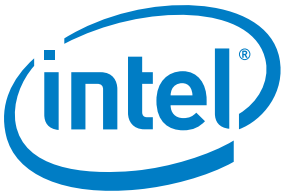

Intel® Pentium®/Celeron® Processor N3000 Family Windows* 7 Input/Output (I/O) Driver

Software Developer Manual

April 2015

Intel Confidential

Document Number: 557898-1.0

Contents

Tables

Revision History

Date	Revision	Description
April 2015	1.0	Initial release.

§

1.0 Introduction

1.1 Operating Systems

This manual set includes information pertaining to the following set of Operating Systems:

- Windows* 7 Ultimate 32 bit SP1
- Windows* 7 Ultimate 64 bit SP1
- Windows* Embedded Standard 32 bit SP1
- Windows* Embedded Standard 64 bit SP1

The I/O drivers are dependent on the Operating System (OS) driver installation.

Note: Minor updates can occur in the GPIO, I²C*, and HS-UART drivers structure definition in the public driver header file from one release to another. (For example, when upgrading from Alpha to Beta or Beta to Gold.) Be sure to recompile the application with the latest public driver header when updates occur.

1.2 Terminology

Table 1. Terminology

Term	Description
GPIO	General Purpose Input Output
HS-UART	High-Speed UART
I/O	Input/Output
I²C*	Inter Integrated Circuit
IOTL	Input / Output Control
OS	Operating System
SCL	Serial Clock
SDA	Serial Data Line

1.3 Reference Documents

Table 2. Reference Documents

Document	Document No./Location
Public Device Installation Functions	https://msdn.microsoft.com/en-us/library/ff549791.aspx
Synchronization and Overlapped Input and Output	https://msdn.microsoft.com/en-us/library/windows/desktop/ms686358(v=vs.85).aspx

§

2.0 General Purpose Input Output (GPIO) Driver

This section provides the programming details and interfaces exposed by the General Purpose Input Output (GPIO) driver for Windows*. The current implementation of the driver exposes the interfaces through Input/Output Controls (IOCTLs), which can be called from the application (user mode) using the Win32 API `DeviceIoControl`. Refer to the MSDN documentation listed in Table 2 for more details on this API.

The following sections provide information about the IOCTLs and how to use them to configure the GPIO hardware.

2.1 Driver Features

The GPIO Driver supports:

- Setting of different function for GPIO hardware
- Writing data to GPIO hardware
- Reading data from GPIO hardware
- Setting the direction of GPIO hardware
- Querying the function of GPIO hardware

2.2 Interface Details

Table 3 lists IOCTLs supported by the driver.

Table 3. Supported IOCTLs

No	IOCTL	Remarks
1	IOCTL_GPIO_READ	Read the data of selected pin of given GPIO controller
2	IOCTL_GPIO_WRITE	Write the data of selected pin of given GPIO controller
3	IOCTL_GPIO_DIRECTION	Set the direction of the selected pin of given GPIO controller
4	IOCTL_GPIO_MUX	Set the function of the selected pin of given GPIO port
5	IOCTL_GPIO_QUERY	Query the function of the selected pin of given GPIO port

2.3 IOCTL Usage Details

This section assumes a single client model where there is a single application-level program configuring the GPIO interface and initiating I/O operations. The following files contain the details of the IOCTLs and data structures used:

- `public.h` – contains IOCTL definitions, data structures and other variables used by the IOCTLs.

2.3.1 IOCTL_GPIO_MUX

This IOCTL is called to set the function mode of the selected pin of the given GPIO controller. The prerequisite is that the device must be installed and opened using the Win32 API `CreateFile`.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.data = function;
DeviceIoControl(hHandle,
   IOCTL_GPIO_DIRECTION,
   & GPIO_PIN_PARAMETERS,
   sizeof(GPIO_PIN_PARAMETERS),
   NULL,
   0,
   &dwSize,
   NULL);
```

2.3.2 IOCTL_GPIO_DIRECTION

This IOCTL is called to set the direction of the selected pin of given GPIO controller. The prerequisite for this is that the device must be installed and opened using the Win32 API `CreateFile` and the pin is set to GPIO function mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.ConnectMode = direction;
DeviceIoControl(hHandle,
   IOCTL_GPIO_DIRECTION,
   & GPIO_PIN_PARAMETERS,
   sizeof(GPIO_PIN_PARAMETERS),
   NULL,
   0,
   &dwSize,
   NULL);
```

2.3.3 IOCTL_GPIO_READ

This IOCTL reads the data of selected pin of the given GPIO controller. The prerequisite is that the device must be installed and opened using the Win32 API `CreateFile`.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
DeviceIoControl(hHandle,
   IOCTL_GPIO_READ,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);
```

2.3.4 IOCTL_GPIO_WRITE

The write operation writes to the selected pin of the GPIO controller. The prerequisite is that the device must be installed and opened using the Win32 API `CreateFile` and the pin direction is set to output.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.data = ConnectModeOutput;
DeviceIoControl(hHandle,
   IOCTL_GPIO_DIRECTION,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);

parameter.pin = pin;
DeviceIoControl(hHandle,
   IOCTL_GPIO_WRITE,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);
```

2.3.5 IOCTL_GPIO_QUERY

This IOCTL is called to query the function mode of the selected pin of the given GPIO controller. The prerequisite for this is that the device must be installed and opened using the Win32 API `CreateFile`.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
DeviceIoControl(hHandle,
   IOCTL_GPIO_QUERY,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);
```

2.4 Structures, Enumeration, and Macros

This section provides the details on the structures, enumerations and macros used by interfaces exposed by the GPIO driver. All the structures, enumerations and macros used by the interfaces are defined in `public.h`.

2.4.1 Structures

2.4.1.1 GPIO Pin Parameters

This structure is a used for preserving information related to the GPIO request.

Table 4. GPIO Pin Parameters

Name	Description
ULONG pin	Select the pin number
union { ULONG data; GPIO_CONNECT_IO_PINS_MODE ConnectMode; } u;	Data in the case of read return the read pin value, Data in the case of write is the data to be written to the pin, Data in the case of mux is the function to be set to the pin, Data in the case of query return the function of pin. ConnectMode in the case of direction set the direction of the pin.

2.4.2 Enumeration

2.4.2.1 GPIO_CONNECT_IO_PINS_MODE

This enum is used for preserving information related to the direction.

Table 5. I/O Connection Mode

Name	Description
CONNECT_MODE_INPUT	Set direction as input
CONNECT_MODE_OUTPUT	Set direction as output

2.4.3 Macros

Currently there are no macros defined for the GPIO driver.

2.5 Error Handling

Since the IOCTL command is implemented using the Windows* API, the return value of the call is dependent on and defined by the OS. On Windows*, the return value is a non-zero value. If the error is detected within or outside the driver, an appropriate system defined value is returned by the driver.

2.6 Inter-IOCTL Dependencies

There are no inter-IOCTL dependencies for GPIO driver. Once the driver is loaded successfully, the IOCTLs listed in Table 3 can be used in any order.

2.7 Programming Guide for GPIO Driver

This section describes the basic procedure for using the GPIO driver from a user mode application. All operations are through the IOCTLs exposed by the GPIO driver. Refer to Section 4.3 for details on the IOCTLs. The steps used to access the GPIO driver from the user mode application are described below:

1. Open the device.
2. Initialize and configure the driver with desired settings through the interfaces exposed.
3. Perform read/write operations.
4. Close the device.

2.7.1 Opening the Device

The GPIO driver is opened using the Win32 `CreateFile` API, and the GUID interface is exposed by the driver.

A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, and then enable an instance of the class for each device object to which user-mode I/O requests might be sent. The GPIO driver registers the following interface.

No	Interface Name
1	GUID_DEVINTERFACE_GPIO

This is defined in `public.h`. Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code can use `SetupDiXxx` functions to find out about registered, enabled device interfaces.

Please refer the following link to get the details about `SetupDiXxx` functions:
http://msdn.microsoft.com/en-us/library/ff549791.aspx

There are three GPIO controllers in the system. First determine which GPIO controller you want to open. You can find the controller type by calling `SetupDiGetDeviceInterfaceDetail` and checking the path name returned.

- If the device path returns “\\?\acpi#int33b2#1”, then the controller is `GPIO SCORE`.
- If the device path returns “\\?\\acpi#int33b2#2”, then the controller is `GPIO NCORE`.
- If the device path returns “\\?\\acpi#int33b2#3”, then the controller is `GPIO SUS`.

2.7.2 Driver Configuration

The following IOCTLs are used to initialize, configure, and query the settings for the GPIO driver:

- IOCTL_GPIO_DIRECTION
- IOCTL_GPIO_MUX
- IOCTL_GPIO_QUERY

The Win32 `DeviceIoControl` API is used to send information to the GPIO driver.

Direction Operation

This IOCTL used to set the pin direction when pin is in GPIO function mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.ConnectMode = direction
DeviceIoControl(hHandle,
   IOCTL_GPIO_DIRECTION,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);
```

Note: The `parameter.u.ConnectMode` is used to set the pin direction.

Mux Operation

This IOCTL used to set pin to select function mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.data = function;
DeviceIoControl(hHandle,
   IOCTL_GPIO_MUX,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);
```

Note: The `parameter.u.data` is used to set the pin function.

Query Operation

This IOCTL used to query the pin function mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
DeviceIoControl(hHandle,
   IOCTL_GPIO_QUERY,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);
```

Note: The `parameter.u.data` is used to save the returned pin function value.

2.7.3 Read and Write Operation

IOCTL_GPIO_READ and IOCTL_GPIO_WRITE are used for read and write operations respectively.

Read Operation

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
DeviceIoControl(hHandle,
   IOCTL_GPIO_READ,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);
```

Note: The `parameter.u.data` is used to save the return pin value.

Write Operation

To write a value to a pin, the pin must first be set to output mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.ConnectMode = ConnectModeOutput;
DeviceIoControl(hHandle,
   IOCTL_GPIO_DIRECTION,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);

parameter.pin = pin;
parameter.u.data = value;
DeviceIoControl(hHandle,
   IOCTL_GPIO_WRITE,
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &parameter,
   sizeof(GPIO_PIN_PARAMETERS),
   &dwSize,
   NULL);
```

Note: The `parameter.u.data` is used to set the value write to the pin.

2.7.4 Close the Device

Once all the operations related to the GPIO driver are finished, the device handle must free the application by calling the Win32 API `CloseHandle`.

```
CloseHandle(hHandle);
```

§

3.0 Inter Integrated Circuit (I^2C*) Driver

This section describes the programming details of the Inter Integrated Circuit (I^2C*) driver for Windows* 7. This includes the information about the interfaces exposed by the driver and how to use the interfaces to drive the I^2C* hardware through Input/Output Controls (IOCTLs), which can be called from the application (user mode) using the Win32 API `DeviceIoControl`. Refer to the MSDN documentation for more details on this API.

The I^2C* is a multi-master serial computer bus that is used to attach low-speed peripherals to a motherboard or embedded system. I^2C* uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock (SCL), pulled up with resistors.

3.1 Driver Features

The I^2C* driver supports:

- Setting different configurations for I^2C* hardware.
- Master device only.
- Setting I^2C* slave device address.
- Mode Select – fast mode (400 kbps) or standard mode (100 kbps) only.
- I^2C* Bus Master byte/multi-byte read transactions.
- I^2C* Bus Master byte/multi-byte write transactions.

3.2 Interface Details

Table 6 . IOCTL Interface Details

No	IOCTL	Remark
1	IOCTL_I2C_EXECUTE_WRITE	Configure slave address, address mode and speed, and then write data to the assigned slave device.
2	IOCTL_I2C_EXECUTE_READ	Configure slave address, address mode and speed, and then read data from the assigned slave device.
3	IOCTL_I2C_EXECUTE_SEQUENCE	Process a serial of Reads/Writes. Each one can have its own configuration. **NOTES:** • **Only ONE STOP bit is produced after all items of one sequence are done.** • **Do not combine two independent serials into one sequence, if each one must produce its respective STOP bit after it is completed.**

3.3 Structures and Macros

3.3.1 Structures

enum I2C_BUS_SPEED

This enum defines the I²C* transmission speeds.

enum I2C_ADDRESS_MODE

This enum defines the address modes for slave device.

struct I2C_SINGLE_TRANSMISSION

This structure contains transmission data and I²C* bus configuration.

struct I2C_SEQUENCE_TRANSMISSION

This structure contains one transmission data of one item in a sequence and related I²C* bus configuration.

3.3.2 Macros

I2C_SEQUENCE_TRANSMISSION_ENTRY

This macro helps initialize a sequence structure, which can contain more than one read/write item.

I2C_SEQUENCE_ITEM_INIT

This macro initializes related I²C* configuration and data buffer pointer of one item in a sequence transmission.

3.4 Error Handling

Since the IOCTL command is implemented using the Windows* API, the return value of the call is dependent on and defined by the OS. For Windows*, the return value is a non-zero value. If the error is detected within or outside the driver, an appropriate system defined value is returned by the driver.

3.5 Programming Guide

This section explains the basic procedure to use the I²C* driver from a user application mode. All operations are performed through the IOCTLs that are exposed by the I²C* driver.

3.5.1 Open Device

The I²C* driver is opened using the Win32 `CreateFile` API. To get the device name, use GUID interface exposed by the driver: **I2C_LPSS_INTERFACE_GUID**, defined in `public.h`.

A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, and then enable an instance of the class for each device object to which user-mode I/O requests might be sent.

Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code can use `SetupDiXxx` functions to find out about registered, enabled device interfaces. Please refer the following site to get details about `SetupDiXxx` functions: http://msdn.microsoft.com/en-us/library/dd406734.aspx

There are multiple I²C* controllers in the Intel® Processor N-series Family Platform and they all share the same GUID. Therefore, when the user-mode application opens the I²C* device using `SetupDiXxx`, it receives a device list of all I²C* controller interfaces. Then, the user-application compares the hardware ID controller list and finds the items that will open the correct I²C* controller.

3.5.2 Read, Write, and Sequence Operation

IOCTL_I2C_EXECUTE_READ, IOCTL_I2C_EXECUTE_WRITE and IOCTL_I2C_EXECUTE_SEQUENCE are used for read, write, and sequence operation, respectively.

Note: Maximum single transfer size is 64k, but this value may be updated in further, check the platform user guide for latest value.

Transmission Block Initialization

Before transmitting, a transmission structure variable must be defined in advance, shown in the following example.

```
I2C_SINGLE_TRANSMISSION transmission;
```

The application needs to use the asynchronous method of IOCTL to do the read/write operation. Before using `DeviceIoControl`, initialize the structure setting it to "overlapped." Refer to the following link to get detailed information: http://msdn.microsoft.com/en-us/library/windows/desktop/ms686358%28v=vs.85%29.aspx.

IOCTL_I2C_EXECUTE_READ Code

The following is an example of IOCTL_I2C_EXECUTE_READ.

```
#include “I2Cpublic.h”

I2C_SINGLE_TRANSMISSION readTransmission;
UCHAR readBuf[BUF_SIZE] = {};
UINT16 slaveAdr         = 0x1C;


readTransmission.Address        = slaveAdr;
readTransmission.AddressMode    = AddressMode7Bit;
readTransmission.BusSpeed       = I2C_BUS_SPEED_400KHZ;
readTransmission.DataLength     = sizeof(readBuf);
readTransmission.pBuffer        = readBuf;

status = DeviceIoControl(
               fileHandler,
                      IOCTL_I2C_EXECUTE_READ,
                      NULL,
                      0,
                      &readTransmission,
                      sizeof(readTransmission),
                      NULL,
                      &Overlapped);

if(status || (GetLastError() == ERROR_IO_PENDING))
{
    status = GetOverlappedResult(
                      fileHandler,
                      &Overlapped,
                      &bytesReturned,
                      TRUE);
     if(status)
     {
         /****
         * Now readBuf contains data that read from slave device.
         ****/
     }
}
```

IOCTL_I2C_EXECUTE_WRITE Code

The following is an example of IOCTL_I2C_EXECUTE_WRITE .

```
#include “public.h”

I2C_SINGLE_TRANSMISSION writeTransmission;
UCHAR writeBuf[BUF_SIZE] = {};
UINT16 slaveAdr         = 0x1C;


writeTransmission.Address        = slaveAdr;
writeTransmission.AddressMode    = AddressMode7Bit;
writeTransmission.BusSpeed       = I2C_BUS_SPEED_400KHZ;
writeTransmission.DataLength     = sizeof(writeBuf);
writeTransmission.pBuffer        = writeBuf;

status = DeviceIoControl(
                        fileHandler,
                        IOCTL_I2C_EXECUTE_WRITE,
                        &writeTransmission,
                        sizeof(writeTransmission),
                        NULL,
                        0,
                        NULL,
                        &Overlapped);

if(status || (GetLastError() == ERROR_IO_PENDING))
{
     status = GetOverlappedResult(
                         fileHandler,
                         &Overlapped,
                         &bytesReturned,TRUE);
     if(status)
     {
         /****
         * Now data in writeBuf have been transmitted to slave device.
         ****/
     }

}
```

IOCTL_I2C_EXECUTE_SEQUENCE Code

The following is an example of IOCTL_I2C_EXECUTE_SEQUENCE.

```
#include “public.h”


I2C_SEQUENCE_TRANSMISSION_ENTRY(2) sequence;
sequence.Size = 2;



USHORT regAddr = 0x0000;
UCHAR  readBuf[BUF_SIZE] = {};
UINT16 slaveAdr         = 0x1C;
UINT32 delayInUs        = 100;


/* Initialize write item in sequence*/
I2C_SEQUENCE_ITEM_INIT(
                        sequence.List[0],
                        AddressMode7Bit,
                        slaveAdr,
                        I2C_BUS_SPEED_400KHZ,
                        SpbTransferDirectionToDevice,
                        delayInUs,
                        sizeof(regAddr),
                        &regAddr);


/* Initialize read item in sequence*/
I2C_SEQUENCE_ITEM_INIT(
                       sequence.List[1],
                       AddressMode7Bit,
                       slaveAdr,
                       I2C_BUS_SPEED_400KHZ,
                       SpbTransferDirectionFromDevice,
                       delayInUs,
                       sizeof(readBuf),
                       readBuf);
status = DeviceIoControl(
                       fileHandler,
                       IOCTL_I2C_EXECUTE_SEQUENCE,
                       NULL,
                       0,
                       &sequence,
                       sizeof(sequence),
                       NULL,
                       &Overlapped);
```

```
if(status || (GetLastError() == ERROR_IO_PENDING))
{
     status = GetOverlappedResult(
                        fileHandler,
                        &Overlapped,
                        &bytesReturned,TRUE);
     if(status)
     {
         /****
         * Now data in regAddr have been transmitted to slave device,
         * and readBuf contains data read from slave device.
         * No STOP bit between item_0 and item_1.
         ****/
     }

}
```

3.5.3 Close Device

Once all operations related to the I²C driver are finished the device handle must free the application by calling the Win32 API `CloseHandle`.

```
CloseHandle(hHandle);
```

§

4.0 High-Speed UART Driver

This section provides the programming details of the High-Speed UART (HS-UART) driver for Windows. This includes information about the interfaces exposed by the driver and how to use those interfaces to drive the HS-UART hardware. The current implementation of the driver exposes the interfaces through the IOCTLs, which can be called from the application (user mode) using the Win32 API `DeviceIoControl`. Refer to the MSDN documentation for more details on this API.

The HS-UART bus is a communication bus that operates in full/half duplex mode. The SoC implements two instances of HS-UART controller that support baud rates between 300 and 3686400. Hardware flow control is also supported.

4.1 Driver Features

The HS-UART Driver allows setting different configurations for HS-UART hardware. It supports:

- Setting the StopBits / Parity Check / Word Length.
- Hardware flow control
- Different Baud rate – from 300 to 3686400.
- Read any setting from current hardware.

4.2 Interface Details

Table 7 lists the HS-UART driver and the supported IOCTLs.

Table 7. HS-UART Driver Support List

No	IOCTL	Description
1	IOCTL_SERIAL_SET_BAUD_RATE	This IOCTL is used to set the baud rate of transmission.
3	IOCTL_SERIAL_SET_LINE_CONTROL	This IOCTL is used to set Parity/StopBits/WordLength information to the devices.
7	IOCTL_SERIAL_SET_TIMEOUTS	This IOCTL is used to set the timeouts for transmission.
25	IOCTL_SERIAL_SET_HANDFLOW	This IOCTL is used to Flow control mode.

4.3 IOCTL Usage Details

4.3.1 IOCTL_SERIAL_SET_BAUD_RATE

The following IOCTL sets the baud rate for the operation.

```
BOOLEAN SetBaudrate(HANDLE hf,ULONG BaudRate_set)
{
    BOOL bResult;
    DWORD junk;
    bResult = DeviceIoControl(hf,IOCTL_SERIAL_SET_BAUD_RATE,
            &BaudRate_set,sizeof(BaudRate_set),NULL,0,&junk,(LPOVERLAPPED)NULL);
    if(bResult)
    {
        printf("Info : BaudRate set    OK.\n");
         return TRUE;
    }
    else
    {
         SetConsoleTextAttribute(hConsole, TEXT_COLOR_RED);
        printf("Error: BaudRate set    failed.\n");
         SetConsoleTextAttribute(hConsole, TEXT_COLOR_DEFAULT);
         return FALSE;
    }
}
```

4.3.2 IOCTL_SERIAL_SET_LINE_CONTROL

The following example of IOCTL is used to set Parity/StopBits/WordLength information.

```
BOOLEAN SetLineCtl(HANDLE hf,UCHAR StopBits,UCHAR Parity,UCHAR WordLength)
{
    BOOL bResult;
    DWORD junk;
    SERIAL_LINE_CONTROL LineCtl;
    LineCtl.Parity = Parity;
        LineCtl.StopBits = StopBits;
        LineCtl.WordLength = WordLength;
    bResult = DeviceIoControl(hf,IOCTL_SERIAL_SET_LINE_CONTROL,

&LineCtl,sizeof(SERIAL_LINE_CONTROL),NULL,0,&junk,(LPOVERLAPPED)NULL);
    if(bResult)
    {
        printf("Info : Linectl set     OK.\n");
         return TRUE;
    }
    else
    {
         SetConsoleTextAttribute(hConsole, TEXT_COLOR_RED);
        printf("Error: Linectl set     failed.\n");
         SetConsoleTextAttribute(hConsole, TEXT_COLOR_DEFAULT);
         return FALSE;
    }
}
```

4.3.3 IOCTL_SERIAL_SET_TIMEOUTS

The following IOCTL is used to set the timeouts for operation.

```
BOOLEAN SetTimeouts(HANDLE hf, ULONG baud)
{

    BOOL bResult;

    DWORD junk;

     COMMTIMEOUTS timeout;

     timeout.ReadIntervalTimeout = intervalTimeout;//1000;

     timeout.ReadTotalTimeoutConstant = readTimeout;

     timeout.ReadTotalTimeoutMultiplier = 0;//(1000*10/baud)+1;

     timeout.WriteTotalTimeoutConstant = writeTimeout;

     timeout.WriteTotalTimeoutMultiplier = 0;//(1000*10/baud)+1;


    bResult = DeviceIoControl(hf,IOCTL_SERIAL_SET_TIMEOUTS,

            &timeout,sizeof(timeout),NULL,0,&junk,(LPOVERLAPPED)NULL);
    if(bResult)
    {
        printf("Info : set Timtout OK.\n");

         printf("Info : Timeout.ReadIntervalTimeout = %d
ms\n",timeout.ReadIntervalTimeout);

         printf("Info : Timeout.ReadTotalTimeoutConstant = %d
ms\n",timeout.ReadTotalTimeoutConstant);

         printf("Info : Timeout.ReadTotalTimeoutMultiplier = %d
ms\n",timeout.ReadTotalTimeoutMultiplier);

         printf("Info : Timeout.WriteTotalTimeoutConstant = %d
ms\n",timeout.WriteTotalTimeoutConstant);

         printf("Info : Timeout.WriteTotalTimeoutMultiplier = %d
ms\n",timeout.WriteTotalTimeoutMultiplier);
```

```
        return TRUE;

    }

    else

    {

        SetConsoleTextAttribute(hConsole, TEXT_COLOR_RED);

       printf("Error: set Timeout failed.\n");

        SetConsoleTextAttribute(hConsole, TEXT_COLOR_DEFAULT);

        return FALSE;

    }

}
```

4.3.4 IOCTL_SERIAL_SET_HANDFLOW

The following IOCTL is used to set the flow control operation mode.

```
BOOLEAN SetHandflow(HANDLE hf,ULONG ControlHandShake,ULONG FlowReplace,LONG XonLimit,LONG
XoffLimit)

{
    BOOL bResult;

    DWORD junk;

    SERIAL_HANDFLOW HandFlow;

    HandFlow.ControlHandShake = ControlHandShake;

        HandFlow.FlowReplace = FlowReplace;

    HandFlow.XoffLimit = XonLimit;

        HandFlow.XonLimit = XoffLimit;

    bResult = DeviceIoControl(hf,IOCTL_SERIAL_SET_HANDFLOW,

            &HandFlow,sizeof(SERIAL_HANDFLOW),NULL,0,&junk,(LPOVERLAPPED)NULL);

    if(bResult)

    {
```

```
        printf("Info : HandFlow set    OK.\n");
         return TRUE;
    }
    else
    {
         SetConsoleTextAttribute(hConsole, TEXT_COLOR_RED);
        printf("Error: HandFlow set    failed.\n");
         SetConsoleTextAttribute(hConsole, TEXT_COLOR_DEFAULT);
         return FALSE;
    }
}
```

4.4 Structures and Macros

4.4.1 Enumerations

Table 8. UART Flow Controls

Name	Description
UART_SERIAL_FLAG_FLOW_CTL_NONE	None Flow Control
UART_SERIAL_FLAG_FLOW_CTL_HW	Hardware Flow Control
UART_SERIAL_FLAG_FLOW_CTL_XONXOFF	XON/XOFF Software Flow Control

4.4.2 HS-UART STRUCT and MICROS

```
typedef struct _PNP_UART_SERIAL_BUS_DESCRIPTOR {

    ULONG BaudRate;

    USHORT RxBufferSize;

    USHORT TxBufferSize;

    UCHAR Parity;

    UCHAR SerialLinesEnabled;

    // follwed by optional Vendor Data

    // followed by resource name string

} PNP_UART_SERIAL_BUS_DESCRIPTOR, *PPNP_UART_SERIAL_BUS_DESCRIPTOR;
```

4.5 Error Handling

Since the IOCTL command is implemented using the Windows* API, the return value of the call is dependent on and defined by the OS. On Windows*, the return value is a non-zero value. If the error is detected within or outside the driver, an appropriate system defined value is returned by the driver.

4.6 Programming Guide

This section describes the basic procedure for using the HS-UART driver from a user mode application. All operations are through the IOCTLs exposed by the HS-UART driver. Refer to Section 4.3 for details on the IOCTLs. The steps to access the HS-UART driver from the user mode application do the following:

1. Open the device.
2. Set the UART device.
3. Perform read/write operations.
4. Close the device.

4.6.1 Open Device

HS-UART driver is opened using the Win32 `CreateFile` API. To retrieve the device name, see below explanation.

FileName is `COM2~COMx`. The COM port numbers are shown in the `DeviceManager`.

4.6.2 Set UART Device

The following IOCTLs are used to set operations respectively. See Section 4.3.

`IOCTL_SERIAL_SET_BAUD_RATE`
`IOCTL_SERIAL_SET_LINE_CONTROL`
`IOCTL_SERIAL_SET_TIMEOUTS`
`IOCTL_SERIAL_SET_HANDFLOW`

4.6.3 Read/Write Operation

Read/Write HS-UART Device by Win32 `ReadFile/WriteFile` API.

4.6.4 Close Device

Once all the operations related to the HS-UART driver are completed, the device handle must be freed by the application by calling the Win32 API `CloseHandle`.

```
CloseHandle(hHandle);
```

§