CIMETRICS

NBS-1x/NBS-20/NBS-4x DEVICE DRIVER FOR WINDOWS 2000

INSTALLATION AND PROGRAMMING MANUAL

Document Version: November 28, 2000

CIMETRICS INC.
55 TEMPLE PLACE
BOSTON, MASSACHUSETTS 02111-1300
U.S.A.

TELEPHONE: (617) 350-7550
FAX: (617) 350-7552
E-MAIL: info@cimetrics.com
WEB: http://www.cimetrics.com

1 Introduction

This document describes Cimetrics' NBS-1x/NBS-20/NBS-4x Device Driver for Microsoft Windows 2000. The Device Driver and your Cimetrics NBS-10, NBS-11, NBS-12, NBS-20, NBS-41, or NBS-42 enable you to develop and deploy PC application programs running under Windows 2000 which do RS-485 and RS-422 asynchronous serial communication. Applications make use of the Device Driver by calling several Win32 API functions. A wide range of baud rates and character formats are supported, including μLAN ("9-bit") for communication with popular microcontrollers.

The NBS-10 is an RS-485/422 card based on Intel's 82510 UART. Unlike the UARTs commonly found in PC serial ports, the 82510 can generate an interrupt when its transmitter's shift register becomes empty, allowing the RS-485 line driver to be turned off in a timely manner. This capability is particularly valuable when a multitasking operating system such as Windows 2000 is used. The NBS-11, NBS-12, NBS-20, NBS-41 and NBS-42 have similar capabilities.

Here is a short summary of the capabilities and requirements of the Device Driver:

Computer Type:
a "PC" which satisfies all requirements for running Microsoft Windows 2000

Operating System:
Microsoft Windows 2000

RS-485/422 Interface:
one or more Cimetrics NBS-10, NBS-11, or NBS-12 cards (“NBS-1x”), one or more NBS-20 cards, and/or one or more NBS-41 or NBS-42 cards (“NBS-4x”)

IRQ Lines:
one unused IRQ line (3-7) per NBS-1x card, one unused IRQ line per NBS-20 card , one PCI IRQ line for NBS-4x cards

Maximum Speed:
288,000 bps with a standard NBS-1x/NBS-20/NBS-4x (18.432 MHz oscillator)

Character Format:
asynchronous, 5/6/7/8/9 data bits, even/odd/mark/space/no parity, 1/1.5/2 stop bits

Suggested Programming Tools:
Microsoft Visual C++, Microsoft Visual Basic, or other 32-bit compiler for developing Windows 2000 applications; users of other programming tools can contact Cimetrics to discuss compatibility

1.1 Product License

The Device Driver has been licensed to you or your company for use with Cimetrics NBS cards. It is not shareware or "free" software. Please read the license agreement before you use this product.

1.2 Technical Support

Limited free technical support is available. Application engineering assistance is available on a paid consulting basis. To obtain technical support, please contact Cimetrics as follows:

E-mail: support@cimetrics.com
web: http://www.cimetrics.com
telephone: +1 (617)350-7550
fax: +1 (617)350-7552

Customers in areas of the world served by a Cimetrics authorized distributor (currently Japan, Germany, Switzerland, and Austria) should contact their local distributor for technical support.

1.3 Backward Compatibility

This product replaces the NBS-1x/NBS-4x Device Driver for Windows NT. Almost all applications written for the NBS-1x/NBS-4x Device Driver for Windows NT will run without modification with this product.

2 Setup

The setup process involves configuring and installing your NBS-1x/NBS-20/NBS-4x card, then installing the Device Driver software. Make notes as you configure the card; some of that information may be necessary when you install the Device Driver. For more complete information on card configuration and cabling, see the NBS card's user's manual.

2.1 NBS-10/NBS-11/NBS-12 Hardware Preparation

The NBS-1x cards are very flexible, but it is very easy to configure a card in such a way that it will not function properly. Here are some of the configuration choices that are relevant to the device driver:

1. The eight consecutive I/O ports used by the card must not conflict with any other peripheral devices in the PC. With Windows 2000, it is generally best *not* to set the base I/O port address to COM1 (0x3F8), COM2 (0x2F8), COM3 (0x3E8), or COM4 (0x2E8), to avoid conflict between the Device Driver and Windows 2000's standard serial device driver. We recommend address 0x300, but you should first check the address of any installed network interface card and other peripherals. (See section 4.4 for further details.)

2. The IRQ line must not be shared with any other device in the PC whose device driver could claim that IRQ line during a Windows 2000 session. For example, if you have a serial mouse connected to COM1 (COM2), the card should not be configured to use IRQ4 (IRQ3). We recommend IRQ5, but you should first verify that no other device in your PC is using that IRQ line. (Audio and Ethernet interfaces sometimes use IRQ5.) IRQ7 is normally reserved for use by the LPT1 device, but often it is not being used. (See section 4.4 for more details.)

3. On the NBS-10, DIP switch bank SW2 must be set appropriately for your application. For most RS-485 (two-wire half-duplex) applications, the following settings should work:

    SW2.1 ON (half-duplex)
    SW2.2 OFF
    SW2.3 OFF
    SW2.4 OFF (the RTS line will be used for transmit/receive switching)

For most RS-422 (four-wire full-duplex) applications, SW 2.1 should be OFF.

2.2 Device Driver Installation and Configuration

2.2.1 NBS-10/11/12 Cards

In order to complete the installation, you will need to know the following configuration information for each NBS-1x that you have installed:

- base I/O port address (NBS-10: controlled by SW1)
- IRQ number (3-7) (NBS-10: determined by J5)
- crystal oscillator frequency (typically 18.432 MHz); the crystal oscillator is a rectangular metal component near the end of the NBS-10 opposite the metal bracket, and its frequency is printed on top of the component

- duplex (half or full) (NBS-10: set by SW2.1) (NBS-11/NBS-12: set to "half")
- transmit/receive direction control line (RTS or DTR) (NBS-10: controlled by SW2.4)

To install the Device Driver for a NBS-1x card, do the following steps in order:
1. Initiate Windows 2000's Add/Remove Hardware application, which is found in the Control Panel.
2. Select "Add/Troubleshoot a device", then from the list select ”Add a new device”.
3. Select “No, I want to select the hardware from a list”.
4. Select type of hardware: "other devices" if this is a first time installation, or "Cimetrics Nine-Bit Serial Adapters" if it is already present in the list of Device classes.
5. Click on the "Have Disk" button.
6. Insert the Device Driver diskette, make any necessary changes to the path so that Windows 2000 can find the NBSXNT.INF file, and click on the "OK" button.
7. Select model "NBS-10".
8. Enter I/O port and IRQ settings for the NBS-10. If there are conflicts with existing devices, you should select a different setting then change the configuration of the NBS-10 card accordingly.
9. "Finish" Device Driver installation.
10. Do not select “Restart”. First activate the Device Manager (right-click My Computer on the desktop and click Properties. Click the Hardware tab, and then click Device Manager).
11. Select "Cimetrics Nine-Bit Serial Adapters -> “NBS-10”. Select "Properties" from the Action menu.
12. Choose "Settings". TType in a device name which can be recognized by your application program. We recommend "NBS10-1" for the first NBS-10 card, "NBS10-2" for the second card, etc. Verify that the Crystal Frequency and Transmit Toggle settings are correct, and make appropriate changes if not.
13. Restart Windows 2000.
14. Check for "Cimetrics Nine-Bit Serial Adapters" in the Device Manager. If there is any problem, there will be a redcross or exclamation symbol ("!") on the entry "NBS-10", in which case you should choose this entry and select "Properties"--there will be an error code in the "Device Status" area, which you should write down before calling Cimetrics for technical support.

About some parameters:

- "Duplex" is either "half" or "full". In most cases, this should be set the same as SW2.1 on the NBS-10 ("half" for NBS-11 and NBS-12 cards). The duplex configuration of the card controls whether a single pair of wires will be used for communication, or whether two pairs will be used (one for reception and one for transmission). The duplex setting of the Device Driver only affects the internal behavior of the Device Driver: in full-duplex mode, read (receive) and write (transmit) requests are processed independently (and can overlap), whereas in half-duplex mode, read and write requests are processed sequentially in the order in which they were received.
- "Transmit Toggle" should be set to correspond to SW2.4 on the NBS-10, if you want the Device Driver to automatically set that bit (either RTS or DTR) to 1 during transmission and 0 during reception, for purposes of enabling or disabling the line driver on the NBS-10. This will be the case if you have a multidrop wiring scheme, as is true in two-wire RS-485 applications. Note that manual control of the RTS and DTR bits is also possible (see section 3.3.2). For 4-wire applications (NBS-10 only), this parameter should usually be set to "None".

2.2.2 NBS-20/41/42 Cards

The first time you insert a NBS-20 or NBS-4x card into your PC, Windows 2000 should inform you that it has found new hardware. You will see a dialog box giving you the opportunity to load a device driver supplied by the manufacturer among a few options. By selecting that option, inserting the Device Driver diskette into your PC's floppy drive, and following the instructions, the Device Driver will be installed.

In order to successfully configure the Device Driver for NBS-20/NBS-4x cards you will need to know the following about the configuration of each such card already installed in your PC:

- Crystal oscillator frequency (18.432 MHz in a standard NBS-40, 16 MHz or 12 MHz in other versions sold by Cimetrics); on NBS-4x cards the crystal oscillator is a metal component that is close to the right edge of the circuit board, and the frequency should be printed on the top of the oscillator.

- Transmit Toggle ("RTS(sw)", "RTS(hw)" or "None"). Transmit Toggle should be set to "RTS(hw)" in most cases, which programs the UART to automatically set and clear the RTS bit in such a way as to turn on the RS-485 driver during transmissions and turn it off during reception. For NBS-20 cards this setting available only with a special RS-485 cable. Select "RTS(sw)" if you want the Device Driver to automatically control the RS-485 transceiver using the RTS line. Note that manual control of the RTS bit is also possible, in which case the appropriate setting is "None".

If the Device Driver was previously installed for any NBS card (NBS-1x , NBS-20 or another NBS-4x card), PnP manager will register a new card automatically. You must only type in a device name that can be recognized by your application program. Otherwise as soon as you insert an NBS-20/NBS-4x card the "Found New Hardware Wizard" will be executed.

Do the following steps in order:

1. Select "Display a list of the known drivers for this device so that I can choose a specific driver"
2. Select Hardware type: "Other devices"
3. Click the "Have disk…" button
4. Insert the Device Driver diskette, make any necessary changes to the path so that Windows 2000 can find the NBSXNT.INF file, and click on the "OK" button.
5. Select model "NBS-40", “NBS-20” or “NBS-20B” as appropriate.
6. "Finish" Device Driver installation. Do not restart Windows 2000 at this time.
7. Activate the Device Manager. (Right click on the "My Computer" icon on the Windows 2000 desktop, select "properties", select "Hardware", then click on the "Device Manager" button.)
8. Select "Cimetrics Nine-Bit Serial Adapters -> “NBS-XX”. Select "Properties". Choose "Settings". Type in a device name that can be recognized by your application program. We recommend "NBSX-1" for the first card, "NBSX-2" for the second, etc. Verify that the Crystal Frequency and Transmit Toggle settings are correct, and make appropriate changes if not.
9. Restart Windows 2000.

You can restart the PC and check for any System Log error messages generated by "NBSXNT" using the Event Viewer (an Administrative Tool).

Configuration changes may not go into effect until Windows 2000 is restarted.

3 Programming

Application programs use of the NBS-1x/NBS-20/NBS-4x Device Driver by calling the following Win32 API functions:

- CreateFile() to get control of an NBS card
- DeviceIoControl() to set communication parameters and control the behavior of the Device Driver
- WriteFile() to transmit
- ReadFile() to receive
- FlushFileBuffers()
- CloseHandle() to relinquish control of an NBS card

This chapter discusses how your application programs can make use of the Device Driver. Programming using the Device Driver is conceptually very similar to programming using the serial device driver provided with Windows 2000. Note that all C and C++ program modules which manipulate the Device Driver should "#include" files WINDOWS.H (which comes with your compiler or SDK) and NTDNBS10.H (found on the Device Driver diskette). We suggest that you look at file NTDNBS10.H in order to understand the structures that will be used by your application program to communicate with the Device Driver. The Device Driver diskette also contains a simple C-language program that incorporates many of the short code examples found in this chapter.

3.1 Initialization

Before your program can use the Device Driver for communication, initialization is required. There are five mandatory initialization steps:

1. Open the device
2. Set the line parameters
3. Set the baud rate
4. Set timeout periods for transmit and receive operations
5. Set flow control and handshake options

Opening the device requires that Win32 API function CreateFile() be called; the other steps each require a call to function DeviceIoControl(). Note that the Device Driver must be started before it can be used. The installation program configures the Device Driver to start automatically when Windows NT boots, but alternatively it can be started manually by your application program (see section 3.1.6).

```
#include <windows.h>
#include <stdio.h>            // for printf(), puts()
#include "ntdnbs10.h"

BOOL InitializeNBS10(NBS10_BAUD_RATE baudRate)
/* returns TRUE if successful, otherwise returns FALSE */
{
       if (!nbs10StartDeviceDriver()) return FALSE;
       if (!nbs10Open()) return FALSE;
       if (!nbs10SetLineParameters()) return FALSE;
```

```
        if (!nbs10SetBaudRate(baudRate)) return FALSE;
        if (!nbs10SetTimeouts()) return FALSE;
        return TRUE;
}
```

When your program is done using the NBS card, the device should be closed.

3.1.1 Opening the device

The first thing your program must do in order to use an NBS card is to open a device, which creates a handle your program will use for all future Device Driver operations. Note that the device name is always preceded by "\\\\.\\" in C-language programs, or "\\.\" if your program is written in a language in which the backslash character is not an escape character.

```
HANDLE nbs10;                // this handle will be used for all Device Driver operations

BOOL nbs10Open()
/* returns TRUE if successful, otherwise returns FALSE */
{
        extern HANDLE nbs10;

        nbs10 = CreateFile(
                        "\\\\.\\NBS10-1",      // from the device name assigned to the NBS card
                        GENERIC_READ | GENERIC_WRITE,
                        0,                              // exclusive access
                        NULL,                           // no security attributes
                        OPEN_EXISTING,                  // the device should already exist!
                        FILE_ATTRIBUTE_NORMAL,          // file attributes and flags
                                                // to allow overlapped I/O, substitute:
                                                // FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED
                        NULL                            // no extended file attributes
                        );
        if (nbs10 == INVALID_HANDLE_VALUE) {
                printf( "CreateFile() failed trying to open the NBS-10, error code %d.\n",
                        GetLastError()
                        );
                return FALSE;
        }
        return TRUE;
}
```

3.1.2 Setting the serial line parameters

The Device Driver can be configured for a very wide range of line parameters using I/O control function IOCTL_NBS10_SET_LINE_CONTROL: 5 to 9 data bits per character, all common parity settings, and 1, 1.5, or 2 stop bits per character. Note that 1.5 stop bits is supported only with 5 data bits. The parameters must first be stored in a structure of type NBS10_LINE_CONTROL. In the following example, the Device Driver is configured for 8 data bits per character, no parity, and 1 stop bit. In a similar manner, it is possible to query the current line control settings using I/O control function IOCTL_NBS10_GET_LINE_CONTROL.

```
BOOL nbs10SetLineParameters()
/* returns TRUE if successful, otherwise returns FALSE */
{
        extern HANDLE nbs10;
        DWORD returned;
        NBS10_LINE_CONTROL nbs10LineControl;
```

```
        nbs10LineControl.StopBits = STOP_BIT_1;      // STOP_BIT_1, STOP_BITS_1_5, or STOP_BITS_2
        nbs10LineControl.Parity = NO_PARITY;         // NO_PARITY, EVEN_PARITY, ODD_PARITY,
                                                     // MARK_PARITY, or SPACE_PARITY
        nbs10LineControl.WordLength = 8;     // data bits per character (5, 6, 7, 8, or 9)

        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_SET_LINE_CONTROL,        // I/O control code
                        &nbs10LineControl,                   // contains the line parameters
                        sizeof(nbs10LineControl),
                        NULL,
                        0,
                        &returned,
                        NULL                                 // not overlapped
                        )) {
                printf( "DeviceIoControl (NBS10_SET_LINE_CONTROL) failed, error code %d.\n",
                        GetLastError()
                        );
                return FALSE;
        }
        return TRUE;
}
```

3.1.3 Setting the communication speed (baud rate)

A standard NBS card has a crystal frequency of 18.432 MHz, and can operate at the following popular speeds: 288 Kbps, 115.2 Kbps, 57.6 Kbps, 38.4 Kbps, 19.2 Kbps, 9600 bps, 2400 bps, 1200 bps, 300 bps, and 110 bps. NBS cards can also run at many other baud rates, as specified by the following mathematical expression:

$$\frac{nbs10CrystalFrequencyInHertz}{32\boxed{n}} \qquad (n = 2, 3, 4, ..., 65535)$$

If you request a speed that the NBS card cannot precisely match, the Device Driver will configure the card for the closest possible speed. However, you should be aware that if you have UARTs running at significantly different speed which are trying to communicate with each other, problems can occur. The amount of baud rate discrepancy that UARTs can tolerate is limited; depending on the UART and character length, discrepancies of as little as 2% can cause problems. If the discrepancy is too great, you have the option of replacing your NBS card's crystal oscillator with a different value to obtain a better match.

Setting the speed is done using I/O control function IOCTL_NBS10_SET_BAUD_RATE, as shown in the following example. In a similar manner, it is possible to query the current communication speed using I/O control function IOCTL_NBS10_GET_BAUD_RATE.

```
BOOL nbs10SetBaudRate(NBS10_BAUD_RATE baudRate)
/* returns TRUE if successful, otherwise returns FALSE */
{
        extern HANDLE nbs10;
        NBS10_BAUD_RATE actualBaudRate;
        DWORD returned;

        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_SET_BAUD_RATE,    // I/O control code
```

```
                        &baudRate,                      // your desired baud rate
                        sizeof(baudRate),
                        &actualBaudRate,        // Device Driver returns the actual baud rate
                        sizeof(actualBaudRate),
                        &returned,
                        NULL                            // not overlapped
                        )) {
                printf( "DeviceIoControl (NBS10_SET_BAUD_RATE) failed, error code %d.\n",
                        GetLastError()
                        );
                return FALSE;
        }
        printf( "The actual baud rate will be %u.\n", actualBaudRate );
        return TRUE;
}
```

3.1.4 Setting timeouts

The maximum duration of receive (read) and transmit (write) operations can be controlled by the programmer. For example, when function ReadFile() is called to receive a transmission from another computer, you will generally want the read operation to abort if the expected transmission is not received within some predetermined period of time. The Device Driver supports an I/O control function, IOCTL_NBS10_SET_TIMEOUTS, which allows the program to specify timeout parameters. The parameters are stored in a structure of type NBS10_TIMEOUTS, which is equivalent to Win32 structure COMMTIMEOUTS. In a similar manner, it is possible to query the current timeout settings using I/O control function IOCTL_NBS10_GET_TIMEOUTS.

Read timeouts are particularly important, so some explanation is in order. There are three timeout parameters for read operations: CONSTANT, MULTIPLIER, and INTERVAL. A read operation of N characters will be completed in not more than CONSTANT + (N * MULTIPLIER) milliseconds, unless CONSTANT and MULTIPLIER are both zero. If INTERVAL is greater than zero, then a timeout will occur if the time between received characters is greater than INTERVAL milliseconds. If a timeout occurs, any characters received up to that point will be returned.

There are several special cases, one of which is of particular interest. If CONSTANT and MULTIPLIER are both zero, and if INTERVAL is equal to MAXDWORD, then the read operation will complete immediately with whatever characters are currently in the Device Driver's internal receive buffer (perhaps none); this case is illustrated in the code example later in this section. For other special cases please refer to the description of structure COMMTIMEOUTS in the Win32 SDK documentation. Note that appropriate values of these timeout parameters will depend on the baud rate.

Write timeouts are most useful when some sort of low-level flow control is being used (for example, CTS handshaking). The two write timeout parameters are CONSTANT and MULTIPLIER. To disable write timeouts, set both parameters to zero.

```
BOOL nbs10SetTimeouts()
/* returns TRUE if successful, otherwise returns FALSE */
{
        extern HANDLE nbs10;
        NBS10_TIMEOUTS rwTimeouts;          // like Win32 structure COMMTIMEOUTS
        DWORD returned;
```

```
        // read: return immediately with the characters that have already been received, if any
        rwTimeouts.ReadIntervalTimeout = MAXDWORD;
        rwTimeouts.ReadTotalTimeoutConstant = 0;
        rwTimeouts.ReadTotalTimeoutMultiplier = 0;
        // write: timeouts are not used
        rwTimeouts.WriteTotalTimeoutConstant = 0;
        rwTimeouts.WriteTotalTimeoutMultiplier = 0;

        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_SET_TIMEOUTS,     // I/O control code
                        &rwTimeouts,                  // contains the timeout parameters
                        sizeof(rwTimeouts),
                        NULL,
                        0,
                        &returned,
                        NULL                          // not overlapped
                        )) {
                printf( "DeviceIoControl (NBS10_SET_TIMEOUTS) failed, error code %d.\n",
                        GetLastError()
                        );
                return FALSE;
        }
        return TRUE;
}
```

3.1.5 Closing the device driver

When your program is done using the Device Driver, it should call CloseHandle():

```
VOID nbs10Close()
{
        extern HANDLE nbs10;

        CloseHandle(nbs10);
}
```

3.2 Transmission and Reception

To transmit, your program should call Win32 API function WriteFile(). Reception is accomplished using Win32 API function ReadFile(). Timeouts are also relevant to transmission and reception (see section 3.1.4).

3.2.1 Communication using 5-8 data bit character formats

Transmission and reception are fairly straightforward for character formats with no more than 8 data bits per character, as illustrated in the following example.

```
BOOL nbs10TransmitString(LPVOID str, DWORD strLen)
// Transmit a character string of length strLen.
// 5, 6, 7, and 8 data bit character formats are supported.
// Returns FALSE if not successful, otherwise returns TRUE.
{
        extern HANDLE nbs10;
```

```
        DWORD bytesWritten;

        if (WriteFile( nbs10,
                       str,                          // transmit this string
                       strLen,                       // length of the string
                       &bytesWritten,                // actual number of characters transmitted
                       NULL                          // not overlapped
                       )) {
                if (bytesWritten == strLen) return TRUE;
                printf("WriteFile error occurred in nbs10TransmitString: "
                       "%u bytes transmitted out of %u.\n",
                       bytesWritten, strLen);
        } else {
                printf( "WriteFile failed, error code %d.\n", GetLastError());
        }
        return FALSE;
}

BOOL nbs10ReceiveString(LPVOID str, DWORD strLen, LPDWORD bytesRcvd)
// Receive a character string of length not to exceed strLen.
// 5, 6, 7, and 8 data bit character formats are supported.
// The actual number of characters received within the timeout period is written to bytesRcvd.
// Returns FALSE if an error occurred, otherwise returns TRUE.
{
        extern HANDLE nbs10;

        if (ReadFile(  nbs10,
                       str,                          // received characters are written here
                       strLen,                       // maximum number of characters to receive
                       bytesRcvd,                    // actual number of characters received
                       NULL                          // not overlapped
                       )) {
                return TRUE;
        } else {
                printf( "ReadFile failed, error code %d.\n", GetLastError());
                return FALSE;
        }
}
```

3.2.2 Communication using a 9 data bit character format

Unlike Windows 2000's serial device driver, the NBS-10 Device Driver can be configured for a character format of 9 data bits. This allows communication on µLAN networks, which specify 9 data bits per character, no parity, and 1 stop bit. Because each character has more than 8 data bits, one byte is not adequate to represent it, so two bytes are used (low-order byte first). As with 5-8 bit character formats, Win32 API function ReadFile() is used for reception and WriteFile() is used for transmission.

The following code example consists of functions for transmitting and receiving a single 9-bit character. Of course, more than one character can be transmitted or received at a time, in which case the size of the buffer in bytes should be an even number equal to two times the number of 9-bit characters to be transmitted or received.

```
BOOL nbs10TransmitNineBitChar(WORD c)
// Transmit a single 9-bit character.
// Returns FALSE if not successful, otherwise returns TRUE.
{
```

```
        extern HANDLE nbs10;
        DWORD bytesWritten;
        UCHAR buffer[2];                        // large enough for one 9-bit character

        buffer[0] = c & 0x00FF;                 // low-order byte first
        buffer[1] = c >> 8;                     // then high-order byte
        if (WriteFile( nbs10,
                        buffer,                 // the character to transmit
                        2,                      // the length of the buffer
                        &bytesWritten,          // the number of bytes transmitted is written here
                        NULL                    // not overlapped
                        )) {
                if (bytesWritten == 2) return TRUE;
                printf("WriteFile error occurred in nbs10TransmitNineBitChar: %u bytes sent.\n",
                        bytesWritten);
                return FALSE;
        } else {
                printf( "WriteFile failed, error code %d.\n", GetLastError());
                return FALSE;
        }
}

INT nbs10ReceiveNineBitChar()
// Receive one 9-bit character.
// returns:     -2 if a low-level error occurred,
//              -1 if nothing was received within the timeout period,
//              or the unsigned 9-bit character received (0x0000-0x01FF)
{
        extern HANDLE nbs10;
        DWORD bytesRead;
        UCHAR buffer[2];                        // large enough for one 9-bit character

        if (ReadFile(  nbs10,
                        buffer,                 // the received data (if any) is written here
                        2,                      // two bytes allows reception of one 9-bit character
                        &bytesRead,             // the number of bytes read is written here
                        NULL                    // not overlapped
                        )) {
                switch (bytesRead) {
                case 0:
                        return -1;              // nothing received within the read timeout period
                case 2:
                        return (buffer[0] | (buffer[1] << 8));       // one 9-bit char received
                }
                printf("ReadFile error occurred in nbs10ReceiveNineBitChar: %u bytes read.\n",
                        bytesRead);
                return -2;                      // this error indicates a bug somewhere
        } else {
                printf( "ReadFile failed, error code %d.\n",  GetLastError());
                return -2;                      // an error
        }
}
```

3.2.3 Changing the size of the device driver's receive buffer

When the NBS card receives data, the Device Driver moves the data into an internal buffer if no read requests are pending. When a program calls ReadFile(), the Device Driver transfers data from the internal receive buffer to a system buffer, then the Windows 2000 Kernel moves the

data to the buffer provided by the caller of ReadFile(). If the internal receive buffer fills up, data can be lost. The internal receive buffer must be sufficiently large, but how large that is will depend on the application. The current size of the internal receive buffer can be obtained by invoking I/O control function IOCTL_NBS10_GET_PROPERTIES, as shown in the code example below. Your program can also check to see if a buffer overflow has occurred (see section 3.6.2).

The application program can also change the size of the internal receive buffer using I/O control function IOCTL_NBS10_SET_QUEUE_SIZE. The following code example shows how this could be done. Keep in mind that if you are using a 9 data bit character format, two bytes are required to buffer each character, whereas 5-8 data bit character formats require only one byte to buffer each character.

```
BOOL nbs10DisplayProperties()
/* This function will display (using printf) some Device Driver properties;
 * returns FALSE if an error occurred */
{
        extern HANDLE nbs10;
        DWORD returned;
        NBS10_DEVICE_PROPERTIES properties;

        // first query the current setting
        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_GET_PROPERTIES,   // I/O control code
                        NULL,
                        0,
                        &properties,                  // the result goes here
                        sizeof(properties),
                        &returned,
                        NULL                          // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_GET_PROPERTIES) failed, error code %d.\n",
                         GetLastError()
                        );
                return FALSE;
        }

        printf("Device Driver version: %hu\n", properties.PacketVersion);
        printf("Maximum baud rate: %lu\n", properties.MaxBaud);
        printf("Current internal receive buffer length: %lu\n", properties.CurrentRxQueue);

        return TRUE;
}

BOOL nbs10SetInternalRcvBufferSize(ULONG newSize)
/* This function adjusts the size of the internal receive buffer (in bytes).
 * Note that IOCTL_NBS10_SET_QUEUE_SIZE will never reduce the size of the
 * buffer, so if newSize < the current size then the buffer size will not change. */
/* returns FALSE if an error occurred */
{
        extern HANDLE nbs10;
        DWORD returned;
        NBS10_QUEUE_SIZE queueSize;

        queueSize.InSize = newSize;
```

```
        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_SET_QUEUE_SIZE,          // I/O control code
                        &queueSize,
                        sizeof(queueSize),
                        NULL,
                        0,
                        &returned,
                        NULL                                 // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_SET_QUEUE_SIZE) failed, error code %d.\n",
                        GetLastError()
                        );
                return FALSE;
        }

        return TRUE;
}
```

3.3 Handshaking Lines and Transmit-Receive Control

The NBS-10 has three "modem control" lines with potential utility. RTS (an output) and CTS (an input) are present on the DB-9 connectors as RS-422 signals. RTS and DTR (an output) can be used for transmit-receive direction control of the NBS-10's RS-485 line driver.

The NBS-11, NBS-12, NBS-20, NBS-41, and NBS-42 use only the RTS line for transmit-receive direction control, and no modem control signals are present on the external RS-485 connectors on those cards.

3.3.1 Automatic control of handshaking lines

The Device Driver can be configured to do certain types of handshaking line control automatically. For example, in RS-485 applications using the NBS-10, either RTS or DTR must be used for transmit-receive direction control: the selected line must be set to '1' during transmissions and '0' at all other times. It is almost always best to allow the Device Driver and the UART to do that automatically; see section 2.2 for details.

The RTS and CTS lines are sometimes used for low-level flow control. For example, a computer may signal its intention to transmit by raising the RTS line, and in response, the device to which it is attached will raise the CTS line when it is able to accept transmitted data. Your program can have this functionality if you configure the Device Driver for RTS Transmit Toggle (see section 2.2) and your program enables CTS handshaking (demonstrated in the following code example). One word of caution: in this mode, if you are not using overlapped I/O (described in section 3.7), you may wish to set the write timeouts to non-zero values (see section 3.1.4) to ensure that an attempted transmission will terminate within a predictable length of time.

```
BOOL nbs10CTSHandshakeMode()
/* returns FALSE if an error occurs */
/* In this mode, data will only be transmitted while CTS=1 */
{
        extern HANDLE nbs10;
        DWORD returned;
        NBS10_HANDFLOW handFlow;
```

```
        // first query the current setting
        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_GET_HANDFLOW,            // I/O control code
                        NULL,
                        0,
                        &handFlow,                           // the result goes here
                        sizeof(handFlow),
                        &returned,
                        NULL                                 // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_GET_HANDFLOW) failed, error code %d.\n",
                         GetLastError()
                        );
                return FALSE;
        }

        handFlow.ControlHandShake |= NBS10_CTS_HANDSHAKE;

        // then modify the setting to enable CTS handshaking
        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_SET_HANDFLOW,            // I/O control code
                        &handFlow,                           // the new setting
                        sizeof(handFlow),
                        NULL,
                        0,
                        &returned,
                        NULL                                 // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_SET_HANDFLOW) failed, error code %d.\n",
                         GetLastError()
                        );
                return FALSE;
        }

        return TRUE;
}
```

3.3.2 Manual control of handshaking lines

Manual control of RTS or DTR is possible when the Device Driver is not configured to use that bit for Transmit Toggle. The Device Driver supports four I/O control operations for those lines:

IOCTL_NBS10_SET_DTR
IOCTL_NBS10_CLR_DTR
IOCTL_NBS10_SET_RTS
IOCTL_NBS10_CLR_RTS

Here is a code example demonstrating how to set the RTS bit; this function will not work if the Device Driver's Transmit Toggle configuration parameter is set to RTS. The other three operations are performed in an identical manner except for the I/O control code.

```
BOOL nbs10SetRTS()
/* sets the RTS bit to 1 in the UART; returns FALSE if an error occurs */
```

```
{
        extern HANDLE nbs10;
        DWORD returned;

        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_SET_RTS,          // I/O control code
                        NULL,
                        0,
                        NULL,
                        0,
                        &returned,
                        NULL                          // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_SET_RTS) failed, error code %d.\n",
                         GetLastError()
                        );
                return FALSE;
        }
        return TRUE;
}
```

The following code example shows how to read the current value of the CTS line using I/O control function IOCTL_NBS10_GET_MODEMSTATUS:

```
INT nbs10ReadCTS()
/* returns -1 if an error occurs, otherwise returns the value of CTS (0 or 1) */
{
        extern HANDLE nbs10;
        DWORD returned;
        ULONG modemStatus;

        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_GET_MODEMSTATUS,  // I/O control code
                        NULL,
                        0,
                        &modemStatus,                 // the Device Driver reads the 82510's modem
                                                      // status register and writes the value here
                        sizeof(modemStatus),
                        &returned,
                        NULL                          // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_GET_MODEMSTATUS) failed, error code %d.\n",
                        GetLastError()
                        );
                return -1;
        }
        return (modemStatus & 0x10) ? 1 : 0;          // mask all but the CTS bit (0x10)
}
```

3.4 Purging the Device Driver Buffers

Purging the Device Driver discards all characters currently waiting in its receive buffer, and aborts any transmit or receive operations currently in progress, as shown in the following example.

```
BOOL nbs10PurgeAll()
/* returns FALSE if an error occurs */
{
        extern HANDLE nbs10;
        DWORD returned;
        ULONG ulPurgeMask =    NBS10_PURGE_TXABORT |  // abort all pending write requests
                               NBS10_PURGE_RXABORT |  // abort all pending read requests
                               NBS10_PURGE_TXCLEAR |  // does nothing at present
                               NBS10_PURGE_RXCLEAR;   // clear the internal receive buffer

        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_PURGE,            // I/O control code
                        &ulPurgeMask,
                        sizeof(ulPurgeMask),
                        NULL,
                        0,
                        &returned,
                        NULL                          // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_PURGE) failed, error code %d.\n",
                        GetLastError()
                        );
                return FALSE;
        }

        return TRUE;
}
```

3.5 Waiting on Device Driver Events

Windows 2000 includes support for events, which provide a mechanism for waiting for something to occur without requiring polling. The Device Driver supports the explicit use of events in a way similar to Win32 functions SetCommMask(), GetCommMask(), and WaitCommEvent(). However, the Device Driver only supports the following "waitable" events: received character, transmitter empty, CTS change, and line status error.

The following example illustrates a situation in which a program thread suspends until a character is received; similar functionality could also be implemented using timeouts and ReadFile() (see section 3.1.4).

```
BOOL nbs10WaitForReceivedCharacter()
/* returns FALSE if an error occurs */
{
        extern HANDLE nbs10;
        DWORD returned;
        ULONG ulWaitMask = NBS10_EV_RXCHAR;          // the event mask: received character
                // other possible masks are:
                // NBS10_EV_TXEMPTY to wait until the UART's transmitter is idle
                // NBS10_EV_CTS indicates a change in CTS
                // NBS10_EV_ERR for line status errors

        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_SET_WAIT_MASK,    // I/O control code
                        &ulWaitMask,
                        sizeof(ulWaitMask),
```

```
                NULL,
                0,
                &returned,
                NULL                            // not overlapped
                )) {
            printf("DeviceIoControl (NBS10_SET_WAIT_MASK) failed, error code %d.\n",
                GetLastError()
                );
            return FALSE;
        }

        if (!DeviceIoControl(
                nbs10,
                IOCTL_NBS10_WAIT_ON_MASK,     // I/O control code
                NULL,
                0,
                &ulWaitMask,
                sizeof(ulWaitMask),
                &returned,
                NULL
                )) {
            printf("DeviceIoControl (NBS10_WAIT_ON_MASK) failed, error code %d.\n",
                GetLastError()
                );
            return FALSE;
        }

        return TRUE;
}
```

3.6 Errors

3.6.1 Win32 function errors

When a program calls a Win32 function in order to use the Device Driver, errors are possible. These errors usually indicate a program bug or an NBS card configuration problem. Win32 functions will indicate an error by returning a certain value, as shown in the following list:

CreateFile() returns INVALID_HANDLE_VALUE if an error occurred
ReadFile() returns FALSE if an error occurred
WriteFile() returns FALSE ...
DeviceIoControl() returns FALSE ...
FlushFileBuffers() returns FALSE ...
CloseHandle() returns FALSE ...

To find out more information about the particular kind of error, a program can call Win32 function GetLastError() immediately after an error occurs. GetLastError() returns a number specifying the most recent error which occurred in that program thread. The error numbers are enumerated and briefly described in Win32 SDK file WINERROR.H.

3.6.2 Low-level communication errors

The low-level communication errors that the Device Driver can detect are as follows.

- *framing errors* are usually caused by a character format discrepancy or significant baud rate mismatch
- *parity errors* most often occur in the same circumstances as framing errors
- *overrun errors* are rare at moderate communication speeds; please contact Cimetrics if you see one
- *buffer overflow errors* can usually be avoided by using a large receive buffer (see section 3.2.3) and by calling ReadFile() frequently
- a *break signal* is a SPACE condition which lasts for more than one character period; some protocols use the break signal as a special control signal, but otherwise it indicates some kind of communication problem (e.g. a large baud rate mismatch)

The Device Driver maintains a set of flags to record the occurrence of these errors. Your program can determine if these errors are occurring using I/O control function IOCTL_NBS10_GET_DEVICE_STATUS, as shown in the following example code.

```
BOOL nbs10DisplayDeviceStatus()
/* Display (using printf) the Device Driver status; returns FALSE if an error occurs */
{
        extern HANDLE nbs10;
        DWORD returned;
        NBS10_DEVICE_STATUS deviceStatus;

        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_GET_DEVICE_STATUS,        // I/O control code
                        NULL,
                        0,
                        &deviceStatus,                        // the result goes here
                        sizeof(deviceStatus),
                        &returned,
                        NULL                                  // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_GET_DEVICE_STATUS) failed, error code %d.\n",
                         GetLastError()
                        );
                return FALSE;
        }

        printf("\n%lu bytes waiting in the receive buffer\n",
                deviceStatus.AmountInInQueue);

        // note: error flags are cleared after each call, so only new errors are displayed
        if (deviceStatus.Errors > 0) {
                if (deviceStatus.Errors & NBS10_ERROR_BREAK)
                        puts("Break detected");
                if (deviceStatus.Errors & NBS10_ERROR_FRAMING)
                        puts("Framing error detected");
                if (deviceStatus.Errors & NBS10_ERROR_OVERRUN)
                        puts("Overrun error detected");
                if (deviceStatus.Errors & NBS10_ERROR_QUEUEOVERRUN)
                        puts("Receive buffer overflow error");
                        // if this error occurs, you might consider increasing the receive buffer
                        // size using function nbs10SetInternalRcvBufferSize()
                if (deviceStatus.Errors & NBS10_ERROR_PARITY)
                        puts("Parity error detected");
        } else {
                puts("No errors");
```

```
        }

        return TRUE;
}
```

If you would rather that your program find out about errors more quickly, you can use error abort mode, as shown in the following code example. If this is selected, then after the occurence of an error, the invocation of any Device Driver service will fail with GetLastError() returning ERROR_OPERATION_ABORTED until IOCTL_NBS10_GET_DEVICE_STATUS is invoked.

```
BOOL nbs10ErrorAbortMode()
/* Returns FALSE if an error occurs, otherwise returns TRUE */
{
        extern HANDLE nbs10;
        DWORD returned;
        NBS10_HANDFLOW handFlow;

        // first query the current setting
        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_GET_HANDFLOW,     // I/O control code
                        NULL,
                        0,
                        &handFlow,                    // the result goes here
                        sizeof(handFlow),
                        &returned,
                        NULL                          // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_GET_HANDFLOW) failed, error code %d.\n",
                         GetLastError()
                        );
                return FALSE;
        }

        handFlow.ControlHandShake |= NBS10_ERROR_ABORT;

        // then modify the setting to enable error abort mode
        if (!DeviceIoControl(
                        nbs10,
                        IOCTL_NBS10_SET_HANDFLOW,     // I/O control code
                        &handFlow,                    // the new setting
                        sizeof(handFlow),
                        NULL,
                        0,
                        &returned,
                        NULL                          // not overlapped
                        )) {
                printf("DeviceIoControl (NBS10_SET_HANDFLOW) failed, error code %d.\n",
                         GetLastError()
                        );
                return FALSE;
        }

        return TRUE;
}
```

3.7 Synchronous vs. Overlapped I/O

I/O in Windows NT can be synchronous or overlapped (asynchronous). Synchronous I/O is conceptually much simpler: when an I/O function such as ReadFile() is called synchronously, that function will not return until the requested I/O operation is complete or the timeout period expires. By contrast, an overlapped I/O function call returns immediately and the I/O operation is executed in the background by a different thread.

Internally, almost all Device Driver operations are overlapped. But the Win32 subsystem will force a call to a Win32 API function to behave synchronously from the standpoint of the application program unless the device was opened with flag FILE_FLAG_OVERLAPPED and the API function was passed a pointer to a structure of type OVERLAPPED.

3.8 Support for Other I/O Control Operations

The Device Driver also supports the following additional I/O control operations (not described in this document):

IOCTL_NBS10_GET_WAIT_MASK
IOCTL_NBS10_SET_CHARS
IOCTL_NBS10_GET_CHARS
IOCTL_NBS10_GET_DTRRTS
IOCTL_NBS10_SET_FIFOLEVEL
IOCTL_NBS10_GET_FIFOLEVEL

The following I/O control codes are defined in file NTDNBS10.H, but have not been implemented. They may be implemented in future versions of the Device Driver.

IOCTL_NBS10_RESET_DEVICE
IOCTL_NBS10_SET_BREAK_ON
IOCTL_NBS10_SET_BREAK_OFF
IOCTL_NBS10_IMMEDIATE_CHAR
IOCTL_NBS10_SET_XOFF
IOCTL_NBS10_SET_XON
IOCTL_NBS10_XOFF_COUNTER
IOCTL_NBS10_LSRMST_INSERT

4 Technical Information You Probably Won't Need

This section contains assorted technical information not essential for the installation, configuration, and general operation of the Device Driver.

4.1 Device Driver Files

The Device Driver is a file, NBSXNT.SYS, which is installed in the directory on your PC's hard disk that contains all Windows NT device drivers (usually C:\WINNT\SYSTEM32\DRIVERS).

The Property Page Provider that is used for configuration of the Device Driver is NBSXPROP.DLL, which is usually installed in directory C:\WINNT\SYSTEM32.

File NTDNBS10.H is an include file for C-language programs which use the Device Driver. You should copy this file from the Device Driver diskette manually.

4.2 The Configuration Registry

According to Microsoft's Windows 2000 System Guide, the configuration registry is "a database repository for information about a computer's configuration." This database is structured like a tree. When you installed and configured the Device Driver, the setup program added some information about your NBS-1x/NBS-20/NBS-4x cards to the configuration registry. In almost all cases, you should use the Device Manager to change the configuration of your NBS-1x/NBS-20/NBS-4x cards. But there are occasional circumstances when you need to directly manipulate the configuration registry, which is done using the Windows 2000 registry editor program (regedit.exe or regedt32.exe), but be aware that incorrect changes can break your system, so ***use extreme caution!***

The following Registry keys are relevant to NBS-1x, NBS-20 and NBS-4x adapters and the Device Driver:

HKLM\System\CurrentControlSet\Control\Class\{C4E89B44-B736-45e6-9CCE-963BB01B9044}
HKLM\System\CurrentControlSet\Services\NbsxNt
HKLM\System\CurrentControlSet\Services\EventLog\System\NbsxNt
HKLM\System\CurrentControlSet\Enum\PCMCIA\CIMETRICS_INC....
HKLM\System\CurrentControlSet\Enum\Root\NineBitSerial
HKLM\System\CurrentControlSet\Enum\PCI\
VEN_10B5&DEV_9050&SUBSYS_000115B5&REV_01

4.3 Disabling and Uninstalling the Device Driver

If you need to completely remove the Device Driver from a PC, we recommend the following procedure:

0. You must be logged into an account which has sufficient privileges.
1. Exit from any programs that are using the Device Driver.
2. Uninstall all NBS-X devices from the Device Manager.

3. Delete file NBSXNT.SYS; usually it will be in directory C:\WINNT\SYSTEM32\DRIVERS.
4. Delete file NBSXPROP.DLL; usually it will be in directory C:\WINNT\SYSTEM32.
5. Delete the OEM?.INF file that is relevant to the Device Driver; usually it will be in directory C:\WINNT\INF.
6. Using the Windows 2000 Registry Editor, delete the NBSXNT service keys and class key (see section 4.2).

4.4 Avoiding Conflicts with Other Peripheral Devices and the Serial Device Driver

When an NBS-1x card is configured, care is required to avoid conflicts with other peripheral devices in the PC. In particular, each NBS-1x card requires exclusive use of eight consecutive I/O ports and one IRQ line. We recommend base I/O port address 0x300 and IRQ 5. However, you should first verify that no other peripheral devices are already using those resources. The Device Manager is useful in helping you to determine which resources are in use, but be aware that it does not always provide complete information.

Windows NT comes with a serial port device driver which is automatically started when Windows NT boots. That driver may recognize an NBS-1x card as a standard 16450-type serial port if the card is configured as COM1 (base I/O port address 0x3F8), COM2 (0x2F8), COM3 (0x3E8), or COM4 (0x2E8). If this happens, then the NBS-1x/NBS-4x Device Driver will not be able to access the NBS-1x.

There are a couple of ways you can get around this problem. Perhaps the simplest way is set the NBS-1x's base I/O port address to something other than COM1-4, in which case the serial port device driver will not notice the presence of the card. This will also ensure that the NBS-1x does not conflict with any serial port. Alternatively, you could disable the serial device driver. Note that disabling the serial device driver will not affect the operation of a serial mouse, which uses a different device driver, but will prevent you from using any of the other standard COM ports in your system. For example, you would not be able to use a modem installed in your PC if the serial device driver is not running.

It may also be possible to instruct the serial device driver to ignore a particular serial port, by modifying the Configuration Registry. A discussion of this procedure is beyond the scope of this document.

5 References

We highly recommend the following book:

> Programming Windows, Fifth Edition, Microsoft Press, 1998. "The definitive guide to the Win32® API."

The following books have much useful information about serial communication in general:

> Campbell, Joe, C Programmer's Guide to Serial Communications, Howard W. Sams & Company, 1993.
>
> Axelson, Jan, Serial Port Complete, Lakeview Research, 1998.

Other useful references are the Microsoft Developer Network Library (available on CD-ROM) and the developers section of Microsoft's web site (http://www.microsoft.com).