Windows & Linux API interface code for PCI/PCIe UARTs' drivers

1.0 INTRODUCTION

PCI and PCIe UARTs will work with standard serial COM ports APIs. However, for advanced features such as Auto RS-485 Half-duplex flow control and controlling MPIOs, special APIs are necessary. This application note will discuss the Windows and Linux software API codes to control the MPIOs and Auto RS-485 Half-duplex flow control.

2.0 WINDOWS API INTERFACE CODE

Exar's PCI/PCIe UART drivers should be first installed for corresponding Windows OS. Then refer to the following sections.

2.1 CONTROLLING MPIOS

The following steps show the method to control the MPIOs.

2.1.1 Header definition file

The definitions shown in **Figure 1** below needs to be added to the header file.

FIGURE 1. DEFINITION IN THE HEADER FILE

```
#include <winioctl.h>

#define FILE_DEVICE_XRPORT  0x00008005
#define XRPORT_IOCTL_INDEX  0x805
#define IOCTL_XRPORT_READ_CONFIG_REG
CTL_CODE(FILE_DEVICE_XRPORT , \
                                    XRPORT_IOCTL_INDEX + 8,  \
                                    METHOD_BUFFERED,     \
                                    FILE_ANY_ACCESS)

#define IOCTL_XRPORT_WRITE_CONFIG_REG
CTL_CODE(FILE_DEVICE_XRPORT , \
                                    XRPORT_IOCTL_INDEX + 9,  \
                                    METHOD_BUFFERED,     \
                                    FILE_ANY_ACCESS)

typedef struct
{
   BYTE    bReg;
   BYTE    bData;

} CONFIG_WRITE, *PCONFIG_WRITE;
```

2.1.2 Application source code functions

The functions shown in Figure 2 below needs to be added to the application source code.

FIGURE 2. FUNCTIONS IN THE SOURCE CODE

```
// include the header file which has the above defines
// The application should supply bReg in the range of
// 0x00 - 0x13 respectively for 0x80 - 0x93
// Driver appends the CONFIG REG BASE 0x80 to this register
BOOL cfgWrite( BYTE bReg, BYTE bValue) {
    BOOL Status;
    DWORD  cbReturned;

    CONFIG_WRITE cfgWrite;
    cfgWrite.bReg = bReg;
    cfgWrite.bData = bValue;
    Status =    DeviceIoControl( m_hPortHandle,  //handle that you got from CreateFile
                    (DWORD) IOCTL_XRPORT_WRITE_CONFIG_REG,
                     &cfgWrite,
                     sizeof(CONFIG_WRITE),
                     NULL,
                     0,
                     &cbReturned,
                     0);
    if (!Status)
    {       MessageBox("Error in Writing...!", "Error", MB_OK|MB_ICONERROR);    }
    return Status;}

BOOL cfgRead( BYTE bReg, BYTE *pbValue ) {
    BOOL Status;
    DWORD  cbReturned;
    Status =    DeviceIoControl( m_hPortHandle,  //handle that you got from CreateFile
                    IOCTL_XRPORT_READ_CONFIG_REG,
                    &bReg,
                    sizeof(BYTE),
                    pbValue,
                    sizeof(BYTE),
                    &cbReturned,
                    0);
    if (!Status)
    {       MessageBox("Error in Reading...!", "Error", MB_OK|MB_ICONERROR);    }
    return Status;}
```

In the function in Figure 2, only the first port handle of the card can be used to access the configuration registers. The configuration register value should be in the range of 0x00-0x13 for PCI and 0x00-0x1B for PCIe UARTs.

2.1.3 Function call

To access the MPIOs, use this code at the appropriate place in your application.

FIGURE 3. FUNCTION CALL

```
HANDLE m_hPortHandle;

yourfunction()
{
        if ((hPortHandle = CreateFile("\\\\.\\COM5", // this should be the first port number of the card.
                                                 // In your  case it may not be COM5
                                   GENERIC_READ | GENERIC_WRITE,
                                   FILE_SHARE_WRITE,
                                   NULL,
                                   OPEN_EXISTING,
                                   FILE_ATTRIBUTE_NORMAL,
                                   NULL
                                   )) == ((HANDLE)-1))
        {
                return FALSE; // failure
        }
        else
        { // success }

        cfgWrite( 0x13, 0x00); // Config reg is offset by 0x80 in the driver, bits set to '0' for output
        // write to MPIOLVL register (cfgWrite (0x10, value) to control the output levels
       // or read to read the current output/input levels

}
```

2.2 Enable RS-485 Half Duplex Flow Control

To enable RS-485 half duplex flow control, the “xrapi.h” file needs to be included in the project. The “xrapi.h” file can be found in the folder with the driver source code.

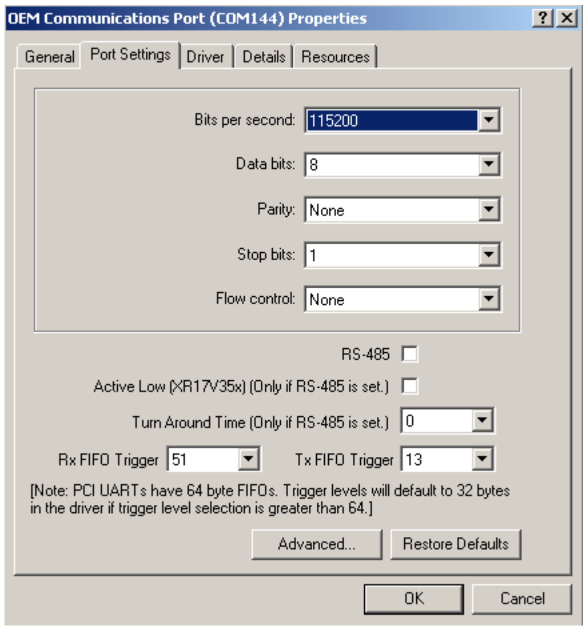
Figure 4. Enable RS-485 Half-duplex Control Mode

After opening and setting up the port using CreateFile and SetCommState, call DeviceIoControl with IOCTL_XRPORT_SET_RS485, as below:

```
//START --

COM_PORT_ENHANCED         portEnhanced;
DWORD  cbReturned;

portEnhanced.RS485 = 1; // 0 for reset

if (!DeviceIoControl( m_hPortHandle,
                    (DWORD) IOCTL_XRPORT_SET_RS485,
                    &portEnhanced,
                    sizeof(COM_PORT_ENHANCED),
                    NULL,
                    0,
                    &cbReturned,
                    0
                    ))
{
          MessageBox( NULL, "Unable to set RS485", NULL, MB_OK);
}

//END --
```

In the Windows 2K/XP (version 4.1.0.0 and newer) and Vista/7 (version 2.2.0.0 and newer) drivers, the RS-485 half-duplex feature can also be enabled (by checking the box next to the "RS-485") through the properties page (see Figure 5).

FIGURE 5. PROPERTIES PAGE



Bits per second: 115200
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None
RS-485 ☐
Active Low (XR17V35x) (Only if RS-485 is set.) ☐
Turn Around Time (Only if RS-485 is set.) 0
Rx FIFO Trigger 51
Tx FIFO Trigger 13

[Note: PCI UARTs have 64 byte FIFOs. Trigger levels will default to 32 bytes in the driver if trigger level selection is greater than 64.]

3.0 LINUX API INTERFACE CODE

The Linux standard serial port driver has support for XR17x15x UARTs in kernel version 2.6.8 and newer (see the "8250_pci.c" file). Exar provides custom drivers for both Linux kernel versions 2.4 and 2.6 for PCI (XR17x15x and XR17V25x) and PCIe (XR17V35x) UARTs. To use some advanced feature like Auto RS-485 half-duplex flow control, custom API codes (see section 3.3 and section 3.4) are necessary.

3.1 LINUX KERNEL VERSION 2.4

If Exar's default Vendor ID and Device ID are used, it is recommended to use the custom driver from Exar's website. If custom Vendor ID and Device ID are used, it is recommended to use the custom driver from Exar's website and modify to the custom Vendor ID and Device ID in the driver. An external EEPROM can be programmed to store the custom Vendor ID and Device ID.

3.2 LINUX KERNEL VERSION 2.6

3.2.1 Linux kernel version 2.6.7 and older

If Exar's default Vendor ID and Device ID are used, it is recommended to use the custom driver from Exar's website. If custom Vendor ID and Device ID are used, it is recommended to use custom driver from Exar's website and modify to the custom Vendor ID and Device ID in the driver. An external EEPROM can be programmed to store the custom Vendor ID and Device ID.

3.2.2 Linux kernel version 2.6.8 and newer

Linux kernel version 2.6.8 and newer have built-in support for Exar's XR17x15x PCI UARTs (see the "8250_pci.c" file) but do not have support for the XR17V25x and XR17V35x UARTs.

3.2.2.1 XR17x15x

If Exar's default Vendor ID and Device ID are used, it is recommended to disable the “8250_pci.c” driver, rebuild the kernel and then use the custom drivers from Exar's website. If custom Vendor ID and Device ID are used, it is recommended to use the custom driver from Exar's website and modify to the custom Vendor ID and Device ID in the driver. An external EEPROM can be programmed to store the custom Vendor ID and Device ID.

3.2.2.2 XR17V25x & XR17V35x

If Exar's default Vendor ID and Device ID are used, it is recommended to use custom driver from Exar's website. If custom Vendor ID and Device ID are used, it is recommended to use custom driver from Exar's website and modify to the custom Vendor ID and Device ID in the driver. An external EEPROM can be programmed to store the custom Vendor ID and Device ID.

See the summary in Table 1.

TABLE 1: VID AND DID REQUIREMENTS FOR USING LINUX CUSTOM DRIVERS

	LINUX KERNEL VERSION	DEFAULT EXAR VENDOR ID AND DEVICE ID	CUSTOM VENDOR ID AND DEVICE ID
XR17x15x	2.4	Use custom driver from Exar's website	Use custom driver from Exar's website and modify VID and DID in custom driver
	2.6.1 -- 2.6.7		
	2.6.8 -- latest	Disable “8250_pci.c” driver, rebuild kernel and then use the rebuilt driver	
XR17V25x & XR17V35x	2.4	Use custom driver from Exar's website without any changes	
	2.6.1 -- 2.6.7		
	2.6.8 -- latest		

Linux kernel version 2.4 and 2.6 drivers are available for download from Exar's website. If a driver for a specific kernel version is not available, send an email to uarttechsupport@exar.com.

3.3 Accessing MPIOs

To access MPIOs register set, the following code is necessary. For the PCI UARTs, please note that the file handle in the following Figure 6 should be that of the first port of the card.

Figure 6. Accessing MPIOs Description

```
//Accessing MPIO registers from the user space:
// define these in the app.

/*    EXAR ioctls */

//#define   FIOQSIZE          0x5460
#define     EXAR_READ_REG           (FIOQSIZE + 1)
#define     EXAR_WRITE_REG          (FIOQSIZE + 2)

struct xrioctl_rw_reg {
      unsigned char reg;
      unsigned char regvalue;
};

//To read MPIO register:
xrioctl_rw_reg input;

input. reg = 0x93; //MPIO register address is different from Windows API
ioctl(fd, EXAR_READ_REG, &input); // fd is the file handle obtained with 'open'
// input.regvalue will have the data read from the reg


//To write to MPIO register:
xrioctl_rw_reg input;

input.reg = 0x93; //MPIO register address is different from Windows API
input.regvalue = 0xFF;
ioctl(fd, EXAR_WRITE_REG, &input); // fd is the file handle obtained with 'open'
```

3.4 Enable auto RS-485 Mode

To enable auto RS-485 half duplex mode, the following code in **Figure 7** is necessary.

Figure 7. Enable auto RS-485 mode

```
//Enabling RS-485 mode from the user space
// define these in the app.

/*   EXAR ioctls */

//#define   FIOQSIZE         0x5460
#define     EXAR_READ_REG          (FIOQSIZE + 1)
#define     EXAR_WRITE_REG          (FIOQSIZE + 2)

struct xrioctl_rw_reg {
      unsigned char reg;
      unsigned char regvalue;
};

//To enable the auto RS-485 mode, the FCTR bit-5 needs to be set to '1'

//To write to FCTR register:
xrioctl_rw_reg input;
input.reg = 0x8;   //FCTR register offset address is 0x8;
ioctl(fd, EXAR_READ_REG, &input); // fd is the file handle obtained with 'open';
                                          // input.reg value will have the data read from the reg

input.regvalue = 0x20;  //Set FCTR bit-5 to '1'
ioctl(fd, EXAR_WRITE_REG, &input); //fd is the file handle obtained with 'open'


//To disable the auto RS-485 mode

xrioctl_rw_reg input;

input.reg = 0x8; //FCTR register offset address is 0x8;
ioctl(fd, EXAR_READ_REG, &input); // fd is the file handle obtained with 'open;
                                          //input.reg value will have the data read from the reg

input.regvalue &= 0xdf; // clear FCTR bit-5
ioctl(fd, EXAR_WRITE_REG, &input); // fd is the file handle obtained with 'open'
```

4.0 SUPPORT

If there are any questions, send them to **uarttechsupport@exar.com**.