Multiple Serial Port Card Support for the IBM Network Station Version 1 Release 3

Version 1.1.5 - September 28, 1999

IBM Network Computer Division

This document is updated regularly. To ensure that you have the latest copy, please check http://www.ibm.com/nc. Click on the `Library` link and then follow the link to `White Papers`.

Multiple serial port card support is being released as a PTF on Network Station Manager for Version 1 Release 3. This is the enabling documentation for that support. The features described in this document pertain only to IBM Network Station Manager Version 1 Release 3 with the enabling PTFs. Those PTFs are:

- OS/400 V3R7 or later w/Group PTF SF99082 applied.
- AIX 4.2.1 with Group PTF IX80941 applied.
- Provided as part of NT 4.0 and NT 4 Windows Terminal Server code.
- VM 2.3 with PTF UA00046.

Customers should not assume that the multiple serial port capability detailed in this document via PCMCIA cards will necessarily persist as future models of the IBM Network Station are announced. The PCMCIA slot may not be available, and/or IBM may devise a better way to provide this capability such as through additional built-in serial ports and/or universal serial bus.

1. Introduction

The IBM Network Station is shipped with a single serial port. This port can be used to attach serial printers, a touchscreen monitor, or other serial devices. The single serial port has been a limitation for some customers who wish to add additional serial devices to the Network Station. The Multiple Serial Port Card support address this requirement by allowing one to four additional serial ports to be added via a PCMCIA card. Thus a single Network Station can be configured to support a total of five serial ports.

While many types of serial devices, such as MICR readers, pole displays, cash drawers, etc., can be connected to the Network Station serial port, IBM does not provide any device drivers for these devices. Application programming is required to send and receive data to these ports/devices. This is done through a TCP/IP socket interface. Since the interfaces use TCP/IP, the application to access the ports can run on any server in the network. The application can also be written in Java to run natively on the Network Station. The programming interface to these additional ports is identical to that used to address the built-in serial port.

References

- *IBM Network Station Manager Installation and Use*.
  This document may be ordered in the US as publication SC41-0664 or accessed on the web at http://www.ibm.com/nc/pubs.

2. PCMCIA Multiple Serial Port Cards

Multiple Serial Card Procurement

The multiple serial port card is a third-party offering. IBM does not provide these cards. We have tested multiple serial port cards from Quatech (http://www.quatech.com). Quatech's part numbers for the various cards they can supply are shown in the following table.

Number of Serial Ports	Part Number
1	SSP-100
2	DSP-100
4	QSP-100

For pricing and availability, please contact Quatech at sales@quatech.com.

Network Station Series 1000 Support

Network Station Series 100 and 300 systems are supplied with built-in PCMCIA adapters. For Series 1000 systems, this adapter is an optional feature on all but the very early ethernet models. The part number for the PCMCIA Adapter on the Series 1000 is 07L8336 and can be ordered in the GEMS ordering system in the US and Canada and the UPOS ordering system in EMEA. Contact your IBM Network Station Sales representative or Business Partner to order these adapters.

2. Enabling the Network Station Multiple Serial Port Support

Enabling the IBM Network Station to support a multiple serial port card is fairly straightforward. While there is no built-in Network Station Manager support for this card, the procedure is very easy and is documented below. Since it involves changes to configuration files, it should only be done by a system administrator who is knowledgeable about Network Station setup.

Prerequisites

The prerequisite for this support is the IBM Network Station Release 3 code with the latest PTF's applied.

Hardware Setup

To set up the PCMCIA multiple serial port card, merely power off the Network Station and insert the card into the PCMCIA slot. Connect the appropriate devices to the native port and the pigtails on the Quatech card.

Software Setup

Software setup requires that changes be made to the configuration files. Add the following lines to to the configuration file **defaults.dft**. This file can be found in the **configs** directory. The initial line (port 1) is required for the native serial port. Add lines for ports 2-5 depending on how many ports are on the Quatech card.

```
set serial-interfaces-table = {
  { 1 <mode> <mode> <baud-rate> <data-bits> <stop-bits> <parity> <handshake> none }
  { 2 <mode> <mode> <baud-rate> <data-bits> <stop-bits> <parity> <handshake> none }
  { 3 <mode> <mode> <baud-rate> <data-bits> <stop-bits> <parity> <handshake> none }
}
```

The following values should be used for the variables in the serial interfaces table, depending on the characteristics of the serial device being attached:

Parameter	Possible Values
mode	`printer`, `serial-daemon`, `terminal`, `input-device` The choices `printer` and `serial-daemon` are equivalent. `input-device` is used when attaching a touchscreen.
baud-rate	`50, 75, 110, 134.5, 150, 200, 300, 600, 1050, 1200, 1800, 2000, 2400, 4800, 7200, 9600, 14400, 19200, 38400, 57600, 76800, and 115200`
data-bits	`7, 8`
stop-bits	`1, 2`
parity	`none, even, odd, space, mark`
handshake	`none, xon/xoff, dtr/dsr, rts/cts`

For example, the default settings for a printer on the native serial port would look as follows:

```
set serial-interfaces-table = {
     { 1 printer printer 9600 8 1 none dtr/dsr none }
}
```

Appendix A. Programming Serial Port Devices

IBM does not supply drivers for any devices attached to the native or Quatech serial ports. Devices such as printers can be usually be attached without additional programming support provided that no special interfaces are required. Non-printer devices usually require support from the controlling application program.

TCP/IP sockets provides a way to interface with devices attached to the serial port(s). Each serial port has a unique socket port number that can be used to address requests to that port. The default values for the ports are as follows:

Serial Port Number	TCP/IP Port Number
1	87
2	5962
3	5963
4	5966
5	5967

Programming Example

The following C code illustrates the steps necessary to use serial port devices. This code is run from a server and drives devices attached to the Network Station. Similar Java code could be written to execute directly on the Network Station. This code is part of a larger program and is not intended to be complete. It is presented to illustrate initializing a socket to a serial port (the native port 87 in this example), sending/receiving data from devices such as printers and cash drawers, and closing the connection.

```
/*********************************************************************/
/*
* Module Name: Serial
*
* Description:
*  Demonstrate how to write and read to a devices attached to
*  a Network Station.
*
*/
/*********************************************************************/

/** INCLUDES *********************************************************/
#include <sys/types.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <sys/errno.h>

/** DEFINES ************************************************************/
#define DEFAULT_HOST      "germ"
#define SERIAL_PORT       87
#define CMD_CUT_PAPER     0x1b69
#define BUFFER_BYTES      80
#define MAX_WRITE         0xFFFF
#define CHECK_STATUS      10000    /* Check printer status after this */
/* many bytes have been sent       */
```

```
#define DEFAULT_PORT      SERIAL_PORT

/* define valid CMDS to printer */
#define CMD_RETURN_HOME                         0x1b3c00
#define CMD_GENERATE_PULSE_FOR_CASH_DRAWER      0x1b70

#define SUCCESS          0
#define FAIL            -1
#define UNKNOWN_ERROR   -1
#define CONNECT_BLOCKED -2
#define OUT_OF_PAPER    -3
#define OFFLINE         -4
#define PRINTER_ERROR   -5

#define NCDESC        0xF0      /* SerialD commands                */
#define GET_STATUS    0xF1
#define GET_WAITING   0xF7
#define BLOCK_WRITE   0xFB
#define BYTES_WAITING 0xFC
#define STATUS        0xFF

#define SERIAL         'S'
#define EMPTY            0        /* Key value of an empty slot in   */
/* "connections" array.            */
#define OUT_OF_PAPER_BIT 0x01     /* Parallel status bits            */
#define OFFLINE_BIT      0x02
#define ERROR_BIT        0x04

#define DCD_BIT          0x40     /* Serial status bits              */
#define ITEMENTRY        1        /* State machine is in transaction */
#define TENDER           2        /* State machine is in tender type */
#define TOTAL            3        /* State machine is in total trans */
#define RESET            4        /* State machine is resetting      */


/** GLOBALS ***********************************************************/
int     errflg,c;               /* used in parsing parameters   */
int     serial_socket;

struct sockaddr_in server;
typedef struct {                    /*                          */
  int  socketDesc;                 /* Socket desc. - key value*/
  int  resendBytes;                /*                        */
  int  bytesSinceLastStatus;       /*                        */
  char connectionType;             /*                        */
} connectionInfo;                   /*                          */

static connectionInfo connections[1];

char hostname[80];
char print_job_desc[80];

/*                                                              */
/*  Nasty global variables to simplify this job                 */
/*                                                              */
char cardNum[10];
char checkNum[10];
char inString[160];

/* Prototypes */
int WritePrintData();
int writeData();
int readData();
void ResetInString();
void ReadInput();
int ReadPrintData();
int CutPrinterPaper();
int WritePrinterString(char *);
int printSendCmd();
int OpenSerialConnection();
int CloseSerialConnection();
int checkStatus();

/*********************************************************************/
/* Function Name: ResetInString()                                    */
/*                                                                   */
/* Use getline to read a line into global string inString            */
/*********************************************************************/
void ResetInString()
{
  int i;

  for (i = 0; i < 80; i++) {
    inString[i] = 0;
  }
}
/*********************************************************************/
/* Function Name: readInput()                                        */
/*                                                                   */
/* Use getline to read a line into global string inString            */
/*********************************************************************/
readInput()
{
  int i;

  for (i = 0; i < 80; i++) {
    inString[i] = 0;
  }
  fgets(inString, 80, stdin);
}
/*********************************************************************/
/* Function Name: PopCashDrawer()                               */
/*                                                                   */
/* Parameters:                                                       */
/*                                                                   */
/* Returns: rv = SUCCEESS if successful                              */
/*              FAIL if failed                                       */
/*********************************************************************/
int PopCashDrawer()
{
  connectionInfo *connInfoPtr = &connections[0];
  char write_data[100] = { "\0" };

  int slen, tslen, bytes_written, total_bytes_written;
  int write_index = 0;
  int rc;
  short popit = POPCASHD;

  /*************************************************************
  * Copy the open drawer escape sequence to the string write_data
  *************************************************************/
  write_data[0] = 0x1B;
  write_data[1] = 0x70;
  write_data[2] = 0;
  write_data[3] = 40;
  write_data[4] = 250;
  write_data[5] = 0;
  rc = write(serial_socket, &write_data, 8);
  return(SUCCESS);
}
/*********************************************************************/
/* Function Name: CutPrinterPaper()                               */
/*                                                                   */
/* Parameters:                                                       */
/*                                                                   */
/* Returns: rv = SUCCEESS if successful                              */
/*               FAIL if failed                                      */
/*********************************************************************/
int CutPrinterPaper()
{

  unsigned short ucmd = CMD_CUT_PAPER;
  connectionInfo *connInfoPtr = &connections[0];
  char write_data[100] = {
          "\0"    };
  int slen, tslen, bytes_written, total_bytes_written;
  int write_index = 0;
  int rc;

  rc = write(serial_socket, &ucmd, 2);
  printf("wrote CUT message with this many bytes: %d\n", rc);

  /*******************************************************************
  * Copy the cut paper escape sequence to the string write_data
  *******************************************************************/
  write_data[0] = 0x1B;
  write_data[1] = 0x69;
  write_data[2] = 0;
  write_data[3] = 0;
  write_data[4] = 0;
  write_data[5] = 0;
  rc = write(serial_socket, &write_data, 8);

  return(SUCCESS);
}
/*********************************************************************/
/* Function Name: WritePrinterString()                               */
/*                                                                   */
/* Parameters:                                                       */
/*                                                                   */
/* Returns: rv = SUCCEESS if successful                              */
/*               FAIL if failed                                      */
/*********************************************************************/
int WritePrinterString(char *aString)
{
  connectionInfo *connInfoPtr = &connections[0];
  char write_data[100] = { "\0" };

  int slen, tslen, bytes_written, total_bytes_written;
  int write_index = 0;
  int rv;

  /*******************************************************************
  * Write some data to the printer. Make sure all data is written   *
  * Note: printWrite() will make every attempt to send all data     *
  * requested but if it is not able to then resendBytes will be the *
  * number of bytes that were not written.                        *
  *******************************************************************/
  strcpy(write_data, aString);
  tslen = strlen(write_data);
  do {
    rv = WriteSerialData(serial_socket, &write_data[write_index],
                         tslen, &bytes_written);
    write_index = bytes_written;
    tslen =  strlen(&write_data[write_index]);
  } while (connInfoPtr->resendBytes);

  printf("\t%s", write_data);

  return(SUCCESS);
}
/*********************************************************************/
/* Function Name: WritePrinterData()                                */
/*                                                                  */
/* Parameters:                                                      */
/*                                                                  */
/* Returns: rv = SUCCEESS if successful                             */
/*                FAIL if failed                                    */
/*********************************************************************/
int WritePrinterData()
{

  connectionInfo *connInfoPtr = &connections[0];
  char write_data[100] = {
          "\0"    };
  int slen, tslen, bytes_written, total_bytes_written;
  int write_index = 0;
  int rv;

  /*******************************************************************
  * Write some data to the printer. Make sure all data is written   *
  * Note: printWrite() will make every attempt to send all data     *
  * requested but if it is not able to then resendBytes will be the *
  * number of bytes that were not written.                        *
  *******************************************************************/
  strcpy(write_data, "Test write data for My Store, Inc.\n");
  tslen = strlen(write_data);
  do {
    rv = WriteSerialData(serial_socket, &write_data[write_index],
                         tslen, &bytes_written);
    write_index = bytes_written;
    tslen =  strlen(&write_data[write_index]);
  } while (connInfoPtr->resendBytes);

  return(SUCCESS);
}

/*********************************************************************/
/* Function Name: WriteSerialData                                   */
/*                                                                  */
/*********************************************************************/
int WriteSerialData(int sd,            /* socket descriptor            */
char *data,     /* pointer to the data           */
int length,     /* leng of print data            */
int *bytesSent) /* number of bytes written (OUT) */
{

  int count = 0;
  int rv = 0;
  int space;
  connectionInfo *ciP = &connections[0];
  *bytesSent = 0;                            /* Init output value  */

  /* check the printer status before we begin writing */
  if (ciP->bytesSinceLastStatus == 0) {
    rv = checkStatus(sd);
    if (rv != SUCCESS) {
      printf("ERROR: check status failed\n");
      return(rv);
    }
  }

  /* Loop while data to send                                        */
  while (length > 0) {
    if (ciP->resendBytes) {
      space = ciP->resendBytes;
    } else {
      /* Use smaller value of MAX WRITE and data length            */
      if (length < MAX_WRITE)
        space = length;
      else
        space = MAX_WRITE;
    }

    rv = writeData(sd, data, space, &count);   /* Write data   */

    if (rv < SUCCESS) {
      *bytesSent += count;
      ciP->bytesSinceLastStatus += count;
      if (ciP->bytesSinceLastStatus >= CHECK_STATUS) {
        ciP->bytesSinceLastStatus = 0;
      }
      return(rv);
    }

    if (ciP->resendBytes) ciP->resendBytes = 0;
    *bytesSent += space;                     /* Update lengths    */
    data += space;
    length -= space;
    count = 0;
    ciP->bytesSinceLastStatus += space;
    if (ciP->bytesSinceLastStatus >= CHECK_STATUS) {
            ciP->bytesSinceLastStatus = 0;
    }
  }
  return(SUCCESS);
}


/*********************************************************************/
/* Function Name: writeData()                                        */
/*                                                                   */
/* Parameters:                                                       */
/*                                                                   */
/*  1.Socket descriptor returned from printOpen                      */
/*  2.Character pointer to print data                                */
/*  3.Length of print data                                           */
/*  4.Number of bytes sent (output)                                  */
/*                                                                   */
/* Returns: See return codes defined at top of file.                 */
/*                                                                   */
/* The procedure creates the NCD block write command and does the    */
/* actual writing of print data to serialD                           */
/*                                                                   */
/*********************************************************************/

int writeData(int sd, char *data, int len, int *bytesSent)
{
```

```
char cmd[4];
int  count = 0;
int  rc;
int  loop = 15;
fd_set write_sd;
connectionInfo *ciP = &connections[0];

*bytesSent = 0;                             /* Init output value  */
cmd[0] = NCDESC;
cmd[1] = BLOCK_WRITE;
cmd[2] = (char)(len >> 8);
cmd[3] = (char)(len);

#ifdef DEGUG
printf("writeData: %d bytes\n", len);
#endif
if (!ciP->resendBytes) {
  /* Write command to socket                                    */
  while (count < 4) {
    rc = write(sd, cmd + count, 4 - count);  /* Write command   */
    if (rc > 0 ) {
      count += rc;
      loop = 10;
    } else {                      /* Error on write             */
      if (errno == EWOULDBLOCK) {
        if (loop) {
          sleep(1);
          loop--;
          continue;
        }
        return(CONNECT_BLOCKED);
      } else {
        printf("Write Command: Unknown error.\n");
        return(UNKNOWN_ERROR);
      }
    }
  }
}

loop = 15;
count = 0;

/* Write data to socket                                           */
while (count < len) {
  rc = write(sd, data + count, len - count); /* Write data       */
  if (rc > 0 ) {
    count += rc;
    loop = 15;
  } else {                      /* Error on write                */
    if (errno == EWOULDBLOCK) {
      if (loop) {
        sleep(1);
        loop--;
        continue;
      }
      ciP->resendBytes = len - count;
      *bytesSent = count;
      return(CONNECT_BLOCKED);
    }
    ciP->resendBytes = len - count;
    *bytesSent = count;
    return(UNKNOWN_ERROR);
  }
}
```

```
  *bytesSent = count;
  return(SUCCESS);                      /* All data sent                 */
}

/*********************************************************************/
/* Function Name: readData()                                         */
/*                                                                   */
/* Parameters:                                                       */
/*                                                                   */
/*   1.Socket descriptor returned from printOpen                     */
/*   2.Pointer to command to send.                                   */
/*   3.Length of command to send.                                    */
/*   4.Pointer to buffer to receive read in data.                    */
/*   5.Amount of data to read.                                       */
/*                                                                   */
/* Returns: See return codes defined at top of file.                 */
/*                                                                   */
/* This procedure sends a serialD command and reads the returned     */
/* data.  A wait of 10 seconds will be done for the write and read   */
/* to take place.  If they to not happen in that time period, it is  */
/* assumed there is an error.                                        */
/*                                                                   */
/*********************************************************************/
int readData(int sd, char *cmd, int cLen, char *buf, int bLen)
{

  fd_set         read_sd, write_sd;
  int rc;
  int count = 0;
  int loop = 30;

  /* Write command to socket                                        */
  while (count < cLen) {
    rc = write(sd, cmd + count, cLen - count);  /* Write command   */

    if (rc > 0 ) {
            count += rc;
    } else {                    /* Error on write                  */
      if (errno == EWOULDBLOCK) {
        if (loop) {
          sleep(1);
          loop--;
          continue;
        }
        return(CONNECT_BLOCKED);
      } else {
        return(UNKNOWN_ERROR);
      }
    }
  }

  count = 0;
  loop = 30;

  /* Read data                                                      */
  while (count < bLen) {
    rc = read(sd, buf + count, bLen - count);  /* Read data        */
    if (rc > 0 ) {
      count += rc;
    } else {                    /* Error on write                  */
      if (errno == EWOULDBLOCK) {
        if (loop) {
          sleep(5);
          loop--;
          continue;
        }
      }
      return(UNKNOWN_ERROR);
    }
  }
  return(count);
}

/*********************************************************************/
/* Function Name: printSendCmd()                                     */
/*                                                                   */
/*********************************************************************/
int printSendCmd()
{

  int rc = FAIL;
  int count = 0;
  char c = '?';
  short ucmd;
  long lcmd;
  char buf[2];
  int rv;
  int bytes_written;

  do {
    printf("\t----------------------------------------------\n");
    printf("\tPrinter Command Selection Menu\n");
    printf("\t----------------------------------------------\n");
    printf("\tMake a selection from the following menu\n\n");
    printf("\t\n");
    printf("\t1) Open the cash drawer\n");
    printf("\t2) Return print head to home location\n");
    printf("\t3) Cut the paper in the customer receipt station\n");
    printf("\tq) Return to main menu\n");
    printf("\t----------------------------------------------\n");
    printf("\t===>");

    fflush(stdin);
    c = getc(stdin);
    switch (c) {
    case '1' : /* send open cash drawer command to the printer */
      ucmd = CMD_GENERATE_PULSE_FOR_CASH_DRAWER;
      if ((rc = write(serial_socket, &ucmd, 2)) > 0)
        printf("Wrote command request 0x%x\n",ucmd);
      else {
        fprintf(stderr, "ERROR: Write failed; retcode=%d; errno=%d\n", rc, errno);
        return FAIL;
      }
      break;
    case '2' : /* send return print head home command */
      lcmd = CMD_RETURN_HOME;
      if ( (rc = write(serial_socket, &lcmd, 4)) > 0)
        printf("Wrote command request 0x%x\n",lcmd);
      else {
        fprintf(stderr, "ERROR: Write failed; retcode=%d; errno=%d\n", rc, errno);
        return FAIL;
      }
      break;
    case '3' : /* send return print head home command */
      ucmd = CMD_CUT_PAPER;
      if ((rc = write(serial_socket, &ucmd, 2)) > 0)
        printf("Wrote command request 0x%x\n",ucmd);
      else {
        fprintf(stderr, "ERROR: Write failed; retcode=%d; errno=%d\n", rc, errno);
        return FAIL;
      }
    case 'q' :
      rv = SUCCESS;
      break;
    default :
      printf("ERROR: Invalid entry, try again\n");
    }
  } while ( c != 'q');

  if (rv == FAIL) {
    printf("ERROR: runMenu failed\n");
    exit(FAIL);
  }
}

/*********************************************************************/
/* Function Name: OpenSerialConnection()                                     */
/*                                                                   */
/* Parameters: char * hostname                                       */
/*                                                                   */
/* Returns: valid file descriptor of socket connection               */
/*                                                                   */
/*********************************************************************/
int OpenSerialConnection(char *hostname)
{

  int sd, retcode;
  struct hostent *host;
  connectionInfo *ciP = &connections[0]; /* XXX only managing 1 connection */
  int on = 1;

  #ifdef DEBUG
  printf("printfOpenSerialSocket Enter -->\n");
  #endif

  /* CREATE A SOCKET */
  if ((sd = socket(AF_INET, SOCK_STREAM, 0)) >= 0) {
    bzero((char *)&server, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_port = DEFAULT_PORT;

    /* FIRST ASSUME THAT THE HOSTNAME IS A DOTTED IP ADDRESS */
    server.sin_addr.s_addr = inet_addr(hostname);
    if (server.sin_addr.s_addr == -1) {
      /* MUST BE A DNS NAME INSTEAD */
      if ((host = gethostbyname(hostname)) == NULL) {
        (void)fprintf(stderr, "ERROR: gethostbyname failed: h_errno=%d.\n",
                      h_errno);
        return FAIL;
      }
      bcopy(host->h_addr, (char *)&server.sin_addr, host->h_length);
    }

    /* OPEN A SOCKET AND SET OPTIONS */
    if ((retcode = connect(sd, (struct sockaddr *)&server,
                           sizeof(struct sockaddr_in))) == 0) {
      if ((retcode = setsockopt(sd, IPPROTO_TCP, TCP_NODELAY,
                                (char *)&on, sizeof(on))) == 0) {
        if ((retcode = setsockopt(sd, SOL_SOCKET, SO_KEEPALIVE,
                                  (char *)&on, sizeof(on))) != 0) {
          (void)fprintf(stderr, "ERROR: SO_KEEPALIVE failed: errno=%d.\n", errno);
        }
      }
      else
        (void)fprintf(stderr, "ERROR: TCP_NODELAY failed: errno=%d.\n", errno);
    }
    else
      (void)fprintf(stderr, "ERROR: connect failed: errno=%d.\n", errno);
  }
  else {
    (void)fprintf(stderr, "ERROR: socket failed: errno=%d.\n", errno);
    retcode = errno;
  }

  if (retcode != 0) {
    printf("ERROR: Socket connection FAILED to %s at port %d.\n",
           hostname, DEFAULT_PORT);
    return FAIL;
  }

  /* Initialize connection information                    */
  ciP->socketDesc = sd;
  ciP->resendBytes = 0;
  ciP->bytesSinceLastStatus = 0;
  ciP->connectionType = SERIAL;

  /* Check the printer status.  If not OK, return an error. */
  if ((retcode = checkStatus(sd)) != SUCCESS) {
    ciP->socketDesc = EMPTY;
    (void)close(sd);
    return(FAIL);
  }

  /* Socket created successfully */
  printf("Socket connection established to %s at port %d.\n",
         hostname, DEFAULT_PORT);
  return sd;
}

/*********************************************************************/
/* Function Name: CloseSerialConnection()                            */
/*                                                                   */
/* Parameters:                                                       */
/*                                                                   */
/*   1.Socket descriptor returned from printOpen                     */
/*                                                                   */
/* Returns: 0                                                        */
/*                                                                   */
/* The API CloseSerialConnection is used to close the connection to  */
/* the printer  at the end of a print job.  Check that all data has  */
/* made it to the printer before closing.                            */
/*                                                                   */
/*********************************************************************/
int CloseSerialConnection(int sd)           /* Socket descriptor           */
{

  char cmd[2];
  char buf[4];
  unsigned short int bytesWaiting;
  int rc;

  cmd[0] = NCDESC;
  cmd[1] = GET_WAITING;

  /* Loop until all bytes have been sent to the printer             */
  do {
    rc = readData(sd, cmd, 2, buf, 4);
    if (rc < SUCCESS) {
      if (buf[1] == (char)BYTES_WAITING) {
        bytesWaiting = *((short int *)(buf + 2));
      } else {
        rc = UNKNOWN_ERROR;
        break;
      }
    } else {
      rc = UNKNOWN_ERROR;
      break;
    }
  } while (bytesWaiting > 0);

  close(sd);
  return(SUCCESS);
}

/***************************************************************@A1A**/
/* Function Name: checkStatus()                                      */
/*                                                                   */
/* Parameters:                                                       */
/*                                                                   */
/*   1.Socket descriptor returned from printOpen                     */
/*                                                                   */
/* This procedure sends the return status command, reads the         */
/* returned bytes and checks for an error condition.                 */
/*                                                                   */
/***************************************************************@A1A**/
int checkStatus(int sd)
{
   char cmd[2];
   char buf[4];
   int  rc = 0;
   int connectionType = SERIAL;

   cmd[0] = NCDESC;
   cmd[1] = GET_STATUS;

   rc = readData(sd, cmd, 2, buf, 4);
   if (rc < SUCCESS) return(rc);

   if (buf[1] == (char)STATUS) { /* Check it is status cmd  */
     if (buf[2] & DCD_BIT) {  /* If the DCD bit is on,   */
       return(OFFLINE);      /* there is an error.      */
     }
     return(SUCCESS);

   } else {
     return(UNKNOWN_ERROR);
   }
}
```

*** End of Document ***