AN1250

Low Cost PC Interface to LONWORKS Based Nodes

With the availability of low cost PCs and software it is possible to create a professional low cost, high quality user interface on a PC to monitor or control a LONWORKS network. To show one possible method of doing this, an application was made connecting a PC to Motorola's Heating Venting Air Conditioning (HVAC) briefcase demo. Motorola's HVAC briefcase demo consists of 6 nodes (3 NEURON nodes, and 3 display nodes): 1 smart setback thermostat with LCD display, 1 compressor node with an LED display, and a fan node with an LED scroll display. The setback thermostat node with LCD display contains a real time clock, temperature sensor, and keypad.

This application will show how it is possible to develop a low cost controller/monitor on a PC. In addition, the NEURON C code and EIA–232 connections are shown to connect a NEURON CHIP to a PC.

The HVAC demo can function as a stand alone demo, or be controlled through a PC. The setback thermostat can be programmed through a keypad to set a temperature setpoint to turn on the compressor and fan. In addition, a PC can be connected through an optional NEURON based board to display and even control the setback thermostat. Figure 1 shows a block diagram of the complete system. Figure 2 shows a more detailed diagram.

The six major building blocks of this system consist of:

1. PC
2. PC application
3. PC interface to a LONWORKS network
4. PC interface application
5. LONWORKS nodes
6. LONWORKS applications

PC AND PC APPLICATION

In this application a PC was used, but a similar approach can be used with a Macintosh as well as other computers. The PC application used was Microsoft's Visual Basic. Visual Basic is an object oriented programming language with the capability to display event driven graphics. Visual Basic is similar in a lot of ways to how a NEURON C program works. When an event becomes true, Visual Basic code behind an graphic object is executed. A sequencer polls each object to determine when it is true. Visual Basic v3.0 supports EIA–232 communications making it easy to connect to a NEURON CHIP.

Visual BASIC is an extremely powerful, easy to use graphical programming language supporting Dynamic Data Exchange (DDE), Dynamic Link Libraries (DLL), and Object Linking and Embedding (OLE). Visual BASIC has the capability to exchange data with other Windows programs that support DDE, such as a spreadsheet, database, or graphics program. Dynamic Link Libraries are libraries, written by other people or by yourself, that your program can call. It is analogous to calling a NEURON C object code function written by someone else. OLE is a method through which Windows applications can use each others resources. For example, a Visual BASIC program can have data presented inside the program as though Excel is running inside it. Visual BASIC is available in DOS and Window versions. In this application, the Visual BASIC windows version was used.

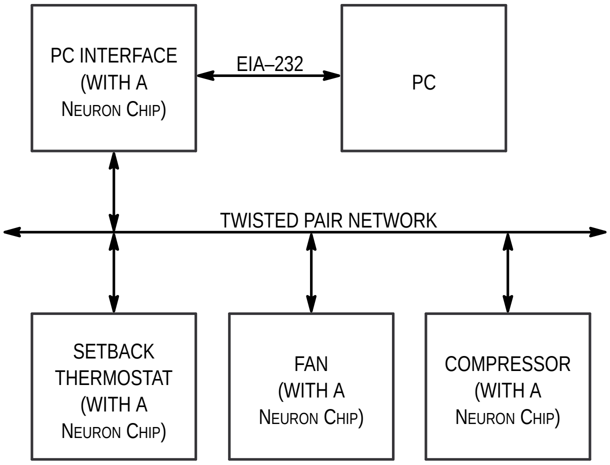


Figure 1. HVAC Demo Block Diagram with PC

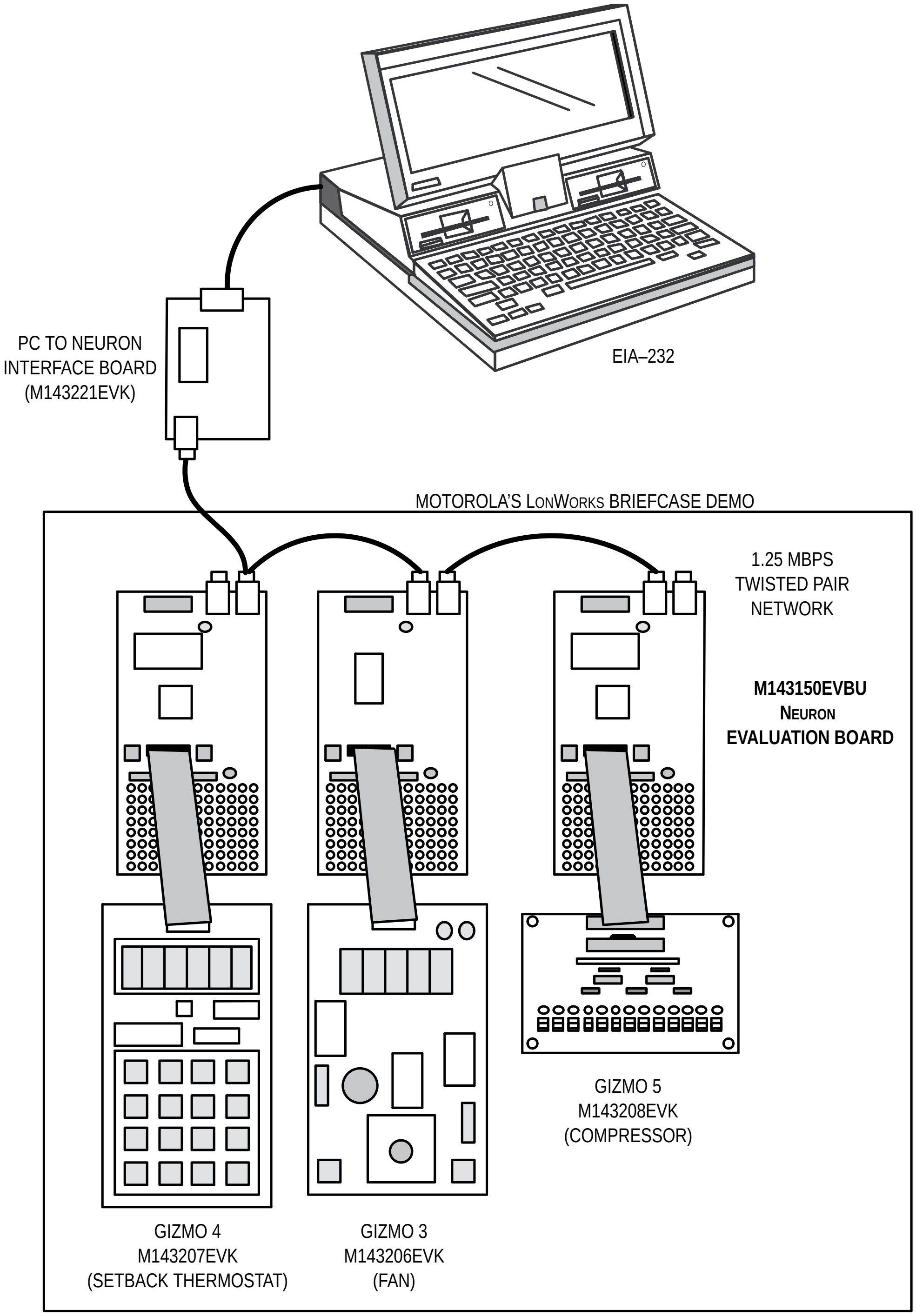
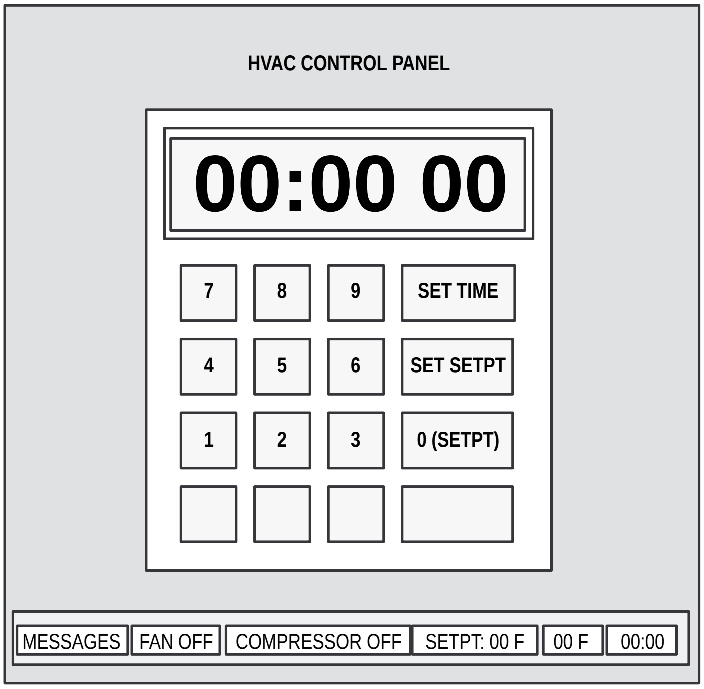


Figure 2. HVAC Block Diagram with PC

BASIC OPERATION

As shown in Figure 3, the Visual BASIC application displays a picture of a keypad similar to the one used in the stand alone HVAC demo. The user can set the time and temperature setpoint for activation of the fan and compressor. The current setpoint can also be displayed. When neither the current setpoint nor the time or temperature setpoint is being displayed, the display defaults to the HVAC's time and temperature. Pressing the keypad in the Visual BASIC application is the same as pressing the keys in the HVAC demo.



HVAC CONTROL PANEL

00:00 00

MESSAGES | FAN OFF | COMPRESSOR OFF | SETPT: 00 F | 00 F | 00:00

Figure 3. Visual BASIC Application User Interface

PC INTERFACE TO A LonWorks NETWORK

The PC interface to the LonWorks network consists of a Neuron Chip communicating through its serial port to the PC. The Neuron Chip supports only half duplex. The communication lines of the Neuron Chip are tied to the LonWorks network. In this case, the LonWorks network is differential direct connect. Since the Neuron Chip supports only half duplex, it must be waiting at an `io_in` function call before data arrives, or else it will miss the data. The PC is set up using RTS/CTS protocol. The Neuron Chip asserts Clear To Send (CTS) so the PC can send data if it has any.

The PC uses Request To Send (RTS) to determine whether or not it can accept data, dependent upon how full its buffers are. RTS is optional in this application because the PC interface board will never fill the PC buffers. RTS will always be asserted (+12 volts) by the PC.

EIA–232 signals are typically between +12 and – 12 volts with –12 volts being the idle state. Optionally, the PC will assert RTS around +12 volts, signifying it is ready to receive data. The PC cannot send data out until the Neuron Chip asserts CTS which arrives at the PC around +12 volts. An EIA–232 transceiver such as Motorola's MC145407 is needed to convert the Neuron Chip's CMOS I/O to EIA–232 levels, and vice–versa. Figure 4 shows the PC to Neuron interface connections used in this application. The Neuron interface board was designed so a standard DB9F to DB9M straight through cable can be used.

Figure 5 shows the PC interface schematic. If RTS is used, the 47 kΩ resistor in Figure 5 keeps RTS high (+10 volts) in case the PC cable gets disconnected.

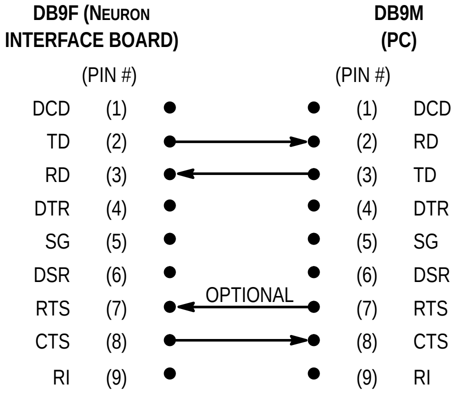


Figure 4. EIA–232 Interface Connections

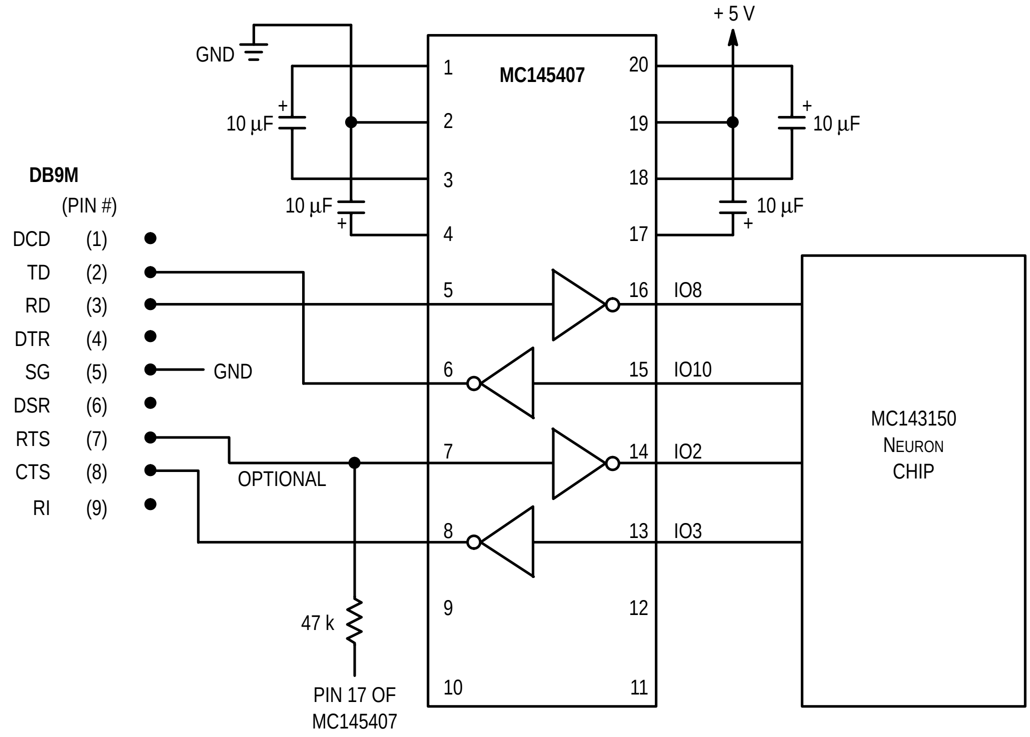


Figure 5. PC Interface

PC INTERFACE APPLICATION

The setback thermostat node is the brain for the stand alone HVAC application. The PC interface application receives data from the PC; using network variables, it tells the smart setback to set the time and temperature. On the other side, the PC interface application sends data every 250 ms to the PC; including the HVAC's time, temperature, and setpoint. The formats to/from the PC are as follows:

NEURON Interface to PC Packet Format:

<B><time><temperature><setpoint>
<compressor on/off><fan on/off><CR>

where:

field	**description**
<**B**>	start of packet
<**time**>: hh:mm	hh is hours, mm minutes
<**temperature**>	xx (in Fahrenheit)
<**setpoint**>	xx (in Fahrenheit)
<**compressor on/off**>	0: off 1: on
<**fan on/off**>	0: off 1: on
<**CR**>	carriage return

PC to NEURON Interface Packet Format:

<D><command><data>

where:

field	**description**
<**D**>	start of packet
<**command**>	"1": set time "2": set setpoint
<**data**>	if<command>= "1"then HHMM where HH: hours MM: minutes if <command> = "2" then xx where xx is 2 digit temperature setpoint

The PC interface application file is shown at the end of this application note.

LONWORKS NODES AND LONWORKS APPLICATIONS

The LONWORKS nodes and LONWORKS applications are documented with the HVAC briefcase demo and not covered in this application note.

CONCLUSION

It took approximately 2 weeks to build up the PC interface hardware, write the NEURON C code for the interface, learn Visual BASIC, and write the Visual BASIC code. The result was an easy to use, high quality PC graphical user interface with application code to monitor and control a LONWORKS network, more specifically, a HVAC application.

More complex code, such as time of day functions to turn on/off the fan or air conditioner can now be performed on a PC. This will save LONWORKS node memory and time to perform these functions. A lower cost MC143120 node might now be used in place of an MC143150 node.

The advantages of this application are the low cost, power, and resources of the PC including readily available hardware and software for the PC and ease of use. MIP drivers and applications, API libraries, or more expensive PC interfaces are unnecessary.

This low cost PC interface is not meant for a network manager or protocol analyzer. It is possible to set up the PC to do some of these functions in a limited way, such as using the PC for sending network management commands to the Neuron interface. If variables are bound to the Network interface, or if the Network interface polls other LonWorks nodes, the PC can then display their values.

The Neuron Chip to PC interface may also be used to connect a modem and other serial devices. There are several good graphical user interfaces on the market, including Microsoft's Visual C++, National Instruments Labview, and Wonderware. These products differ significantly in cost, methodology, and learning curves. Another option is to develop your own graphical user interface using such programs as Zinc, Borland C++, and Microsoft C++, to name a few.

To sum up, the PC can be an inexpensive way to monitor and control a LonWorks network. Using existing PC software, a Neuron Chip can be used to interface a PC to an existing or new LonWorks network.

SOURCE CODE FOR THE PC INTERFACE NODE

```
/*********************************************************************************************
Filename: pctobc.nc
Copyright Motorola, Inc

0.1    02/17/94   DRS    original
       03/30/94   DRS    Change so this nodes polls fan and compressor nodes
                         add polling NVcomp_state_in
                         add compress_state, fan_state
       01/11/95   DRS    documentation

Description:    Be an interface between a PC (laptop) and briefcase
                demo. Will allow PC to change settings (time,
                temperature setpoint) on demo. Also pass info.
                every 500 ms from demo to PC.

                Packets to PC will be in the following format:
                <start><time><temperature><setpt><compressor><fan><CR>
                      where  <start> = ‘B’
                             <time> = hrmn where hr:hours, mn:minutes
                             <temperature> = tt (degrees F, 0 = 99)
                             <setpt> = ss (degrees F, 0 – 99)
                             <compressor> = 1:on, 0:off
                             <fan> = 1:on, 0:off
                note:  all data displayed on PC is what is sent over.
                      No range checking is done by the PC.

Link Memory Usage Statistics:
ROM Usage:
       System Data                         2  bytes
       Application Code & Const Data     743  bytes
       Library Code & Const Data           0  bytes
       Self–Identification Data           18  bytes
                                       –––––
       Total ROM Requirement             763  bytes
       Remaining ROM                   15621  bytes

EEPROM Usage:     (not necessarily in order of physical layout)
       System Data & Parameters           74  bytes
       Domain & Address Tables            20  bytes
       Network Variable Config Tables     18  bytes
       Application EEPROM Variables        0  bytes
       Library EEPROM Variables            0  bytes
       Application Code & Const Data       0  bytes
       Library Code & Const Data           0  bytes
                                       –––––
       Total EEPROM Requirement          112  bytes
       Remaining EEPROM                  400  bytes
```

```
RAM Usage:        (not necessarily in order of physical layout)
       System Data & Parameters           572  bytes
       Transaction Control Blocks         109  bytes
       Appl Timers & I/O Change Events      8  bytes
       Network & Application Buffers      300  bytes
       Application Ram Variables          154  bytes
       Library RAM Variables                0  bytes
                                        -----
       Total RAM Requirement             1143  bytes
       Remaining RAM                      905  bytes

required header files : control.h

Notes:
1.  PCPLUS communication program and EIA232:
         Set  up PCPLUS on the PC the following way:
             command      description
             -------      -----------
             pcplus<CR>   run communication program
             <CR>         do this after
         RTS  is an output from the PC staying high (+12v) until
             the PC’s buffers are full, then it goes low (–12V).
         CTS  is an input to the PC enabling it to transmit.

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

/******************************** Compiler directives **************************************/

#pragma  scheduler_reset
#pragma  enable_io_pullups

#pragma  num_addr_table_entries 1
#pragma  one_domain
#pragma  app_buf_out_priority_count 0
#pragma  net_buf_out_priority_count 0


#define  timerl 100        // bring CTS low every 100 ms to check for PC data
#define  max_char_from_PC 30
#define  max_packet_size 60    // this # should be 2’s max_char_from_PC

struct temp_time {
    unsigned int temp;
    unsigned int minutes;
    unsigned int hours;
};

struct temp_time data_out;

struct time {
    unsigned int hours;
    unsigned int minutes;
};


/******************************** Include files ***************************************/
#include <control.h>

/********************************* I/O Objects ****************************************/

IO_3 output bit CTS;       // clear to send output
IO_2 input bit RTS;        // optional request to send input
IO_8 input serial baud(4800) RXD;       // read data from PC
IO_10 output serial baud(4800) TXD;       // send data to PC
```

```
/****************************** Network Variables *************************************/
network input struct temp_time pctobc_temp_in;    // temperature (setback node)
network input struct temp_time pctobc_setpt_in;   // setpoint (setback node)
network input struct time NV_time_in;   // BC time
network input boolean NVfan_state_in    // TRUE: fan is flashing
network input boolean NVcomp_state_in;  // TRUE: compressor is on
network output struct temp_time bind_info(unackd)  NV_timesetpt_out;
    // send setback node time and set point data

/************************ Network resource tuning pragmas *****************************/
// none

/************************************ Globals *****************************************/
char input_but[max_packet_size];        // complete packet from PC
char input_buf1[max_char_from_PC];        // Input from PC (1st time)
char input_buf2[max_char_from_PC];        // Input from PC (2nd time)
char * buf_ptr;         // pointer into buffer
boolean packet_found = FALSE;    // what looks like a good packet is not found.
boolean compress_state = FALSE;         // compressor off
boolean fan_state = FALSE;   // fan off
int last_num_chars;     // keeps a running total of characters received
int temp;
char out_char[1];
struct bcd digits;      // holds BCD data to be sent to PC
                        // digits.d1 most significant nibble in ms byte
                        // digits.d2 least significant nibble in ms byte
                        // digits.d3 most significant nibble
                        // digits.d4 least significant nibble
                        // digits.d5 most significant nibble in ls byte
                        // digits.d6 least significant nibble in ls byte

struct { // data from bc
    unsigned int hours;        // time
    unsigned int minutes;
    unsigned int temperature;
    unsigned int setpoint;
} bc_data;

struct temp_time bc_setpoint;

/************************************ Timers ******************************************/
mtimer repeating check_CTS;
mtimer repeating get_data_from_bc;      // every 500 ms poll bc
                                        // then send to PC

/*********************************** Functions ****************************************/
boolean append_packet( )
/* 0.1 drs 02/16/94 original
description:    assert CTS, append data to input_buf[ ] if any
                and return append_packet = TRUE if 1st char. = ‘D’
                and last char. is a CR.
*/
{
boolean packet;
int i;
int num_chars1; // keeps track of # of chars. read from 1st read
int num_chars2; // keeps track of # of chars. read from 2nd read

        packet = FALSE;
        num_chars1 = 0;
        num_chars2 = 0;
        io_out( CTS, 0 );     // enable cts
        num_chars1 = io_in( RXD, input_buf1, max_char_from_PC );
        io_out( CTS, 1 );     // disable cts
        // read serial buffer again in case PC can’t stop sending data
        // when CTS is disabled. Maybe PC in middle of sending a byte out.
        num_chars2 = io_in( RXD, input_buf2, max_char_from_PC );

        // append data over to where final packet goes
        if ( num_chars1 != 0 ) { // if data append it to input_buf
            for ( i = last_num_chars; i < last_num_chars + num_chars1; i++ ) {
                input_buf[i] = input_buf1[ i – last_num_chars ];            // append
            }
            last_num_chars = last_num_chars + num_chars1;
        }

        if ( num_chars2 != 0 ) { // if data append it to input_buf
            for ( i = last_num_chars; i < last_num_chars + num_chars2; i++ ) {
                input_buf[i] = input_buf2[ i – last_num_chars ];            // append
            }
            last_num_chars = last_num_chars + num_chars2;
        }

        if ( last_num_chars > 0 ) {    // something there
            if ( input_buf[0] != ‘D’ ) {
            // A packet is started and packet is invalid
                last_num_chars = 0;   // reset count of total characters read
                packet = FALSE;
            }
            else if ( input_buf[ last_num_chars – 1 ] == ‘/r’ ) {
            // 1st char. a ‘D’ and last char. a carriage return
                packet = TRUE;
            }
        } // something there
        return( packet );
}

// This function converts a hex character to 2 ASCII characters
// and sends the characters to out the TXC port to the PC
//
void putch_hex(unsigned int hex_char)
{
  out_char[0] = ( hex_char >> 4 ) & 0x0f;   // keep lower nibble
  if( out_char > 9 )
    out_char[0] += 0x37;
  else
    out_char[0] += 0x30;

  io_out( TXD, out_char, 1 );  // output 1 char. out the 232 port to the PC
  out_char[0] = hex_char & 0x0f;
  if(out_char > 9)
    out_char[0] += 0x37;
  else
    out_char[0] += 0x30;
  io_out( TXD, out_char, 1 );  // output 1 char. out the 232 port to the PC
}

//
// This function converts two ascii characters to a decimal digit
//
unsigned char to_dec(unsigned char msb,unsigned char lsb)
{
  return( (msb – 48) * 10 + (lsb – 48) );
}
```

```
/************************************* Reset *****************************************
when (reset) {
    bc_data.hours = 0;
    bc_data.minutes = 0;
    bc_data.temperature = 0;
    bc_data.setpoint = 0;

    check_CTS = timer1;     // repeating timer when to assert CTS
                            // to check for PC data
    get_data_from_bc = 500;    // every 500 ms poll bc and then send to PC

/******************************** Priority When Clauses ***********************************

// none

/****************************** Non–Priority When Clauses *********************************
when ( timer_expires(check_CTS) { // go get next character(s)
/* note:     a timer is used (‘data_timer’) because this allows
             less time in this when clause so if network data comes
             in, can spend less time in a when clause and more
             getting data out of the application buffers. If want
             to change this time, either change the timer, or even
             take it out and replace it with ‘when ( 1 )’. Remember
             that when reading in serial data, if no characters, there
             is a 20 character time out. How ever many times that is used
             may be the worst case best time to get back into this
             when clause.
*/
  packet_found = append_packet( ); // append more data if any
                         // to input_buf[].
                         // also returns true if
                         // when finds what looks like a good packet.
  check_CTS = timer1;
}

when ( packet_found ) {  // process packet
// packet format: <D><command><data>
   switch( input_buf[1] ) { // select from type of packet byte
    case ‘1’:     // set time <D><1><xxxx><CR>
      if ( last_num_chars == 7 ) {
           NV_timesetpt_out.temp = 255;      // code for do not use
           // convert ASCII HHMM in input_buf[2–5] to unsigned int.
           bc_data.hours = NV_timesetpt_out.hours =
                to_dec(input_buf[2], input_buf[3]);
           bc_data.minutes = NV_timesetpt_out.minutes =
                to_dec(input_buf[4], input_buf[5]);
      }
           break;
    case ‘2’:  // set setpoint <D><2><xx><CR>
      if ( last_num_chars == 5 ) {
           // convert ASCII set point in input_buf[2–3] to unsigned int.
           bc_data.setpoint = NV_timesetpt_out.temp =
                to_dec(input_buf[2], input_buf[3]);
           NV_timesetpt_out.hours = 255;     // code for do not use
           NV_timesetpt_out.minutes = 255;  // code for do not use
      }
      break;
    default:   // bad packet
      break;
  }
  packet_found = FALSE;    // finished last packet
  last_num_chars = 0;    // reset # of bytes collected in packet
  for ( temp = 0; temp < max_packet_size; temp++ ) {    // not needed but helps in d
      input_buf[temp] = 0;
  }
}
```

```
when ( nv_update_fails ) {
}

when ( nv_update_occurs(NV_time_in) ) {     // BC to PC time (HHMM)
    bc_data.hours = NV_time_in.hours;       // HH time
    bc_data.minutes = NV_time_in.minutes;          // MM time
}

when ( nv_update_occurs(pctobc_temp_in) ) {             // BC to PC temperature
    bc_data.temperature = pctobc_temp_in.temp;            // BC temperature
}

when ( nv_update_occurs(pctobc_setpt_in) ) {            // BC to PC setpoint
    bc_data.setpoint = pctobc_setpt_in.temp;            // BC setpoint
}

when ( nv_update_occurs(NVcomp_state_in) ) {
    if (NVcomp_state_in == TRUE) {
        compress_state = TRUE;
    }
    else {
        compress_state = FALSE;
    }
}

when ( nv_update_occurs(NVfan_state_in) ) {
    if (NVfan_state_in == TRUE;
      fan_state = TRUE;
    }
    else {
      fan_state = FALSE;
    }
}

when ( nv_update_fails(NVcomp_state_in) ) {        // compressor not responding
    compress_state = FALSE; // assume off
}

when ( nv_update_fails(NVfan_state_in) ) {         // fan not responding
    fan_state = FALSE;    // assume off
}

when( timer_expires(get_data_from_bc) ) {
// every 500 ms send data to PC and poll fan and compressor for status
    poll(NVcomp_state_in);  // compressor state
    poll(NVfan_state_in);   // fan state
    get_data_from_bc = 500; // 500 ms repetitive timer

// packet consists of: <start><time><temperature><setpt><compressor><fan><CR>
    out_char[0] = ‘B’;  // Beginning of packet character
    io_out(TXD, out_char, 1);  // send out 232 port

// output time (hours only)
    bin2bcd( (long) bc_data.hours, &digits);
    out_char[0] = digits.d5 + 0x30;         // high time BCD digit converted to ASCII
    io_out( TXD, out_char, 1);
    out_char[0] = digits.d6 + 0x30;         // low time BCD digit converted to ASCII
    io_out( TXD, out_char, 1);

// output time (minutes only)
    bin2bcd( (long) bc_data.minutes, &digits);
    out_char[0] = digits.d5 + 0x30;         // high time BDC digit converted to ASCII
    io_out( TXD, out_char, 1);
    out_char[0] = digits.d6 + 0x30;         // low time BCD digit converted to ASCII
    io_out( TXD, out_char, 1);
// output time (temperature)
    bin2bcd( (long) bc_data.temperature, &digits);
    out_char[0] = digits.d5 + 0x30;         // high temp. BCD digit converted to ASCII
    io_out( TXD, out_char, 1);
    out_char[0] = digits.d5 + 0x30;         // low temp. BCD digit converted to ASCII
    io_out( TXD, out_char, 1);

// output time (setpoint)
    bin2bcd( (long) bc_data.setpoint, &digits);
    out_char[0] = digits.d5 + 0x30;         // high stpt BCD digit converted to ASCII
    io_out( TXD, out_char, 1);
    out_char[0] = digits.d6 + 0x30;         // low stpt BCD digit converted to ASCII
    io_out( TXD, out_char, 1);

// output compressor on/off
    if ( compress_state == TRUE ) {         // compressor is on
                                            // (i.e. LEDs scrolling)
      io_out(TXD, "1", 1);      // output to PC compressor is on
    }
    else { // compressor is off (i.e. LEDs not flashing)
      io_out(TXD, "0", 1);      // output to PC compressor is off
    }

// output fan on/off
    if ( fan_state == TRUE ) {   // fan is actually on (i.e. LED flashing)
      io_out(TXD, "1", 1);      // output to PC fan is on
    }
    else { // fan is actually on (i.e. LED flashing)
      io_out(TXD, "0", 1);      // output to PC fan is off
    }

// a <CR> ends the packet
    io_out(TXD, "\r", 1);     // <CR>
}
```