.5 Read Byte (Immediate)

Now that we know where the data is located in memory, we will introduce a new opcode, 04, also known as "Read Byte Immediate[1]". We know from opcode 1B, "Read DB Pointers" that DB001 is located at memory address 8024. The PC500/2000 stores pointers to the start of memory, so we must read the value stored in 8024 to find the first byte stored in DB001. Incidentally, the value we stored in DB001:DW000 was 1101h.



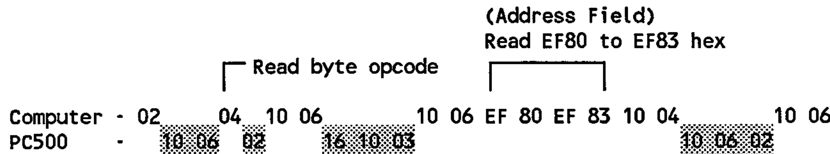
The PC503 responds to this session with (notice 1101h returned in the message):

After the response, the standard epilogue follows.

---

[1] This opcode will read the status of the bytes requested as soon as the command is processed by the PLC. This opcode will *not* wait until the end of scan. For this reason, special care must be taken in order to properly synchronize with the PLC program.

Other memory addresses can be read using the same 04 opcode. The only difference is to substitute a different address into the Address Field. For example, to read QB000 through QB003, a similar message would look something like this:



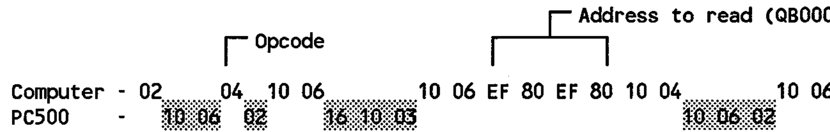

```
00000000000000000000000000000000000000000000000001010010110100101010110100010100
0000000000000000000000000000000000000000000000000030620620306206A206603061104062

10082774080000010100101101001000101101 0E8E810100100000011010010110100100010110 10
06004001100000703062062030620642066030 6F0F0040620600000203062062030620642066030 6

E8E81010010000003101001011010010001011010E8E81010010000005101001011010010001011 0
F1F1040620600000403062062030620642066030 6F2F2040620600000603062062030620642066 03

10E8E8101001000000710100101101001000101101 0E8E810100100000011010010110100100010 1
06F3F3040620600000803062062030620642066030 6F0F004062060000020306206203062064206 6
```

- Read QB000, QB001, QB002 and QB003

The protocol analyzer display shows how each byte (QB000, QB001, QB002 and QB003) was read using an individual byte read.

Notice that the message is similar to reading a Data Block, except that the address to read is now changed to EF80, EF81, EF82 and EF83 corresponding to QB000, QB001, QB002 and QB003 respectively. These addresses were found from the 1B opcode.

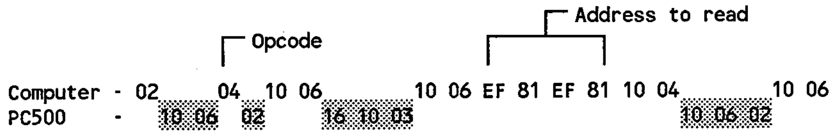
The following display shows the breakdown of the message required to read the first 8 output bits (QB000).



The PC503 responds with the data requested:

The value found at QB000 was 12h (0001 0010 binary).

By changing the Address value, different QB's can be read. The following message reads QB001 instead of QB000.



The PC503 responds with the data requested:

Any address in the PC500/2000 can be read in this manner. Simply substitute the address (as found using the 1B opcode) in the address field. Remember, the following key addresses (in the PC503):

PII digital	EF00 to EF1F
PII analog	EF40 to EF7F
PIQ digital	EF80 to EF9F
PIQ analog	EFC0 to EFFF
Timers	EC00 to ECFF
Retentive Counters	ED00 to ED0F
Non-retentive Counters	ED10 to ED3F
Retentive Flags	EE00 to EE3F
Non-retentive Flags	EE40 to EE7F
OB address list	DC00 to DDFF
FB address list	DE00 to DFFF
PB address list	E000 to E1FF
SB address list	E200 to E3FF
DB address list	E400 to E5FF
System Data	EA00 to EBFF

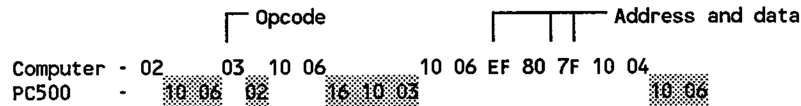
5.6 Write Byte (Immediate)

Now that we know how to read most any address, the next step is writing to memory. For that we use a new opcode 03 (Write Byte Immediate).

```
0000000000001010010110100101011010001010010082774080000010100101101001000101101
0000000000000306206203062064206603061104062060040011000007030620620306206420660 30

0E8E810100100000011010010110100100010110 10E8E8101001000000110100101101001000 1011
6F0F004062060000020306206203062064206603 06F0F0040620600000203062062030620642066 0

010E8E8101001000000110100101101001000101101 0E8E81010010000001101001011010010001 0
306F0F004062060000020306206203062064206603 06F0F00406206000002030620620306206420 6

11010E8E8101001000000110100101101001000101101 0E8E81010010000001101001011010010 00
60306F0F004062060000020306206203062064206603 06F0F00406206000002030620620306 2064 2
                                                                           BH
1011010E8E8101001000000110100101101001000101101 0E8E81010010000001101001011010HC0
0660306F0F0040620600000203062062030620642066030 6F0F00406206000002030620620306▒2

1000101101 0E8710100101101001000101101 0E8E8101001000000710100101101001000101101 0E
0632066030 6F0F040620620306206420660306F0F00406206000000F030620620306206420660306F
                                                                  BH
8E8101001000000710100101101001000101101 0E8E8101001000000710100101101 0HC0100010 11
0F00406206000000F030620620306206420660306F0F00406206000000F0306206203 06▒2064206 60
```

- Read QB000 (= 12), then change to 7F

The sequence of bytes is similar to the read message, except the address field now contains the data to be downloaded to the PC503.



5.7 Cabling

The recommended cabling between the RS-232 port of the PC500 or PC2000 and the RS-232 COM port is with the NLCC-3100 cable kit. Any other non-Westinghouse cable kit is not recommended.

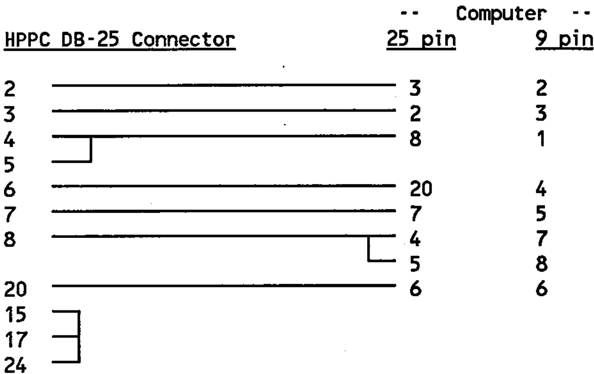
PC500 Pinout (DB-15 female)

1	shield	11	+9 VDC (current loop forcing power supply)
2	RX-	12	shield
3	5.2 VDC	13	+9 VDC (current loop forcing power supply)
4	24 VDC	14	5.2 VDC
5	shield	15	shield
6	TX+		
7	TX-		
8	shield		
9	RX+		
10	N.C.		

6.0 HPPC Program Loader Protocol

The manual describes the asynchronous program loader protocol of the HPPC 1500 and HPPC 1700. This protocol is used on either the IOP or SIM RS-232 port.

6.1 Cabling



This cabling will connect a computer that has an IBM PC standard serial port to the serial port connection on either the HPPC Input Output Processor (IOP) or the serial port connection on the Serial Interface Module (SIM). The SIM is connected to the HPPC via the remote I/O High Speed Link. Note that the HPPC remote I/O system permits remote Advanced Program Loaders (APL), or devices emulating APLs, to communicate with the HPPC as if they were directly connected to the HPPC IOP.

6.2 Configuring HPPC

The HPPC will accept communications over its serial ports in any of three distinct formats; synchronous HDLC, asychronous 9600 baud and asynchronous 1200 baud. The serial port on the HPPC defaults to the last protocol used. To change to the asynchronous protocol, follow this step by step procedure:

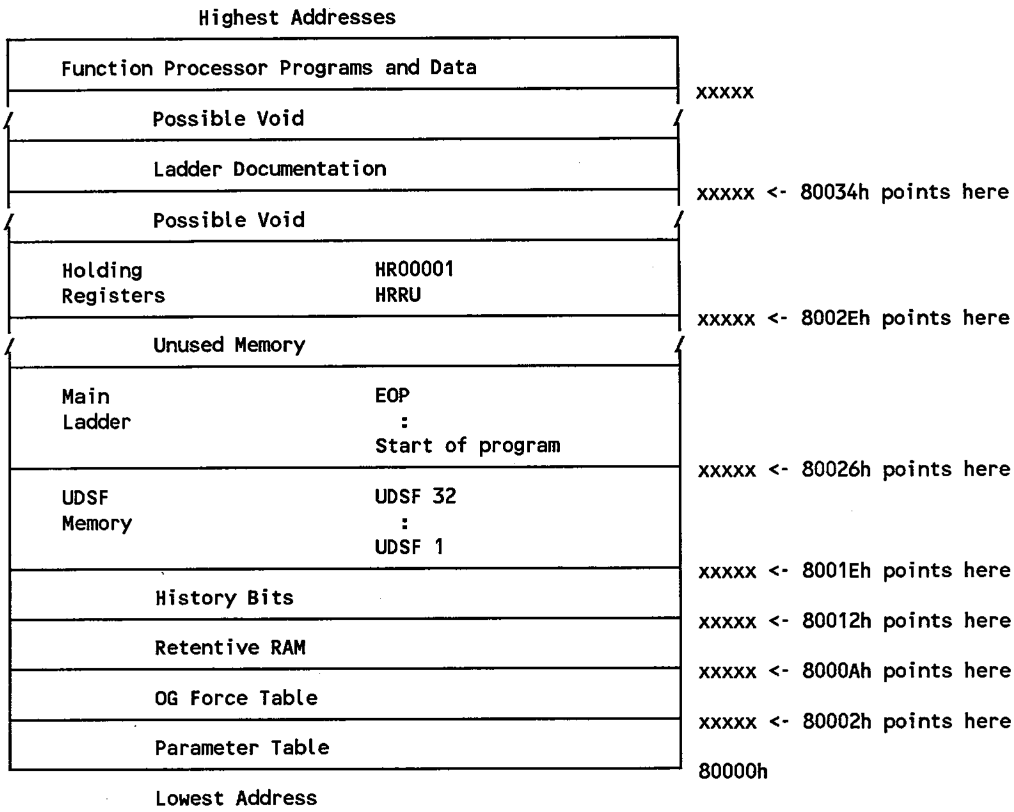
- Remove and restore power to the HPPC. The HPPC starts looking for synchronous, asynchronous 9600 baud and asynchronous 1200 baud, in that order.
- Immediately after applying power to the HPPC, send, and continue sending a valid asynchronous message to the HPPC at either 9600 or 1200 baud.
- The HPPC should auto-synch and respond within 10 seconds. If not, repeat with the first step.
- If the HPPC does properly respond, the default communications becomes asynchronous and this new information is written to the parameter table. The next time the HPPC powers up, this restart sequence will not have to be repeated.

If the HPPC does not see a valid asynchronous message during the start-up phase, it will revert the protocol stored in the parameter table. If that protocol was already asynchronous, then the HPPC remains in asynchronous mode.

NOTE:
- A computer can also be connected to any of the serial ports on any of the SIMs. The communications protocol for each SIM is configurable and is set up through the NLSW-158x program loader software (available from Westinghouse).
- All messages should not take longer than 10 seconds to transmit
- No intercharacter gaps of longer than 500 mS are allowed.
- Bit rates of 19200, 9600 and 1200 are supported by the SIM
- Bit rates of 9600 and 1200 are supported by the IOP (and NLSW-158x APL software)

- If a character is received while transmitting, the transmit should be aborted and the receiver reset. This condition indicates that the frames have gone out of synchronization.

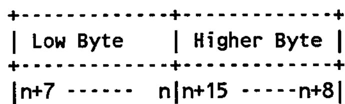
6.3 HPPC Memory Map



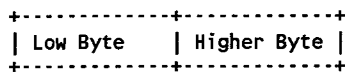
6.3.1 HPPC Data/Operand Field

All user memory starts at address 80000h. Memory addresses range from 00000h to FFFFFh (0 - 1 Meg). It requires 6 hexadecimal digits to describe on memory address, therefore three bytes are needed to store an address. Some address pointers use four bytes (the upper byte is set to zero). The HPPC protocol defines addresses in this manner:

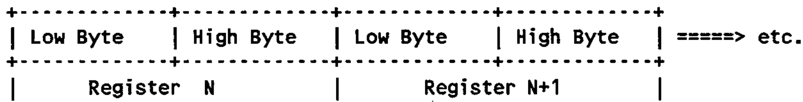
The format for bit oriented data appears below. This format occurs in 1 to 16 byte groupings



The format for the "number of bytes" operand appears below. This operand can either be one or (as shown below) two bytes in length.



The format for "Register Data Operand" is shown below. This operand can be from 2 to 256 bytes in length.



```
+------------+------------+------------+------------+
| Low Byte   | High Byte  | Low Byte   | High Byte  | =====> etc.
+------------+------------+------------+------------+
|      Register  N        |       Register N+1      |
```

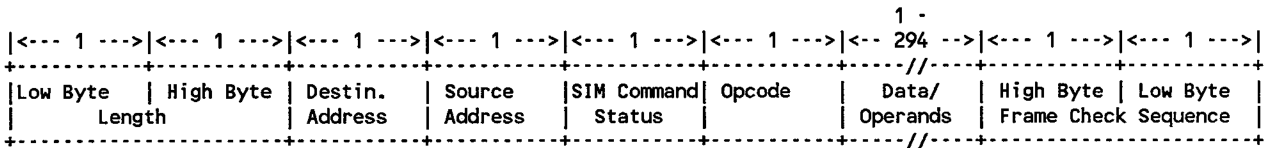
6.3.2 Holding Registers

Note that the Holding Register area is indexed downward in memory. HR1 located in the highest address in the Holding Register table, while HR2 is the next 16 bit address lower in memory. The base address of the highest Holding Register and the length of the HR table in memory can be found by reading the parameter table.

6.4 Asynchronous Protocol

This section describes the protocol used to transmit data from a smart device emulating an Advanced Program Loader (APL). Note that all messages originate with the APL. The APL can request messages from either the Input Output Processor (IOP) or the Serial Interface Module (SIM). However, the protocol requires that messages passed to the IOP and the SIM contain different destination addresses (see page 57).

6.4.1 Message Structure



1	1	1	1	1	1	1 - 294	1	1
Low Byte	High Byte	Destin. Address	Source Address	SIM Command Status	Opcode	Data/ Operands	High Byte	Low Byte
Length							Frame Check Sequence	

6.4.2 Address Fields

When sending a message to the HPPC, you must specify both your address (source) and the address of the device that you wish to speak with (IOP, SIM, etc.). The HPPC protocol assigns a unique address to each item on the remote I/O system. A smart device emulating an APL always has a source address of 40h.

SIM 0:	00h
SIM 1:	01h
SIM 2:	02h
:	:
SIM F:	0Fh
IOP	20h
APL	40h
Global	FFh

When sending a request to the HPPC, always select the destination address of 20h (IOP) *even if you are connected to a SIM.* When the response is returned from the IOP, however, the source address will have been changed to the address of the SIM you are connected to.

6.4.3 Opcodes

The following table summarizes the available opcodes that are supported by the HPPC / APL connection:

Initialize	00h
Retest	01h
Configure	02h
Edit Off	03h
Edit On	04h
Search/Address	05h
Search/Network	06h
Search/Register	07h
Repack	08h
Monitor	09h
Bit Write	0Ah
Data Write	0Bh
Read	0Ch
Read Para. Table	0Dh
Delete	0Eh
Insert	0Fh
Error	10h
Delete Text	20h

Delete Label	21h
General Edit On	22h
General Edit Off	23h

Refer to the Westinghouse document "HPPC System Communications Specification" for information on these other Opcodes.

6.4.3.1 **Read Parameter Table** (Opcode 0Dh)

Data and programs can be relocated. For this reason, it is important to find the starting addresses of various memory sections. Once, for example, the starting address of Holding Registers is known, it is a simple matter to read or write that address. When the Read Parameter Table function is transmitted to the HPPC, the HPPC responds with the starting location in memory of the Parameter Table. This parameter table contains valuable information (location of HRs in memory, location of main and UDSF ladder, etc.).

```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|Low Byte   | High Byte | Destin.   | Source    |  00 hex   |  0D hex   | Bytes to  | High Byte | Low Byte  |
|       Length          | Address   | Address   |           |           |   Read    | Frame Check Sequence  |
+-----------------------+-----------+-----------+-----------+-----------+-----------+-----------------------+
```

The response looks like:

```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<-- 1 -->|<-- 1 -->|<--- 3 --->|<- xx ->|<--- 1 --->|<--- 1 --->|
+-----------+-----------+-----------+-----------+---------+---------+-----------+--------+-----------+-----------+
|Low Byte   | High Byte | Destin.   | Source    | 00 hex  |0D hex   |   Base    | Data   | High Byte | Low Byte  |
|       Length          | Address   | Address   |         |         |  Address  |        | Frame Check Sequence  |
+-----------------------+-----------+-----------+---------+---------+-----------+--------+-----------------------+
```

If the message sent to the HPPC specified a number of bytes to read (greater than 0), then the response includes not only the starting address of the parameter table, but also the first portion (up to 256 bytes) of that parameter table.

A typical message could look like the following:

```
Sent -> 09 00 20 40 00 0D FF D4 86
Rec  <- 0A 01 40 00 00 0D 00 00 08 8A 00 56 04 08 00 6A 06 08 00 80 08 08 00 02 0C 08 00 02 0C 08 00 40 00
                          PT base  LXSum OG Force    OG Image    IOP1 base   IOP 2 base  Hst bit add # hist bits

        00 00 10 01 82 0C 08 00 03 00 00 00 85 0C 08 00 03 00 00 00 FE BF 0A 00 00 00 ....
        Hgst. Hgst. UDSF addr.              Ladder base             HR base add
        UDSF  Main                          address
        Ntwk  Ntwk
```

Note that the 54th through the 57th byte form an address that points to R00001 (Holding Register 1).

```
ABFFE   R00001
ABFFC   R00002
  :       :
```

Subsequent Holding Registers are placed in successively lower memory addresses. Note that the memory is addressed on *byte* boundaries. You must decrement by *two* in order to access the next word address.

6.4.3.2 **Read Memory** (Opcode 0Ch)

There are two methods for reading blocks of data from the HPPC. Read Memory (Opcode 0Ch) and Monitor (Opcode 09h) both read memory. Refer to the section on Monitor (page 62) for more information on the differences between these opcodes. Note that the READ MEMORY function is executed immediately by the HPPC, while the MONITOR function returns the value requested at the end of the next available HPPC scan.

The Read opcode is typically used to read a contiguous block of system memory.

```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|Low Byte   | High Byte | Destin.   | Source    |  00 hex   |  0C hex   | Bytes to  | High Byte | Low Byte  |
|       Length          | Address   | Address   |           |           |  Read     | Frame Check Sequence  |
+-----------------------+-----------+-----------+-----------+-----------+-----------+-----------------------+
```

The read acknowledgment and data returned looks like this:

```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<-- 1 -->|<-- 1 -->|<--- 1 --->|<- n ->|<--- 1 --->|<--- 1 --->|
+-----------+-----------+-----------+-----------+---------+---------+-----------+---//--+-----------+-----------+
|Low Byte   | High Byte | Destin.   | Source    | 00 hex  | 0C hex  | Number of | Data  | High Byte | Low Byte  |
|       Length          | Address   | Address   |         |         |  Bytes    |       | Frame Check Sequence  |
+-----------------------+-----------+-----------+---------+---------+-----------+---//--+-----------------------+
```

Refer to Section 6.3.1, page 55 for information on the format of the Data field.

NOTE: The normal re-synchronization procedure is to read location zero with length zero. The HPPC responds with a READ acknowledgment and zero bytes of data (no data).

6.4.3.3 **Data Write** (Opcode 0Bh)

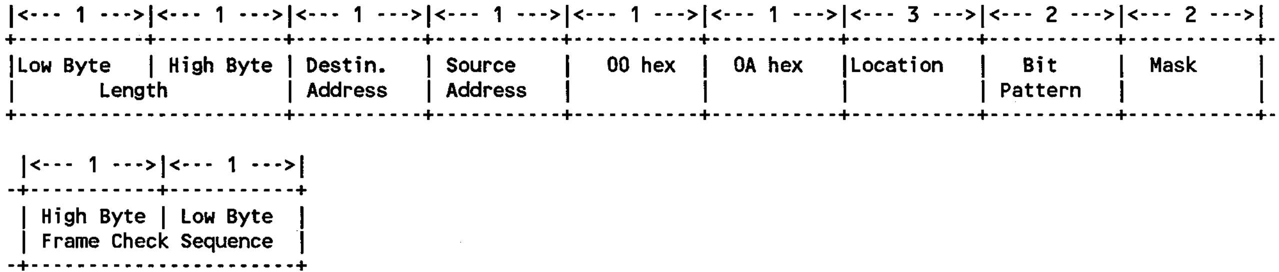
```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 3 --->|<--- 1 --->|<--- n --->|
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+----//-----+-
|Low Byte   | High Byte | Destin.   | Source    |  00 hex   |  0D hex   |Starting   |Number of  |Data to    |
|       Length          | Address   | Address   |           |           |Address    |Bytes      |Write      |
+-----------------------+-----------+-----------+-----------+-----------+-----------+-----------+----//-----+-

 |<--- 1 --->|<--- 1 --->|
-+-----------+-----------+
 | High Byte | Low Byte  |
 | Frame Check Sequence  |
-+-----------------------+
```

The data returned looks like this:

```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<-- 1 -->|<-- 1 -->|<--- 1 --->|<--- 1 --->|
+-----------+-----------+-----------+-----------+---------+---------+-----------+-----------+
|Low Byte   | High Byte | Destin.   | Source    | 00 hex  | 0C hex  | High Byte | Low Byte  |
|       Length          | Address   | Address   |         |         | Frame Check Sequence  |
+-----------------------+-----------+-----------+---------+---------+-----------------------+
```

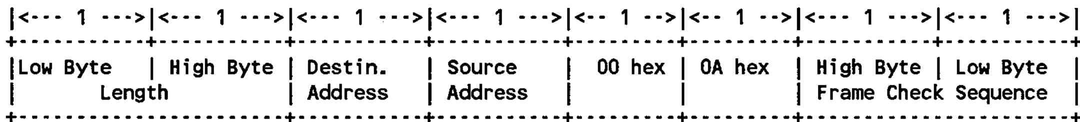
6.4.3.4 **Bit Write** (Opcode 0A)

This command is used when it is desired to change just one or more bits in a memory location without disturbing the other bits. The IOP does a read/modify/write operation with the Multibus locked to insure that no other bits are changed.



```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 3 --->|<--- 2 --->|<--- 2 --->|
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-
|Low Byte   | High Byte | Destin.   | Source    |  00 hex   |  0A hex   |Location   |   Bit     |  Mask     |
|       Length          | Address   | Address   |           |           |           | Pattern   |           |
+-----------------------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-

 |<--- 1 --->|<--- 1 --->|
-+-----------+-----------+
 | High Byte | Low Byte  |
 | Frame Check Sequence  |
-+-----------------------+
```

The data returned looks like this:



```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<-- 1 -->|<-- 1 -->|<--- 1 --->|<--- 1 --->|
+-----------+-----------+-----------+-----------+---------+---------+-----------+-----------+
|Low Byte   | High Byte | Destin.   | Source    | 00 hex  | 0A hex  | High Byte | Low Byte  |
|       Length          | Address   | Address   |         |         | Frame Check Sequence  |
+-----------------------+-----------+-----------+---------+---------+-----------------------+
```

This command is set up to do bit-writes on 16 bit words, however, by use of the Mask operand, this command can also be used to perform a bit write on bytes. Setting a bit in the mask will cause the corresponding bit in the Bit Pattern field to overwrite the present bit in the specified location.

See Section 6.3.1, page 55 for a description of the format of the Bit Pattern and Mask words.

6.4.3.5 Monitor (Opcode 09h)

```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 3 --->|<--- 1 --->|<--- 1 --->|
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-
|Low Byte   | High Byte | Destin.   | Source    |  00 hex   |  09 hex   |Network    |Number of  |Number of  |
|       Length          | Address   | Address   |           |           |Number     |Byte Addrs |Word Addrs |
+-----------------------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-

 |<--- 1 --->|<--- n --->|<--- n --->|<--- n --->|<--- 1 --->|<--- 1 --->|
-+-----------+-----//----+-----//----+-----//----+-----------+-----------+
 |Number of  |List of    |List of    |List of    | High Byte | Low Byte  |
 |DWord Addrs|Byte Addrs |Word Addrs |DWord Addrs| Frame Check Sequence  |
-+-----------+-----//----+-----//----+-----//----+-----------+-----------+
```

The data returned looks like this:

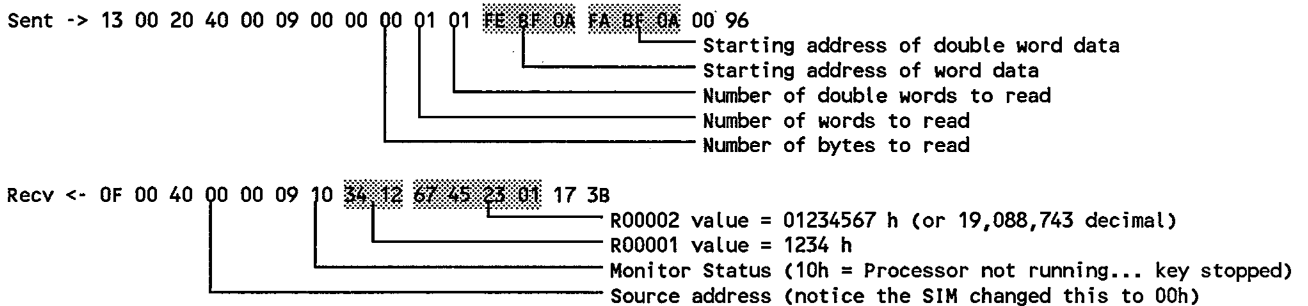
```
|<--- 1 --->|<--- 1 --->|<--- 1 --->|<--- 1 --->|<-- 1 -->|<-- 1 -->|<-- 1 -->|
+-----------+-----------+-----------+-----------+---------+---------+---------+
|Low Byte   | High Byte | Destin.   | Source    | 00 hex  | 09 hex  |Monitor  |
|       Length          | Address   | Address   |         |         |Status   |
+-----------------------+-----------+-----------+---------+---------+---------+

 |<--- n --->|<--- n --->|<--- n --->|<--- 1 --->|<--- 1 --->|
-+-----//----+-----//----+-----//----+-----------+-----------+
 |   Byte    |   Word    |Double Word| High Byte | Low Byte  |
 |   Data    |   Data    |    Data   | Frame Check Sequence  |
-+-----//----+-----//----+-----//----+-----------+-----------+
```

<u>Monitor Status</u>

Note that only one of these bits will be set at a given time.

Bit 0 Special Function data valid
Bit 1 Network not found
Bit 2 Network MCR-controlled
Bit 3 Network Skip-controlled
Bit 4 Processor not running
Bit 5-7 Unused and set to 0

For example, the following request was sent to the HPPC for R00001 and double register R00002 (R00002 and R00003)



Two advantages of the MONITOR function are:

- All data is from the same scan and at end of scan
- Random addresses may be mixed into one request

6.5 Frame Check Sequence

The FCS used in asynchronous communications must be generated/checked with software unlike the FCS used in HDLC which is usually generated/checked with hardware. Reverse CCITT-CRC is used in asynchronous communications, whereas CCITT-CRC is used in HDLC. This FCS is calculated over the entire message including the length field. On reception, if the FCS is calculated over the entire message including the received FCS, the result will be 0 for correct message reception. The following program (written in 8086 assembler) may be used to generate the FCS for asynchronous communications. Note that this look up table must be used, since due to an early error, one location in this table does not match the $X^{16} + X^{12} + X^5 + 1$ polynomial used by the reverse CCITT-CRC algorithm.

To assist users who have access to scanners, the following program was printed in HP LaserJet+ Courier 10. This listing is also available from Westinghouse. Contact the Advanced Products Support Center at 800-542-7883 in the U.S. from 8 am to 5 pm Eastern Time, Monday through Friday.

```
;-----------------------------------------------------------------;
;FindCRC                                                          ;
; Entry: DS:[BX]   Points to start of message that CRC must be   ;
;                  calculated for                                 ;
;        CX        Number of bytes in message                     ;
;                                                                 ;
; Exit:  AX        CRC                                            ;
;-----------------------------------------------------------------;

     prog
     assume  cs:prog

CRC_Table:
DW      0,   4129,   8258,  12387,  16516,  20645,  24774,  28903
DW -32504, -28375, -24246, -20177, -15988, -11859,  -7730,  -3601
DW   4657,    528,  12915,   8786,  21173,  17044,  29431,  25302
DW -27847, -31976, -19589, -23718, -11331, -15460,  -3073,  -7202
DW   9314,  13379,   1056,   5121,  25830,  29895,  17572,  21637
DW -23190, -19125, -31448, -27383,  -6674,  -2609, -14932, -10867
DW  13907,   9842,   5649,   1584,  30423,  26358,  22165,  18100
DW -18597, -22662, -26855, -30920,  -2081,  -6146, -10339, -14404
DW  18628,  22757,  26758,  30887,   2112,   6241,  10242,  14371
DW -13876,  -9747,  -5746,  -1617, -30392, -26263, -22262, -18133
DW  23285,  19156,  31415,  27286,   6769,   2640,  14899,  10770
DW  -9219, -13348,  -1089,  -5218, -25735, -29864, -17605, -21734
DW  27814,  31879,  19684,  23749,  11298,  15363,   3168,   7233
DW  -4690,   -625, -12820,  -8755, -21206, -17141, -29336, -25271
DW  32407,  28342,  24277,  20212,  15891,  11826,   7761,   3696
DW    -97,  -4162,  -8227, -12292, -16613, -20678, -24743, -28808
DW -28280, -32343, -20022, -24085, -12020, -16083,  -3762,  -7825
DW   4224,    161,  12482,   8419,  20484,  16421,  28742,  24679
DW -31815, -27752, -23557, -19494, -15555, -11492,  -7297,  -3234
DW    689,   4752,   8947,  13010,  16949,  21012,  25207,  29270
DW -18966, -23093, -27224, -31351,  -2706,  -6833, -10964, -15091
DW  13538,   9411,   5280,   1153,  29798,  25671,  21540,  17413
DW -22565, -18438, -30823, -26696,  -6305,  -2178, -14563, -10436
```

```
DW   9939,  14066,   1681,   5808,  26199,  30326,  17941,  22068
DW  -9908, -13971,  -1778,  -5841, -26168, -30231, -18038, -22101
DW  22596,  18533,  30726,  26663,   6336,   2273,  14466,  10403
DW -13443,  -9380,  -5313,  -1250, -29703, -25640, -21573, -17510
DW  19061,  23124,  27191,  31254,   2801,   6864,  10931,  14994
DW   -722,  -4849,  -8852, -12979, -16982, -21109, -25112, -29239
DW  31782,  27655,  23652,  19525,  15522,  11395,   7392,   3265
DW  -4321,   -194, -12451,  -8324, -20581, -16454, -28711, -24584
DW  28183,  32310,  20053,  24180,  11923,  16050,   3793,   7920

CALC_CRC: PUSH BX       ;save these registers
          PUSH CX
          PUSH SI
          MOV  SI,BX
          CLD           ;forward string (not necessary)
          XOR  AX,AX    ;zero out
          DEC  AX       ;start at FFFFh
CRCCALC:  MOV  BL,[SI] ;BL <- datab
          XOR  BL,AH    ;BL <- datab XOR high CRC
          XOR  BH,BH    ;BH <- 0
          SHL  BX,1     ;mult x 2 to get word offset
          MOV  AH,AL
          XOR  AL,AL    ;AX <- low crc * 100h
          XOR  AX, WORD PTR CS:CRC_TABLE[BX]
                        ;AX <- low crc * 100h XOR with
                        ;Table (datab XOR high crc)
          INC  SI       ;point to next byte in message
          LOOP CRCCALC ;decrement CX and continue til 0

          POP  SI       ;restore saved registers
          POP  CX
          POP  BX
          RET           ;return to calling program
```

6.6 Key Parameter Table Locations

80000 Ladder Checksum
80002 OG Force Bit Table Base Address
80006 Retentive RAM, Base Address, OG Image
8000A Retentive RAM, Base Address, IOP 1
8000E Retentive RAM, Base Address, IOP 2
80012 History Bit Table Base Address
80016 History Bit Table Length
80018 Highest History Bit used
8001A UDSF Ladder, Highest Network number
8001C Main Ladder Highest Network number
8001E UDSF Ladder Base address
80022 UDSF Ladder length
80026 Main Ladder Base address
8002A Main Ladder Length
8002E Holding Register Base Address
80032 Holding Register Length (HRU)
80034 Documentation Base Address
80038 Documentation Length
8003C LP Input Group Image Base address, IOP 1
80040 Number of IGs, IOP 1
80042 LP Input Group Image Base address, IOP 2
80046 Number of IGs, IOP 2
80048 LP Output Group Image Base address, IOP 1
8004C Number of OGs, IOP 1
8004E LP Output Group Image Base address, IOP 2
80052 Number of OGs, IOP 2
80054 LP Input Register Image Base address, IOP 1
80058 Number of IRs, IOP 1
8005A LP Input Register Image Base address, IOP 2
8005E Number of IRs, IOP 2
80060 LP Output Register Image Base address, IOP 1
80064 Number of ORs, IOP 1
80066 LP Output Register Image Base address, IOP 2
8006A Number of ORs, IOP 2
8006C 13 spare words
80086 Memory Partition Block CRC
80088 High Byte: LP Firmware Version, Low Byte: LP ID Code
8008A Shared Memory Base address
8008E Shared Memory Length
80090 Maximum I/O
80092 Spare
80094 Spare
80096 Spare
80098 Supported Special Functions Table (16 words)
800B8 High Byte: SM Firmware Low Byte SM ID Code

6.7 Error Response

If the HPPC does not understand a message it will either ignore the request (issue no response) or return an error response. This section describes the error responses that could be returned.

The general format of an error response is:

1	1	1	1	1	1	1	2	1	1
Low Byte Length	High Byte	Destin. Address	Source Address	00 hex	10 hex (Error)	Error Code	Edit L.O. Register	High Byte Frame Check Sequence	Low Byte

Error Codes

01h Link Error (communications error)
10h Illegal Opcode (HPPC didn't recognize the opcode)
11h Invalid Operand (Monitor: requested too much data. Other Opcodes: self-explanatory)
12h Invalid Frame Length (HPPC didn't receive the right number of characters in request)
20h Keyswitch Error (Cannot change ladder when in Program/Protect mode)
21h Search Word not found (Reference does not exist)
23h Edit Request denied (Attempted to change ladder, but another loader already editing ladder)
24h Edit Session Not in Progress (Attempted to change ladder, but Edit not previously established)
25h Mode Error (attempted to reconfigure HPPC, but not in STOP mode)
26h Channel Error (command was recognized, but cannot be executed through this HPPC port)
4Eh Illegal Channel error (

7.0 More Information

More information can be found by referring to other Westinghouse Programmable Controller literature, particularly:

- NLAM-B206 PC1100/PC1200 Systems Manual
  - PC1100/1200 Program Loader RS-232/RS-485 pinouts
  - Connecting a leased line modem (wiring diagrams)
  - Ladder functions to set serial port parameters, send and receive binary data, etc.
- NLAM-B817 NLSW785/786 Programming Manual
  - Using NLSW-785/786 on PC1000 LAN
  - Using NLSW-785/786 direct connect or on leased line or dial-up modems
- NLAM-B310 NCMZ-1799 Systems Manual
  - Pinouts of NCMZ-1799 RS-232 communications ports
- NLAM-B806 HPPC APL Manual
  - Pinouts of RS-232 Serial Port
- NLAM-B204 Distributed I/O Manual
  - Connecting HPPC remote I/O High Speed Link
- NLAM-B500 PC500 Systems Manual
  - Memory Map and PLC database
- NLAM-B2000 PC2000 Systems Manual
  - Memory Map and PLC database
- NLAM-B501 NL-580 Communications Module Systems Manual
  - Pinout for RS-232 and TTY Current Loop
  - Connecting PC500 to other RS-232 based devices
- HPPC System Memory Map and Parameter Table Technical Description
  - PLC Memory Map
  - Ladder Database
  - Internal program documentation database
- HPPC System Communication Specification
  - Asynch and HDLC message structure
  - Complete description of all IOP, SIM and APL message structures
- NLAM-B260 PLCLINK Software Manual for Lotus™ 1-2-3
  - Connect Lotus or Symphony™ to Westinghouse PLC
  - V2.1 Supports 6 Byte, PC1000 LAN, HPPC Asynch and Sync and Westnet
- NLAM-B202 PLCBIOS IBM PC to PLC Communication Drivers for IBM PC
  - Supports all protocol of PLCLINK
  - Allows programmer in C, Assembly, QuickBASIC or BASICA communicate with PLC