OXFORD SEMICONDUCTOR

Application Note ***OXAN 5*** ***Software examples for the OX16C95x products***

Contents

Introduction

This document provides software examples for the OX16C95x family of products. This software can be used as example code for development of device drivers. Designers are encouraged to treat this code as a starting point for their own development and fully test their own code in the application environment.

Description

Fundamental I/O Operations

The OX16C95x consists of 35+ independent registers, yet to maintain backward compatibility with earlier devices, it has only 3 address lines – and hence only 8 unique I/O locations. For this reason the registers are grouped into 4 specific sets, each requiring different access conditions.

This section treats each of the four register sets (shown below) in turn, giving examples in C on how their registers are accessed.

1. Standard Register Set (450/550 compatible registers)
2. 650 Compatible Register Set
3. 950 Specific Register Set
4. 950 Indexed Control Register Set

Standard Register Access

This section gives details on how to access the OX16C95x standard register-set (550 compatible registers). These registers are the easiest to access and are described in the table below, a detailed explanation of these registers can be found in section 6 of the device data sheet.

Offset	Register	Description	R/W
000	THR	Transmitter Holding Register	W
000	RHR	Receiver Holding Register	R
001	IER	Interrupt Enable Register	R/W
010	FCR	FIFO Control Register	W
010	ISR	Interrupt Status Register	R
011	LCR	Line Control Register	R/W
100	MCR	Modem Control Register	R/W
101	LSR	Line Status Register	R
110	MSR	Modem Status Register	R
111	SPR	Scratch Pad Register	R/W
Access to the following registers require LCR[7] = 1			
000	DLL	Divisor Latch Low-byte	R/W
001	DLM	Divisor Latch High-byte	R/W

Table 1: Standard Register Set

Accessing these registers is simply a matter of reading and writing the specified offsets from the base address of the device. This can be done in C using the standard `_inp` and `_outp` functions included with `conio.h`.

For easier readability however, two macros have been defined to perform read and write operations. These functions will be used throughout the remainder of this document. It should also be noted that various simple types are also used extensively (such as `BYTE`, `WORD` etc). These can be included from `windows.h`.

```
#include <conio.h>

#define RD(addr)       _inp(addr)
#define WR(addr, data) _outp(addr, data)
```

In addition the structure DEVINFO is used by most of the functions defined here, as a container for information about the UART device. It is common for device drivers to use this type of structure to encapsulate related data. A most basic example of this structures content is given below:

```
typedef struct _DEVINFO{
      PDEVINFO device;
      BYTE uartType;
      // :
      // etc.
}DEVINFO, *PDEVINFO;
```

For completeness the following two functions have been included for accessing standard registers. These are simply wrappers, using the `DEVINFO` structure, for the `RD` and `WR` macros.

```
BYTE Read(PDEVINFO device, BYTE offset){
      return RD(device->device->baseAddr + offset);
}

void Write(PDEVINFO device, BYTE offset, BYTE value){
      WR(device->device->baseAddr + offset, value);
}
```

The only standard registers requiring special attention are the divisor latch word registers, DLL and DLM. Example functions for setting and reading the divisor word are given below.

```
#define DLL_OFFSET            0
#define DLM_OFFSET            1
#define LCR_OFFSET            3
#define LCR_DL_ACCESS_KEY     0x80

WORD ReadDivisor(PDEVINFO device){
      WORD dlldlm;
      BYTE oldLCR;
      // Store the current value of LCR and then
      // set the top bit to allow divisor latch access
      oldLCR = RD(device->baseAddr + LCR_OFFSET);
      WR(device->baseAddr + LCR_OFFSET, oldLCR | LCR_DL_ACCESS_KEY);
      //Construct the divisor word the restore LCR and return the value
      dlldlm = (RD(device->baseAddr + DLM_OFFSET)<<8);
      dlldlm += RD(device->baseAddr + DLL_OFFSET);
      WR(device->baseAddr + LCR_OFFSET, oldLCR);
      return dlldlm;
}


void WriteDivisor(PDEVINFO device, WORD divisor){
      BYTE oldLCR;
      // Store the current value of LCR and then
      // set the top bit to allow divisor latch access
      oldLCR = RD(device->baseAddr + LCR_OFFSET);
      WR(device->baseAddr + LCR_OFFSET, oldLCR | LCR_DL_ACCESS_KEY);
      // Write the divisor latch word then restore LCR
      WR(device->baseAddr + DLL_OFFSET, divisor & 0x00FF);
      WR(device->baseAddr + DLM_OFFSET,(divisor & 0xFF00)>>8);
      WR(device->baseAddr + LCR_OFFSET, oldLCR);
}
```

Using the four functions described in this section it is possible to fully configure 450/550 compatibility mode (see section 0 for more information on getting started). This basis can then be built upon to configure the more advanced features of the device.

NOTE: Although some registers may be accessed regardless of the state of LCR[7], it is strongly recommended that this bit is only set immediately prior to accessing DLL and DLM, and is cleared immediately afterwards (as in the above routines).

650 Compatible Register Access

This group of registers is used solely for configuring automatic flow control, with the exception of EFR bit 4 that is used to enable Enhanced Mode (See data sheet section 13.1). The table below gives a brief description of this register set.

Offset	Register	Description	R/W
010	EFR	Enhanced Features Register	R/W
100	XON1	XON1 Flow control character	R/W
101	XON2	XON2 Flow control character	R/W
110	XOFF1	XOFF1Flow control character	R/W
111	XOFF2	XOFF2 Flow control character	R/W

Table 21: 650 Compatible Register Set

Because these register offsets overlap the standard register set, a special access code must be written to LCR in order to access them. This access code (0xBF) corresponds to an invalid LCR mode. Writing it results in the bit 7 of LCR being latched but none of the other bits being changed.

NOTE: As with the divisor latch access bit, some standard registers may also be accessed with LCR = 0xBF. It is, however, strongly advised that this value is written and restored immediately prior to and following 650 compatible register accesses only.

Below are two example functions that can be used to access the 650 compatible registers.

```
#define LCR_OFFSET            3
#define LCR_650_ACCESS_KEY    0xBF

BYTE Read650(PDEVINFO device, BYTE offset){
      BYTE result, oldLCR;
      //Store the current LCR then write the access code
      oldLCR = RD(device->baseAddr + LCR_OFFSET);
      WR(device->baseAddr + LCR_OFFSET, LCR_650_ACCESS_KEY);
      //Read the register
      result = RD(device->baseAddr + offset);
      //Restore LCR and return the result
      WR(device->baseAddr + LCR_OFFSET, oldLCR);
      return result;
}


void Write650(PDEVINFO device, BYTE offset, BYTE value){
      BYTE oldLCR;
      //Store the current LCR then write the access code
      oldLCR = RD(device->baseAddr + LCR_OFFSET);
      WR(device->baseAddr + LCR_OFFSET, LCR_650_ACCESS_KEY);
      //Write the register
      WR(device->baseAddr + offset, value);
      //Restore LCR
      WR(device->baseAddr + LCR_OFFSET, oldLCR);
}
```

950 Specific Register Access

This register set consists of four registers, outlined in the table below. The first three of these registers provide additional status information for the device. The final one, ICR, is used as a common access window into the indexed control register set. This will be discussed in the following section (0).

Offset	Register	Description	R/W
001	ASR	Additional Status Register	R/W*
011	RFL	Receiver FIFO Fill Level (0-128)	R
100	TFL	Transmitter FIFO Fill Level (0-128)	R
101	*ICR*	*Indexed Control Register set common access point*	*R/W*

Table 3: 950 Specific Register Set

*Only the bottom two bits of ASR can be written on the 950 and only bit 0 can be written on the 954

Again, access to the first three of these registers requires the use of a special key to enable them. In this case the key must be written to ACR in the indexed control register set. As this has not yet been discussed, for the purpose of this example, we will assume the existence of two functions: `UnlockAdditionalStatus` and `LockAdditionalStatus`. These will be described fully in the following section.

NOTE: The following functions rely on the observation of the previous notes in this section. They will not work correctly if LCR was last written with 0xBF or if LCR[7] is set when they are called.

It is more practical to define a set of individual functions to access these registers, owing to varying features of their operation. The first two, given below are used to access ASR.

```
BYTE ReadASR(PDEVINFO device){
      //Returns the data stored in the ASR register
      BYTE retVal;
      UnlockAdditionalStatus(device);
      retVal = RD(device->baseAddr + ASR_OFFSET);
      LockAdditionalStatus(device);
      return retVal;
}

void WriteASRBit(PDEVINFO device, BYTE bit, BOOL value){
      // Sets the specified ASR bit to 1 if value = TRUE 0 if value = FALSE
      BYTE currentASR;
      if((bit==0)||(bit==1)){ //Only allow writable bits to be set
            UnlockAdditionalStatus(device);
            currentASR = RD(device->baseAddr + ASR_OFFSET);
            if(value){
                  // OR bit in if setting
            currentASR |= (1 << bit);
            }else{
                  // Mask bit out if clearing
            currentASR &= ~(1 << bit);
            }
            WR(device->baseAddr + ASR_OFFSET, currentASR);
            LockAdditionalStatus(device);
      }
}
```

The following function can be used to read FIFO fill levels. Due to the way these registers are updated, it is possible to read spurious values occasionally. To avoid this causing problems, the registers should be read until two consecutively read values are the same (i.e. the values are stable).

```
#define RECEIVE_FIFO    0
#define TRANSMIT_FIFO   1

#define RFL_OFFSET      3
#define TFL_OFFSET      4

BYTE ReadFIFOLevel(PDEVINFO device, BYTE fifo){
      BYTE level1, level2, offset;
      //Decide which FIFO we are looking at
      if(fifo == RECEIVE_FIFO) offset = RFL_OFFSET; else offset = TFL_OFFSET;
      UnlockAdditionalStatus(device);
      do{   // Read until two values the same
            level1 = RD(device->baseAddr + offset);
            level1 = RD(device->baseAddr + offset);
      }while (level2 != level2);
      LockAdditionalStatus(device);
      return level1;
}
```

NOTE: In very high-speed applications on a slow bus or machine, it may not be physically possible to do this due to the fill level changing more frequently than I/O accesses can occur. In this case it would be necessary to specify an appropriate *allowed difference* between consecutive values, which must be met before the reading is accepted. If the above routine was used in this situation, it would not return until either RFL reached 128 or TFL reached 0 (depending on which register was being read).

The final register in this set, ICR, is discussed in the following section.

950 Indexed control Register Set Access

As its name suggests, this register set indexed control. This simply means that to access a given register in this set, first an index must be written to the scratchpad register SPR (in the standard register set). The indexed register is then read or written via a common location (The ICR register mentioned in the previous section).

Although the use of SPR for indexing facilitates access to a further 256 registers, only 12 of these locations are used (14 on the 950). Accessing locations that do not appear in the table below may cause unpredictable results and should be avoided.

SPR Index	Register	Description	R/W
0x00	ACR	Advanced Control Register	R/W
0x01	CPR	Clock Prescaler Register	R/W
0x02	TCR	Times Clock Register	R/W
0x03	CKS*	Clock source register	R/W
0x04	TTL	Transmitter Trigger Level	R/W
0x05	RTL	Receiver Trigger Level	R/W
0x06	FCL	Flow Control Low trigger level	R/W
0x07	FCH	Flow Control High trigger level	R/W
0x08	ID1	Identification Register 1	R
0x09	ID2	Identification Register 2	R
0x0A	ID3	Identification Register 3	R
0x0B	REV	Revision Identification Register	R
0x0C	CSR	Channel reset Register	R/W
0x0D	NMR*	Nine bit Mode Register	R/W

Table 4: 950 Indexed control Register Set
* These registers are only available on the OX16C950B

Owing to the size of this register set, it is sensible to define a pair of generic functions for reading and writing its registers. Writing these registers is simple and is achieved using the method described above. The function below performs this operation.

```
#define SPR_OFFSET      7
#define ICR_OFFSET      5
#define ACR_INDEX       0x00

void WriteICR(PDEVINFO device, BYTE index, BYTE value){
      // Writes the ICR set register indexed by the index
      // parameter with value
      WR(device->baseAddr + SPR_OFFSET, index);
      WR(device->baseAddr + ICR_OFFSET, value);
      //Record changes made to ACR *
      if (index==ACR_INDEX) device->shadowACR = value;
}
```

* See following read function.

Reading ICR registers is slightly more involved as reading must first be enabled. This, in itself, requires a write to an ICR register. ACR (index 0) bit 6 is the read enable bit. This must be set to 1 to enable reading of the indexed control register set. This is all achieved with the following function:

```
#define ACR_ICR_READ_EN 0x40

BYTE ReadICR(PDEVINFO device, BYTE index){
      // Reads the ICR set register indexed by the index
      // parameter with value
      //Enable read access
      BYTE retVal;
      WriteICR(device,       ACR_INDEX,       (BYTE)(device->shadowACR       |
ACR_ICR_READ_EN));
      WR(device->baseAddr + SPR_OFFSET, index);
      retVal = RD(device->baseAddr + ICR_OFFSET);
      //Disable read access
      WriteICR(device,       ACR_INDEX,       (BYTE)device->shadowACR       &
~ACR_ICR_READ_EN));
      return retVal;
}
```

NOTE: Because ACR must be written before any ICR register can be read, and ACR *is* an ICR register, ACR can not be read without first overwriting it. This means that in order to read ACR we need to maintain a local copy of what was last written to it (the device itself never modifies the contents of ACR).

In the above examples, this copy is kept in a variable called `shadowACR` which is a member of the `DEVINFO` data. This variable can be initialised to zero prior to the first ACR access (i.e. after a device reset / power up) as, at this point, the contents of ACR are known to be zero.

Once again, it is recommended that the read enable bit is only set during reads of these registers, and is disabled again immediately afterwards.

Now functions have been defined which can read and write ICR registers, it is a simple task to define the functions mentioned in the previous section that toggle the access enable for the 950 specific register set. To access these registers ACR bit 7 must be set. The functions are therefore defined as follows:

```
void UnlockAdditionalStatus(PDEVINFO device){
      // Set the top bit of ACR to enable
      // 950 specific register set access
      device->shadowACR |= ACR_950_READ_EN;
      WriteICR(device, ACR_INDEX, device->shadowACR);
}

void LockAdditionalStatus(PDEVINFO device){
      // Clear the top bit of ACR to disable
      // 950 specific register set access
      device->shadowACR &= (~ACR_950_READ_EN);
      WriteICR(device, ACR_INDEX, device->shadowACR);
}
```

Getting Started

Identifying the OX16C95x UART Device

Identifying the OX16C95x device is a simple matter now that the functions for accessing the various registers of the device have been defined. All devices have a four-byte identification code that resides in four read only ICR registers. The first three bytes, when concatenated, form the part number of the device in hexadecimal, and the final part is a zero based revision number (0 = Revision A, 1 = B etc.). The table below shows how the IDs of the most recent devices are represented in the four registers.

Device	ID1	ID2	ID3	REV
OX16C950 Revision B	0x16	0xC9	0x50	0x00
OX16C954 Revision A	0x16	0xC9	0x54	0x00

Table 5: Device Identification Register Contents

The code given below can be used to identify a device and store its type. This can later be used for device specific operations. This function also makes it very simple to verify the existence of a UART device at a given address, something that is fairly complex to achieve reliably with earlier devices.

```
#define ID1_INDEX 0x08
#define ID2_INDEX 0x09
#define ID3_INDEX 0x0A
#define REV_INDEX 0x0B


BOOL DetectOX16C95x(PDEVINFO device){
      //Reads the 95x ID registers and stores their value
      BYTE id1, id2, id3, rev;
      BOOL detected = FALSE;
      id1 = ReadICR(device, ID1_INDEX);
      id2 = ReadICR(device, ID2_INDEX);
      id3 = ReadICR(device, ID3_INDEX);
      rev = ReadICR(device, REV_INDEX);

      if((id1==0x16)&&(id2==0xC9)&&((id3&0xF0)==0x50)){
            device->uartType = id3 & 0x0F;
            device->uartRev  = rev;
            device->shadowACR = 0;
            detected = TRUE;
      }
      return detected;
}
```

Mode Selection

The OX16C95x device can be operated in several modes to provide backward compatibility with previous UART devices (16C45x/55x/65x and 75x). These modes are summarised in the table below:

Mode	FIFO Size	Configuration			
		FCR[0]	FCR[5]	EFR[4]	FIFOSEL
450	1	0	X	X	X
550	16	1	0	0	0
Extended 550	128	1	X	0	1
650 (& 950)	128	1	X	1	X
750	128	1	1	0	0

Table 6: Device Mode Configuguration Options

For a full description of the features available in each mode, see the device data sheet section 5.

NOTE: The FIFOSEL pin is internally pulled down. The only time this pin need be connected therefore, is to enable extended 550 mode.

With the exception of Extended 550 mode then, all modes are configured by setting none, one, or two of the configuration bits shown in the table. This can be done with the following code fragments:

450 Mode:
This mode requires no configuration, this is the reset/power-up mode of the device.

550 Mode:
Setting 550 mode is simply a matter of writing the FCR register with the FIFO enable bit (bit 0):

```
#define FCR_FIFO_EN     1

DEVINFO device;
      :
Write(&device, FCR_OFFSET, FCR_FIFO_EN);
```

650 & 950 Modes:
There is no difference between the configuration of 650 and 950 modes. When this mode is used in conjunction with 950 specific features however, it will be referred to as 950 mode. Configuring this mode requires EFR[4] to be written. This is usually done *before* FCR is written to enable the FIFO:

```
Write650(&device, EFR_OFFSET, EFR_ENHANCED_MODE_EN);
Write(&device, FCR_OFFSET, FCR_FIFO_EN);
```

750 Mode:
To enable 750 mode FCR[5] must be set. This bit is guarded by LCR[7] (i.e. LCR[7] must be set in order to write to it). The code would therefore look something like this:

```
//Store current LCR value, unlock FCR[5], enable FIFO and restore LCR
BYTE oldLCR = Read(&device, LCR_OFFSET);
Write(&device, LCR_OFFSET, oldLCR | LCR_DL_ACCESS_KEY);
Write(&device, FCR_OFFSET, FCR_FIFO_EN | FCR_750MODE_EN);
Write(&device, LCR_OFFSET, oldLCR);
```

Because this code sets and clears the access key in LCR, encapsulating it in a function would provide a neater and safer solution.

NOTE: As FCR is not readable, it may also be useful to maintain a shadow copy of this registers contents (updated whenever FCR is written) in the device information structure.

Basic Operation Configuration

This section highlights the basic configuration required to get the OX16C95x device operating. This configuration can then be built upon, using the information in further sections, to make use of the more advanced features of the device. This basic configuration covers the following items:

- Setting the baud rate divisor word
- Setting the data framing mode (parity, stop bits etc.)
- Enabling internal loopback mode (for diagnostic purposes)
- Using LSR to for polled mode transmission/reception and data error checking.
- Transmitting and receiving data in polled mode (no interrupts)

Each of the above points are covered in turn and then brought together in a small example test program that configures the UART and verifies a 1MB data transfer in internal loop back mode.

Setting the baud rate divisor word
This is simple now that we have defined a function for writing the divisor word value. The baud rate of the device (serial bits per second) is specified by the following formula:

$$BaudRate = \frac{InputClkFrequency}{BaudRateDivisor \times ClocksPerBit}$$

Where *InputClkFrequency* is the frequency of the input clock to the device (typically 1,843,200Hz in standard applications) and *ClocksPerBit* is 16 by default (although this value is configurable from 4 to 16 inclusive in 950 mode – see section 0). Assuming these standard values then, the definitions below can be used to configure some standard baud rates:

```
#define DIVISOR_BAUD_110      0x0300
#define DIVISOR_BAUD_300      0x0180
#define DIVISOR_BAUD_600      0x00C0
#define DIVISOR_BAUD_1200     0x0060
#define DIVISOR_BAUD_2400     0x0030
#define DIVISOR_BAUD_4800     0x0018
#define DIVISOR_BAUD_9600     0x000C
#define DIVISOR_BAUD_19200    0x0006
#define DIVISOR_BAUD_28800    0x0004
#define DIVISOR_BAUD_38400    0x0003
#define DIVISOR_BAUD_57600    0x0002
#define DIVISOR_BAUD_115200   0x0001
```

e.g.

```
WriteDivisorWord(&device, DIVISOR_BAUD_115200);
```

Setting the data framing mode (parity, stop bits etc.)
The data framing used by both the UART transmitter and receiver is configured in the LCR register. This allows selection of the following:

- Number of data bits per character (5,6,7 or 8)*
- Number of stop bits to append to each character (1, 1.5 *[5-Bit data only]* or 2)
- Type of parity generation/checking to used (none, odd, even, forced high or forced low)

* 9-Bit data mode is available on the OX16C950 but this is configurable in a separate register.

Using a combination of the following definitions can make mode selection much easier.

```
#define LCR_5_BIT_DATA        0x00
#define LCR_6_BIT_DATA        0x01
#define LCR_7_BIT_DATA        0x02
#define LCR_8_BIT_DATA        0x03

#define LCR_1_STOP_BIT        0x00
#define LCR_1_5_STOP_BITS     0x04
#define LCR_2_STOP_BITS       0x04

#define LCR_NO_PARITY         0x00
#define LCR_ODD_PARITY        0x08
#define LCR_EVEN_PARITY       0x18
#define LCR_FORCE_HIGH_PARITY 0x28
#define LCR_FORCE_LOW_PARITY  0x38

#define LCR_FORCE_BREAK       0x40
```

Some common framing modes are defined below (care must be taken not to select an illegal mode):

```
#define LCR_MODE_8N2          LCR_8_BIT_DATA      |      LCR_2_STOP_BITS      |
LCR_NO_PARITY
#define LCR_MODE_8E1          LCR_8_BIT_DATA     |     LCR_1_STOP_BIT         |
LCR_EVEN_PARITY
#define LCR_MODE_8O1          LCR_8_BIT_DATA     |     LCR_1_STOP_BIT         |
LCR_ODD_PARITY
#define LCR_MODE_7E2          LCR_7_BIT_DATA      |      LCR_2_STOP_BITS      |
LCR_EVEN_PARITY
#define LCR_MODE_7O2          LCR_7_BIT_DATA      |      LCR_2_STOP_BITS      |
LCR_ODD_PARITY
#define LCR_MODE_5E1_5        LCR_5_BIT_DATA     |     LCR_1_5_STOP_BITS     |
LCR_EVEN_PARITY
```

Configuring the given mode is simply a matter of writing the constructed code to LCR, e.g.

```
Write(&device, LCR_OFFSET, LCR_MODE_8N2);
```

Enabling internal loopback mode
This mode is configured by setting bit 4 of MCR. Primarily used for testing, this mode internally connects the following pins together:

- SOUT to SIN
- RTS# to CTS#
- DTR# to DTR#
- OUT1# to RI#
- OUT2# to DCD#

This allows a single device to send data and signals to itself, hence allowing its input and output circuits to be tested without attaching external equipment. For production testing however, it is more realistic to loop the signals back externally, so the device’s I/O buffers and any associated line drivers are also tested.

Using the Line Status Register (LSR)
The LSR register stores information about the status of the transmitter, receiver and received characters.

Bit 0 of LSR indicates the availability of one or more characters in the receive FIFO. In polled mode reception (where interrupts are not used) LSR bit 0 is tested and, if set, the receiver FIFO is continually read until this bit is cleared again (i.e. all available data has been read). The code below shows this in its simplest form:

```
#define RHR_OFFSET            0
#define LSR_DATA_AVAILABLE    0x01
BYTE data;
      :
      :
while( (Read(&device, LSR_OFFSET) & LSR_DATA_AVAILABLE) == 0); // Do nothing
data = Read(&device, RHR_OFFSET);
```

Similarly LSR bits 5 & 6 reflect the status of the transmitter. When bit 5 is set, the *FIFO* is empty. When bit 6 is set, both the FIFO *and* shift register are empty i.e.. the transmitter is idle. (This implies that bit 6 will always go high exactly one character time *after* bit 5).

Polled mode transmission can therefore be achieved using the following code:

```
#define THR_OFFSET            0
#define LSR_DATA_AVAILABLE    0x01
BYTE data;
      :
      :
while( (Read(&device, LSR_OFFSET) & LSR_FIFO_EMPTY) == 0); // Do nothing
Write(&device, THR_OFFSET, data);
```

The remaining bits of LSR identify various data errors. These are described in the following table:

LSR bit	Name	Description
1	Overrun Error	A character was received when the FIFO was already full
2	Parity Error	The character was received with incorrect parity
3	Framing Error	The character was received with at least one invalid stop bit
4	Break	The SIN line was low for at least the whole character, including the parity bit and the first stop bit.
7	Data Error	There is at least one character with errors in the FIFO

Table 7: LSR Error Definitions

NOTE: The parity error, framing error and break bits are stored for each character in the receiver FIFO. The bits actually in LSR reflect those of the next character to be read. Other errors apply to all characters but are cleared next time LSR is read.

The following is a simple skeletal LSR error handler:

```
#define LSR_OVERRUN_ERROR     0x02
#define LSR_PARITY_ERROR      0x04
#define LSR_FRAMING_ERROR     0x08
#define LSR_BREAK             0x10
#define LSR_DATA_ERROR        0x80
#define LSR_ERROR_MASK        (LSR_OVERRUN_ERROR | LSR_PARITY_ERROR | \
                              LSR_FRAMING_ERROR | LSR_BREAK | LSR_DATA_ERROR)


BOOL HandleLSRErrors(PDEVINFO device, BYTE lsr){

      if ((lsr & LSR_ERROR_MASK) == 0) return FALSE;

      if(lsr & LSR_OVERRUN_ERROR){
            // Code to handle overrun
            printf("Overrun Error!\n");
      }
      if(lsr & LSR_PARITY_ERROR){
            // Code to handle parity error
            printf("Parity Error!\n");
      }
      if(lsr & LSR_FRAMING_ERROR){
            // Code to handle framing error
            printf("Framing Error!\n");
      }
      if(lsr & LSR_BREAK){
            // Code to handle break
            printf("Break!\n");
      }
      if(lsr & LSR_DATA_ERROR){
            // Code to handle data error
            printf("Data Error!\n");
      }
      return TRUE;
}
```

Modem Control and Status

The modem control and status registers allow the states of the various modem pins to be set and monitored respectively. Because these are standard registers, they can be accessed using the basic read and write operations (provided (ACR[7] is not set and the last value written to LCR was not 0xBF).

The following definitions have been provided to assist in setting/getting pin status.

```
// Modem Control Register Definitions
#define MCR_OFFSET              4
#define MCR_DTR                 0x01
#define MCR_RTS                 0x02
#define MCR_OUT1                0x04
#define MCR_INTERRUPT_EN        0x08

// Modem Status Register Definitions
#define MSR_OFFSET              6
#define MSR_DELTA_CTS           0x01
#define MSR_DELTA_DSR           0x02
#define MSR_RI_TRAILING_EDGE    0x04
#define MSR_DELTA_DCD           0x08
#define MSR_CTS                 0x10
#define MSR_DSR                 0x20
#define MSR_RI                  0x40
#define MSR_DCD                 0x80
```

For example, to activate the RTS and DTR outputs:

```
Write(&device, MCR_OFFSET, MCR_DTR + MCR_RTS);
```

Notice that the MSR register has three delta bits that are set whenever their respective line changes state. This allows for the detection of edges on the CTS, DSR and DCD inputs (bit 2 is also set on the falling edge of RI). For example, to detect CTS going active:

```
BYTE msr = Read(&device, MSR_OFFSET);

if( (msr & (MSR_CTS + MSR_DELTA+CTS)) = (MSR_CTS + MSR_DELTA_CTS) ){
      // CTS has gone active since the last read of MSR
}

if( (msr & (MSR_CTS + MSR_DELTA+CTS)) = MSR_DELTA_CTS ){
      // CTS has gone in-active since the last read of MSR
}
```

NOTE: If the INT_SEL# pin is tied low, bit 3 of MCR (OUT2) is used to enable the interrupt output pin for the device (MCR[3]=1 enables interrupts, MCR[3]=0 disables interrupts), as well as setting the state of the OUT2 pin. If the INT_SEL# pin is tied high however, the interrupt line is permanently enabled and MCR[3] does not affect it.

Example: Transmitting and receiving data in polled mode (no interrupts)

```
void main(int argc, char *argv[]){
      BOOL running = TRUE;
      BYTE outData = 0, inData, lsr;
      WORD inCount = 0, kb = 0; // Transfer counters

      // Get the base address argument if specified
      if(argc < 2){
            printf("Usage: LoopTest [base]\n"
                   "Where [base] is the UART base address in hex\n\n");
            return;}
      sscanf(argv[1], "%x", &device.baseAddr);

      // Configure the UART
      Write(&device, LCR_OFFSET, LCR_MODE_8E1);  // 8Bit data, even parity, 1
stop
      WriteDivisor(&device,  DIVISOR_BAUD_115200);//  115.2  kBaud  (1.8432  MHz
clk)
      Write(&device, FCR_OFFSET, FCR_FIFO_EN);
      Write(&device, MCR_OFFSET, MCR_INTERNAL_LOOP);

      printf("\tTranfering data: ");

      do{
            // Form a new data byte to send
            outData = (outData + 1) % 255;

            // Wait for Tx FIFO to empty before writing
            while((Read(&device, LSR_OFFSET) & LSR_FIFO_EMPTY)==0);
            Write(&device, THR_OFFSET, outData);

            // Wait for data to be received before reading
            do{   lsr = Read(&device, LSR_OFFSET);
            }while((lsr & LSR_DATA_AVAILABLE)==0);

            // Check last LSR for errors
            if(HandleLSRErrors(&device, lsr))running = FALSE;

            // Read and check received data
            inData = Read(&device, RHR_OFFSET);
            if(inData != outData){
                  printf("Incorrect Data Received!");
                  running = FALSE;
            }

            // Update transfer counter on every kB
            inCount++;
            if(inCount > 1024){
                  inCount %= 1024;
                  kb++;
                  printf("%.4dkB\b\b\b\b\b\b", kb);
                  if(kb == 1024) running = FALSE;
            }
            // Exit loop if escape is pressed
            if((kbhit())&&(getch() == 27)) running = FALSE;

      }while(running);
      printf("\n\n");
}
```

Interrupts

The OX16C95x device can be configured to generate interrupts on the events listed below. (All devices have a separate interrupt line for each UART channel, with the exception of the 954 in Motorola mode).

- Line status errors (parity, framing etc.) Priority 1
- Received data Priority 2a
- Received data timeout (data available for more than four character times) Priority 2b
- Space available for data to transmit Priority 3
- Modem status (change in CTS, DCD etc.) Priority 4
- XOFF detection in in-band flow control Priority 5
- Special character detection Priority 5
- 9th Bit set in nine-bit data mode (OX16C950B only) Priority 5
- CTS Change of state (for 650 compatibility – used to monitor out-of-band flow control) Priority 6
- RTS Change of state (as above) Priority 6

To use an interrupt, the relevant enable bit must be written to IER and the interrupt pin enabled (see below). Once this is done, occurrence of an enabled event will result in the interrupt pin being asserted, and the interrupt status register (ISR) being updated to reflect the **highest priority** interrupt currently pending (where priority 1 is the highest). See data sheet section 10.2 for a description of ISR contents when reporting interrupts.

If the INT_SEL# pin is tied low, bit 3 of MCR (OUT2) is used to enable the interrupt output pin for the device (MCR[3]=1 enables interrupts, MCR[3]=0 disables interrupts), as well as setting the state of the OUT2 pin. If the INT_SEL# pin is tied high however, the interrupt line is permanently enabled and MCR[3] does not affect it.

The following definitions may be useful in enabling the above interrupts:

```
#define IER_OFFSET                 1
#define IER_RX_INTERRUPT_EN        0x01
#define IER_TX_INTERRUPT_EN        0x02
#define IER_LSTAT_INTERRUPT_EN     0x04
#define IER_MSTAT_INTERRUPT_EN     0x08
#define IER_CHR_INTERRUPT_EN       0x20
#define IER_RTS_INTERRUPT_EN       0x40
#define IER_CTS_INTERRUPT_EN       0x80
```

For example, to enable receiver and transmitter interrupts:

```
DEVINFO device;
      :
      :
Write(&device, IER_OFFSET, IER_RX_INTERRUPT_EN | IER_TX_INTERRUPT_EN);
```

Standard FIFO Trigger Levels

Receiver data available and transmitter space available interrupts can be triggered at various FIFO fill levels. The configuration options for these levels vary depending on the operating mode of the UART. Theses options are summarised in the following table, but first is a definition of the two trigger levels:

- **Receiver Trigger Level**
  The number of characters to be transferred to the receiver FIFO before a receiver data available interrupt is asserted.
- **Transmitter Trigger Level**
  When the number of characters in the transmit FIFO falls *below* this value, a transmitter interrupt is asserted.

UART Mode	FIFO Size	Receiver Trigger Level Options	Transmitter Trigger Leve Options
450	1	1	1
550	16	1,4,8,14	1
Ext. 550	128	1,32,64,112	1
650	128	16,32,112,120	16,32,64,112*
750	128	1,32,64,112	1
950	128	1 to 128	0 to 128

Table 8: FIFO Trigger Levels

* To enable 650 compatible transmit trigger levels, FCR[3] must also be set. Otherwise the trigger level defaults to 1.

In the case of the 550, extended 550, 650 and 750 modes, the four options are configured using FCR bits 6 and 7. For example, in 550 mode FCR[6:7] = 00 gives a trigger level of 1, 01 gives 4, 10 gives 8 etc. In 650 mode, this is also true of the transmit trigger level, which is set using FCR[4:5]. The 950 mode offers fully configurable trigger levels that are discussed in more detail in section 0.

The following definitions can be used for configuring standard trigger levels:

```
#define FCR_DMA_MODE                0x08 // Set to use 550 Tx Trigger Levels

#define FCR_RX_TRIGGER_OPT1         0x00
#define FCR_RX_TRIGGER_OPT2         0x40
#define FCR_RX_TRIGGER_OPT3         0x80
#define FCR_RX_TRIGGER_OPT4         0xC0

#define FCR_TX_TRIGGER_OPT1         0x00
#define FCR_TX_TRIGGER_OPT2         0x10
#define FCR_TX_TRIGGER_OPT3         0x20
#define FCR_TX_TRIGGER_OPT4         0x30
```

For example, to set Rx trigger to 32 and Tx trigger level to 16 in 650 mode, use the following:

```
DEVINFO device;
      :
      :
Write(&device,FCR_OFFSET,FCR_RX_TRIGGER_OPT2|FCR_TX_TRIGGER_OPT1
|FCR_DMA_MODE);
```

Using Enhanced Features

Flexible Baud Rate Generation (Using TCR and CPR)

The 16C450 and 550 devices use the following equation to derive a baud rate from the system UART clock:

$$BaudRate = \frac{InputClkFrequency}{BaudRateDivisor \times 16}$$

Because this system uses 16 system clocks per serial bit, the maximum baud rate is limited to a sixteenth of the input clock frequency. The OX16C95x offers extended flexibility of baud rate by introducing two new parameters into the equation, the clock prescaler register (CPR) and the times clock register (TCR). (These are both ICR registers and reside at indexes 1 and 2 respectively – see section 0 for details on how to write to ICR registers).

$$BaudRate = \frac{InputClkFrequency}{BaudRateDivisor \times TCR \times PRESCALER}$$

Where TCR is the value in the TCR register (16 or 4)* and PRESCALER is the value in the CPR register (8 to 255) divided by 8. (*PRESCALER* can therefore range from 1 to 31.875 in steps of 0.125).

The TCR facility allows the option to quadruple the baud rate by using a minimum of 4 system clocks per bit as opposed to 16. The prescaler option allows non-standard frequency UART clocks to be scaled down to standard speeds (e.g. 1.8432 MHz) for compatibility, while maintaining the option for high speed operation when the prescaler is disabled (i.e. CPR = 8 so prescaler = 1).

* The OX16C950B offers all TCR values in the range 4 to 16 for even more flexible operation.

Enabling The Clock Prescaler Register (CPR)
The prescaler is enabled by setting bit 7 of the MCR register, which is only accessible in enhanced mode (when EFR bit 4 is set). This can therefore be achieved using the following function:

```
void SetPrescalerEnable(PDEVINFO device, BOOL state){
      BYTE efr, mcr;
      // Store EFR and enable enhanced mode
      efr = Read650(device, EFR_OFFSET);
      Write650(device, EFR_OFFSET, (BYTE)(efr | EFR_ENHANCED_MODE_EN));
      // Get current MCR value
      mcr = Read(device, MCR_OFFSET);
      // Set the bit according to the state requested
      if(state) mcr |= MCR_PRESCALER_EN; else mcr &= ~MCR_PRESCALER_EN;
      // Write new value and restore EFR
      Write(device, MCR_OFFSET, mcr);
      Write650(device, EFR_OFFSET, efr);
}
```

When the prescaler is disabled, it is bypassed and has no effect. At power up the prescaler enable bit in MCR is set to the complement of the CLKSEL pin *, therefore tying this pin low will force the prescaler to be enabled on power up / reset.

The reset state of the CPR register is 0x20 (divide by 4). Therefore the OX16C95x device with CLKSEL tied low can use a 7.3728 MHz clock in place of a 1.8432MHz device, and maintain compatibility with existing software. A driver may then disable the prescaler by clearing MCR[7] (overriding the value set by the CLKSEL pin) to achieve a four times increase in baud rate.

NOTE: If a 1.8432MHz crystal is to be used and compatibility is required, CLKSEL* should be tied high in order to ensure the prescaler is *disabled* on power up and reset.

The following table gives the prescaler values required for compatibility mode for various popular crystal frequencies (i.e. the prescaler required to scale the clock down to 1.8432MHz). Also given is the maximum available baud rates in TCR = 16 and TCR = 4 modes.

Clock Frequency (MHz)	Compatibility Mode Divisor	CPR value	Error (%)	Max. Baud rate (TCR = 16)	Max. Baud rate (TCR = 4)
1.8432	1.000	0x08	0.00	115,200	460,800
7.3728	2.000	0x10	0.00	460,800	1,843,200
14.7456	8.000	0x80	0.00	921,600	3,686,400
18.432	10.000	0x50	0.00	1,152,000	4,608,000
32.000	17.375	0x8B	0.08	2,000,000	8,000,000
33.000	17.875	0x8F	0.16	2,062,500	8,250,000
40.000	21.75	0xAE	0.22	2,500,000	10,000,000
50.000	27.125	0xD9	0.01	3,125,000	12,500,000
60.000*	31.875	0xFF	2.12	3,750,000	15,000,000

Table 9: Example clock options and their assosiated maximum baud rates
* 60MHz clock is only available on the OX16C950B

Using the Times Clock Register (TCR)
The TCR register is used to set the number of channel (internal) clocks per serial bit. In previous devices this has been fixed at 16. The OX16C954 allows this value to also be set to 4 to quadruple the maximum baud rate available with any given system clock. The OX16C950B allows any value in the range 4-16 for even greater flexibility.

NOTES:

1. Writing 16 to TCR actually stores the value 0x00 in the register, which corresponds to 16 clocks per bit. This is the power up / reset state of TCR.

2. TCR is always enabled, all that is required to change it is a write to it with the new value (TCR is located at index 2 of the indexed control register set, see section 0).

3. It is recommended that TCR values other than 16 are only used when baud rates higher than the maximum available at TCR = 16 are required for any given system clock. e.g. Use a divisor of 1 and TCR = 4 to enable 460.8 kBaud with a 1.8432MHz clock. For the same baud rate with a 7.3728MHz clock however, use a divisor of 1 and TCR=16 in favour of a divisor of 4 with TCR=4.

4. TCR=4 can be used to achieve lower power consumption for a given baud rate because a system clock which is four times slower can be employed to achieve the same results.

Using 950 Trigger Levels

The OX16C95x has totally configurable trigger levels for receiver and transmitter interrupts as well as configurable flow control XON and XOFF points. These trigger levels are set using a group of four ICR set registers, and enabled by setting bit 5 of ACR. The functionality of these registers is summarised in the table below:

Register	ICR Index	Description	Valid values
TTL (Transmitter interrupt trigger level)	0x04	When the Tx FIFO fill level drops below this value, a transmitter interrupt occurs	0-128[1,2]
RTL (Receiver interrupt trigger level)	0x05	When the Rx FIFO fill level reaches this value, a receiver interrupt occurs	1-128[1,3]
FCL (Lower flow control threshold)	0x06	The receiver FIFO level at which the UART signals the remote transmitter to start transmitting (e.g. sends XON)	0-128[4]
FCH (Upper flow control threshold)	0x07	The receiver FIFO level at which the UART signals the remote transmitter to stop transmitting (e.g. sends XOFF)	1-128[4]

Table 10: 950 Trigger Level Registers

1. Interrupts must be enabled for these to be asserted on the interrupt pin (see section 0)
2. Setting TTL=0 is a special case whereby the transmitter interrupt is not triggered until the shift-register, as well as the FIFO, are empty (i.e.. the transmitter is idle).
3. RTL=0 must be avoided or a receiver interrupt will be present when no data is available. All these registers are however reset to zero, hence this register must be programmed before 950 trigger levels are enabled.
4. These registers only have an effect when automatic flow-control (in band or out of band) is enabled

The levels themselves can be set using the `WriteICR` function already defined in section 0. The following function however provides a cleaner interface for enabling these registers:

```
#define ACR_950_TRIGGER_EN 0x20

void Set950TriggerEnable(PDEVINFO device, BOOL state){
      // Set the bit according to the state requested
      if(state)
            device->shadowACR |= ACR_950_TRIGGER_EN;
      else
            device->shadowACR &= ~ ACR_950_TRIGGER_EN;
      // Write new value
      WriteICR(device, ACR_INDEX, device->shadowACR);
}
```

NOTE: When 950 trigger levels are enabled, trigger levels set in the FCR register are overridden.

Enabling and Disabling the Transmitter and Receiver

The transmitter and receiver can be enabled and disabled independently using the bottom two control bits in ACR (Index 0x00 of the ICR set). Setting bit 0 will disable the receiver, setting bit 1 will disable the transmitter. The following two simple functions provide a more clear interface by which to achieve this.

```
#define ACR_RX_DISABLE  0x01
#define ACR_TX_DISABLE  0x02

void SetReceiverEnable(PDEVINFO device, BOOL state){
      // Set the bit according to the state requested
      if(state)
            device->shadowACR |= ACR_RX_DISABLE;
      else
            device->shadowACR &= ~ACR_RX_DISABLE;
      // Write new value
      WriteICR(device, ACR_INDEX, device->shadowACR);
}
```

```
void SetTransmitterEnable(PDEVINFO device, BOOL state){
      // Set the bit according to the state requested
      if(state)
            device->shadowACR |= ACR_TX_DISABLE;
      else
            device->shadowACR &= ~ACR_TX_DISABLE;
      // Write new value
      WriteICR(device, ACR_INDEX, device->shadowACR);
}
```

NOTE: Changes to these bits are not recognised until the current character being received (in the case of bit 0) or transmitted (in the case of bit 1) is complete. Note also, that in-band flow control characters may still be received and transmitted in any state.

Also note that on 0X16C950B devices, if data is written to THR directly after disabling the transmitter, one character may be allowed to escape before the transmitter is disabled. In this case it is necessary to poll for Tx idle (LSR[6]) immediately after disabling the transmitter, and not write to THR until this bit is high.

Using Automated Out-of-band Flow Control

The OX16C95x device can be configured to automatically generate its own flow control signals and responses using the CTS, RTS, DSR and DTR pins. Each pin can be enabled individually. The definition of what each does is given below.

- **Automatic CTS or DSR flow control:**

The CTS/DSR input pins are used to enable and disable the transmitter. Transmission is disabled when then pin is held high and enabled when it is held low. These pins are normally connected to RTS and DTR respectively on the remote receiver.

- **Automatic RTS or DTR flow control:**

The fill level of the receiver FIFO controls the RTS/DTR output pin. When this level reaches an upper flow control threshold, the pin is asserted to disable the remote transmitter. The pin is not then de-asserted until the receiver FIFO is read to a level equal to or below the lower flow control threshold.

In 950 mode, these thresholds are defined by the FCH (upper threshold), and FCL (lower threshold) registers in the ICR (See section 0). For thresholds in other modes, refer to the device data sheet section 8.1.

For readability separate functions to enable/disable the use of each pin for automatic flow control are given below.

```
#define EFR_AUTO_RTS_EN 0x40
#define EFR_AUTO_CTS_EN 0x80
#define ACR_AUTO_DSR_EN 0x04
#define ACR_AUTO_DTR_EN 0x08


void SetAutoCTSEnable(PDEVINFO device, BOOL state){
      // Sets the state of automatic CTS flow control enable bit to state
      BYTE efr = Read650(device, EFR_OFFSET);
      // Set the bit according to the state requested
      if(state) efr |=  EFR_AUTO_CTS_EN;
      else      efr &= ~EFR_AUTO_CTS_EN;
      // Write new value
      Write650(device, EFR_OFFSET, efr);
}


void SetAutoRTSEnable(PDEVINFO device, BOOL state){
      // Sets the state of automatic RTS flow control enable bit to state
      BYTE efr = Read650(device, EFR_OFFSET);
      // Set the bit according to the state requested
      if(state) efr |=  EFR_AUTO_RTS_EN;
      else      efr &= ~EFR_AUTO_RTS_EN;
      // Write new value
      Write650(device, EFR_OFFSET, efr);
}

void SetAutoDSREnable(PDEVINFO device, BOOL state){
      // Sets the state of automatic DSR flow control enable bit to state
      if(state) device->shadowACR |=  ACR_AUTO_DSR_EN;
      else      device->shadowACR &= ~ACR_AUTO_DSR_EN;
      // Write new value
      WriteICR(device, ACR_INDEX, device->shadowACR);
}


void SetAutoDTREnable(PDEVINFO device, BOOL state){
      // Sets the state of automatic DTR flow control enable bit to state
      if(state) device->shadowACR |=  ACR_AUTO_DTR_EN;
      else      device->shadowACR &= ~ACR_AUTO_DTR_EN;
      // Write new value
      WriteICR(device, ACR_INDEX, device->shadowACR);
}
```

NOTE: Automatic DTR flow control can not be used if DTR is configured for BDOUT or 1x Tx CLK in the CKS register, or if the RS-485 buffer enable bit is set in ACR, as these features override the functionality of the DTR pin.

Using Automated In-band Flow Control

The OX16C95x also supports automated in-band flow control, using XON and XOFF characters transmitted by the remote receiver to disable/enable transmission accordingly. This operates on the same principle as out-of-band flow control defined in the previous section. Two categories of in-band flow control can be enabled:

- **Automatic in-band receive flow control:**
  When an XOFF character is received from the remote receiver, transmission is disabled until an XON is received.

- **Automatic in-band transmit flow control:**
  XON and XOFF characters are sent back to the transmitter according to the fill levels of the receiver FIFO. As with out-of-band flow control, XOF is sent when the upper flow control threshold is reached, and XON is sent when the receiver is read to a level equal to or below the lower threshold.

The are 10 different modes in which in-band flow control can be configured, using different combinations of the 4 XON and XOFF characters stored in the 650 compatible register set. To make programming easier, these are listed in the table below, complete with the required value to write to the lower half of the Enhanced Features Register EFR.

Mode	Transmit Mode	Receive Mode	EFR[0:3] Value
0	Disabled	Disabled	0000 (0x0)
1		XON1/XOFF1	0010 (0x2)
2		XON2/XOFF2	0001 (0x1)
3	XON1/XOFF1	Disabled	1000 (0x8)
4		XON1/XOFF1	1010 (0xA)
5		XON2/XOFF2	1001 (0x9)
6		XON1 or 2/XOFF1 or 2	1011 (0xB)
7	XON2/XOFF2	Disabled	0100 (0x4)
8		XON1/XOFF1	0110 (0x6)
9		XON2/XOFF2	0101 (0x5)
10		XON1 or 2/XOFF1 or 2	0111 (0x7)

Table 11: In-Band Flow Control Modes

Any of the listed modes can be configured using the simple lookup table function below:

```
void SetInBandFlowControlMode(PDEVINFO device, BYTE mode){
      // Sets the automatic inband flow control mode to the
      // specified mode index in the table above
      BYTE
modeTable[11]={0x0,0x02,0x01,0x08,0x0A,0x09,0x0B,0x04,0x06,0x05,0x07};
      BYTE efr = Read(device, EFR_OFFSET) & 0xF0;
      Write(device, EFR_OFFSET, (BYTE)(efr | modeTable[mode]));
}
```

NOTES:

1. XON/XOFF characters should be written to the appropriate registers prior to enabling in-band flow control. For more information on setting flow control characters, see section 0.

2. Additionally, when using in-band receive flow control, setting bit 5 of MCR will enable XON-Any mode. This treats any received character as a valid XON character before transferring it to the receiver FIFO. In all other modes, XON/XOFF characters are stripped form the received data stream and are invisible to the user.

In-Band Flow Control Status
Various facilities exist for determining the status of in-band flow control operation. These are summarised here:

- Bit 0 of ASR reflects in-band receive flow control status (i.e.. the current state of the transmitter). ASR[0]=0 indicates that the transmitter is enabled as normal. ASR[0]=1 indicates that the transmitter has been disabled by a received XOFF character.
- Bit 1 of ASR reflects in-band transmit flow control status (i.e.. the current state of the remote transmitter). ASR[0]=0 indicates that the remote transmitter is enabled as normal. ASR[0]=1 indicates that the remote transmitter has been disabled by sending an XOFF character to it.
- Bit 4 of the interrupt status register will set every time an XOFF character is received and cleared when an XON is received (the same as ASR[0]). An interrupt can also be generated on this event by setting bit 5 of the interrupt enable register IER.

Using Special Character Detection
The OX16C95x offers a facility to generate interrupts upon the reception of a given special character. To enable this feature the following steps are required:

- The device must be in enhanced mode (EFR[4]=1)
- The special character to detect must be loaded into the XOFF2 location in the 650 Compatible Register set
- Special character detection must be enabled (EFR[5]=1)
- IER[5] must be set to enable the interrupt

When a special character is received, a level five interrupt is generated (ISR[4:0] = 10000b). It must then be verified that this is indeed a special character and not a normal XOFF (which shares the same interrupt priority) by reading ASR bit 4. This bit will only be set if a true special character was received. The following functions simplify these operations:

```
#define EFR_SPECIAL_CHAR_EN  0x20
#define ASR_SPECIAL_CHAR_DET 0x10

void SetSpecialCharDetectEnable(PDEVINFO device, BOOL state, BYTE character){
      BYTE efr, ier;
      // Enable enhanced mode and special character detection
      // Note: when called with state=FALSE enhanced mode in NOT disabled
      efr = Read650(device, EFR_OFFSET);
      ier = Read(device, IER_OFFSET);
      if(state){
            // Also enable Enhanced mode if necessary
            efr |= (EFR_ENHANCED_MODE_EN | EFR_SPECIAL_CHR_EN);
            ier |= IER_CHR_INTERRUPT_EN;
            Write650(device, XOFF2_OFFSET, character);
      }else{
            // Only turn off special char detect bit when disabling
            efr &= ~EFR_SPECIAL_CHR_EN;
            ier &= ~IER_CHR_INTERRUPT_EN;
      }
      Write650(device, EFR_OFFSET, efr);
      Write(device, IER_OFFSET, ier);
}


BOOL CheckSpecialChar(PDEVINFO device){
      // Get the special character detection
      // indication bit from ASR to verify special character
      return ReadASR(device) & ASR_SPECIAL_CHR_DET;
}
```

NOTES:

1. Parity and framing do not have to be valid for a special character to be recognised
2. The OX16C950B device offers more advanced special character detection in nine-bit data mode.

Transmitting and Receiving Nine-bit Data (OX16C950 only)

The single channel UART allows an additional nine-bit data mode of operation for specialist multi-drop applications. This is enabled by setting the bottom bit of the Nine bit Mode Register (NMR) at index 0x0D in the ICR set. In this mode the data length set in LCR[0:1] is ignored and parity is disabled (Hence LCR[5:3] are also ignored). Bit 1 of NMR can also be set to enable interrupt generation on reception of data with the 9th bit set (see 950 data sheet section 15.9 for more details).

The following function can be used to enable/disable nine-bit mode. An additional Boolean parameter allows for the setting of the 9th bit interrupt enable.

```
#define NMR_9BIT_MODE_EN      0x01
#define NMR_9BIT_INTERRUPT_EN 0x02

void Set9BitModeEnable(PDEVINFO device, BOOL state, BOOL bit9int){
      BYTE nmr = ReadICR(device, NMR_INDEX);
      if(state)
            // We are enabling feature
            if(specialInt) nmr |= (NMR_9BIT_MODE_EN + NMR_9BIT_INTERRUPT_EN);
            else           nmr &=~(NMR_9BIT_INTERRUPT_EN);
      else
            // Clear both bits if we are disabling
            nmr &= ~ NMR_9BIT_MODE_EN;

      // Write new value
      WriteICR(device, NMR_INDEX, nmr);
}
```

The following functions are examples of how to encapsulate the read and write operations that require access to a second register for the ninth bit of each character.

```
void Send9BitData(PDEVINFO device, WORD data){
      // Sends bottom 8-bits to THR and top bit to SPR
      // for 9-bit mode transmission
      BYTE lsb = data & 0x00FF;
      BYTE msb = (data & 0x0100) >> 8;
      Write(device, SPR_OFFSET, msb);
      Write(device, THR_OFFSET, lsb);
}

WORD Receive9BitData(PDEVINFO device, BYTE *lsr){
      // Receive bottom 8-bits from RHR and the 9th bit
      // from LSR[2] - also returns lsr for error checking
      WORD data;
      *lsr = Read(device, LSR_OFFSET);
      data = Read(device, RHR_OFFSET);
      // Set bit 9 of data if LSR[2] is set
      if(*lsr & LSR_PARITY_ERROR) data |= 0x0100;
      return data;
}
```

This mode also provides for more sophisticated special character detection, allowing for the detection of up to four individual special characters*. (This is possible because automatic in-band flow control is not available in this mode, and hence the XON/XOFF character registers are free to be used). The following function allows these characters to be specified using an index `charNum` (1 to 4) and a 16-bit word for the character (of which only the bottom 9-bits are used).

```
void Set9BitSpecialChar(PDEVINFO device, BYTE charNum, WORD chr){
      BYTE nmr;
      chr &= 0x01FF; // Mask off to 9bits only
      if((charNum > 4)||(charNum < 1)) return;
      charNum--;
      // Write the lower 8-bits of the special character
      Write650(device, (BYTE)(XON1_OFFSET + charNum), (BYTE)(chr & 0x00FF));
      // Write the top bit into its appropriate NMR location
      chr = chr >> 8; // (Either 0 or 1)
      nmr = ReadICR(device, NMR_INDEX);
      nmr |= chr << (charNum + 2);
      WriteICR(device, NMR_INDEX, nmr);
}
```

This final function, which must only be called after 9bit mode has been enabled, enables the detection of special characters set with the previous function. Now when a special character is transferred to the receiver FIFO, a level 5 interrupt will be generated.

```
void Set9BitSpecialCharDetectEnable(PDEVINFO device, BOOL state){
      BYTE ier = Read(device, IER_OFFSET);
      BYTE efr = Read650(device, EFR_OFFSET);
      // Set the enable bit according to the state requested
      if(state){
            ier |= IER_CHR_INTERRUPT_EN;
            // Enhanced mode must be enabled first
            Write650(device, (BYTE)(EFR_OFFSET, efr | EFR_ENHANCED_MODE_EN));
      }else ier &= ~ IER_CHR_INTERRUPT_EN;
      // Write new value
      Write(device, IER_OFFSET, ier);
}
```

- To identify which character has been detected it must be read in using the read data function provided.

Data Transfer Using an Isochronous Clock

All OX16C95x devices allow for an Isochronous mode of operation whereby data can be received and transmitter using a 1x clock (i.e.. baud rate = clock frequency). However, the way in which this mode of operation is enabled differs for the various devices. The devices are therefore discussed separately.

OX16C950

This device is the more versatile of the set, with a register (CKS in the ICR set) dedicated to clock configuration. For a full discussion of this register, refer to section 15.8 of the 950 data sheet. The 950 device allows for different clocks to be used for the transmitter and the receiver. This is enabled by configuring the DTR pin as a 1x transmitter clock output and sending the clock to the receiving device. Assuming the receiving device is also an OX16C950, this clock signal can then be input on the DSR pin and used to drive the receiver.

This allows data rates of anything up to the maximum crystal frequency to be obtained. (i.e.. up to 60Mbps). Note however, that because the clock signal is being sent down the line, a suitable line protocol and driver must be selected in order to pass the high frequency clock without attenuation.

To configure this mode of operation, we set the following options in CKS:

- DSR configured as receiver clock source (CKS[0:1] = 01)
- Receiver set to isochronous mode (CKS[3] = 1)
- DTR configured as bit rate transmitter clock output (CKS[5:4] = 01). This overrides ACR[4:3]*
- Transmitter set to isochronous mode (CKS[7] = 1)

* As selecting this mode uses the DTR pin all other DTR configurations are overridden. This includes RS-485 buffer enabling and automatic DTR flow control.

This can therefore be set up using the following code:

```
#define CKS_ISOCHRONOUS_MODE_EN 0x9D

void Set950IsochronousEnable(PDEVINFO device, BOOL state){
      // Turns isochronous mode on or off in the 950
      if(state)
            WriteICR(device,CKS_INDEX, CKS_ISOCHRONOUS_MODE_EN);
      else
            WriteICR(device,CKS_INDEX, 0);
}
```

NOTE: The above constant also sets CKS[2] to disable the BDOUT pin. This is not necessary but is recommended to reduce noise and power consumption.

OX16C954
This device offer a slightly less flexible isochronous mode, whereby the transmitter and receiver are both clocked by the same signal, which must be applied to the RI pin of both the transmitting device and the receiving device.

Isochronous mode is enabled in these devices by setting CPR to the reserved value of 0x00 in enhanced mode. This can be achieved with the following code:

```
#define CPR_ISOCHRONOUS_MODE_EN 0x00

void Set954IsochronousEnable(PDEVINFO device, BOOL state){
      // Turns isochronous mode on or off in the 954
      BYTE efr = Read650(device, EFR_OFFSET);
      BYTE mcr = Read(device, MCR_OFFSET);
      if(state){ // Set required bits to turn on
            efr |= EFR_ENHANCED_MODE_EN;
            mcr |= MCR_PRESCALER_EN;
      }else{ // Clear only the CPR enable to turn off
            mcr &=~MCR_PRESCALER_EN;
      }
      // Write new values
      Write650(device, EFR_OFFSET, efr);
      WriteICR(device,CPR_INDEX, CPR_ISOCHRONOUS_MODE_EN);
      Write(device, MCR_OFFSET, mcr);
}
```

Configuring DMA Transfer Signalling
The TXRDY and RXRDY pins on the OX16C95x device can be used for direct memory access (DMA) control signals. Each UART channel has its own TXRDY/RXRDY pair. For pin compatibility with earlier devices however, the OXC16C954 performs a logical OR of these outputs before presenting them as a single pair of pins on the ICs package.[1]

The basic function of these signals is as follows:

- TXRDY becomes active when there is room for more data in the transmitter
- RXRDY becomes active when there is a given amount of data to be read from the receiver

The mode in which the FIFO is operating determines the point at which these signals change state. The following table gives details:

FIFO mode	**TXRDY**		**RXRDY**	
	Active when…	In-active when…	Active when…	In-active when…
DMA mode 0 or Byte mode	Transmit FIFO is empty	Transmit FIFO contains data	Receiver FIFO contains data	Receiver FIFO is empty
DMA mode 1	Transmit FIFO is not full	Transmit FIFO is full	Receiver FIFO level reaches RTL or a receiver timeout occurs*	Receiver FIFO is empty

Table 12: DMA Signalling Control Signal Definition
* RTL is defined as the receiver interrupt trigger level. See section 0 for how RTL is set in the various different modes of operation.

Notice from the above, that RXRDY operates with hysteresis when in DMA mode 1. The RXRDY signal becomes active when the trigger is reached, but is not de-asserted again until all the data has been read.

Enabling DMA signalling is simply a matter of setting the required configuration in the FIFO control register (FCR) using the FIFO enable bit FCR[0] and DMA mode bit FCR[3], as shown below:

Mode	FCR[3]	FCR[0]
DMA mode 0	0	X
DMA mode 1	1	1

Table 13: DMA Mode Configuration

Configuring Automatic RS-485 Buffer Enabling

In systems using the RS-485 protocol, the OX16C95x device can be configured to provide an automatic buffer enable signal (on the DTR pin) used to switch line drivers in and out of their tri-state mode. The pin can be configured either active-high or active-low and the pin is active only when the transmitter contains data. I.e. the buffers are enabled all the time the transmitter is sending data and disabled whenever it is idle. The control for the DTR pin in this mode is actually derived directly from LCR[6], the transmitter empty bit.

To use DTR for this purpose bit 4 of ACR must be set, bit 3 then controls the sense. The following function provides this functionality.

```
#define ACR_RS485_HIGH_EN 0x18
#define ACR_RS485_LOW_EN  0x10

void SetRS485BufferEnable(PDEVINFO device, BOOL state, BOOL activeHigh){
      if(state)
            // We are enabling feature – decide on pin sense
            if(activeHigh) device->shadowACR |= ACR_RS485_HIGH_EN;
            else device->shadowACR |= ACR_RS485_LOW_EN;
      else
            // Clear both bits if we are disabling
            device->shadowACR &= ~ ACR_RS485_HIGH_EN;

      // Write new value
      WriteICR(device, ACR_INDEX, device->shadowACR);
}
```

Enabling Sleep-Mode

The OX16C95x provides a sleep mode option for lower power consumption when idle. This option has two possible configurations, one for 950 & 650 modes (normal sleep mode) and one for 750 compatible mode (alternative sleep mode). The functions below can be used to enable sleep mode:

Use this function when operating in enhanced mode (EFR[4]=1) i.e.. 650 or 950 modes:

```
#define IER_SLEEP_MODE_EN     0x10
#define IER_ALT_SLEEP_MODE_EN 0x20

void SetSleepModeEnable(PDEVINFO device, BOOL state){
      // Sets the state of the sleep mode enable bit to state
      // will not work while LCR[7] or ACR[7] are set
      BYTE ier = Read(device, IER_OFFSET);
      // Set the bit according to the state requested
      if(state) ier |=  IER_SLEEP_MODE_EN;
      else      ier &= ~IER_SLEEP_MODE_EN;
      // Write new value
      Write(device, IER_OFFSET, ier);
}
```

Use this function when not operating in enhanced mode (EFR[4]=0) i.e.. 750 mode

```
void SetAltSleepModeEnable(PDEVINFO device, BOOL state){
      // Sets the state of the sleep mode enable bit to state
      // will not work while LCR[7] or ACR[7] are set
      BYTE ier = Read(device, IER_OFFSET);
      // Set the bit according to the state requested
      if(state) ier |=  IER_ALT_SLEEP_MODE_EN;
      else      ier &= ~IER_ALT_SLEEP_MODE_EN;
      // Write new value
      Write(device, IER_OFFSET, ier);
}
```

The following function can be used to check that a sleep mode enable request was successful. It will return TRUE if the device is in the sleeping state (see data sheet section 10.4 for sleep mode conditions).

```
BOOL CheckSleeping(PDEVINFO device){
      // Returns TRUE if the specified deive is asleep
      // will not work while LCR[7] or ACR[7] are set
      BYTE ier = Read(device, IER_OFFSET);
      return ier & (IER_SLEEP_MODE_EN | IER_ALT_SLEEP_MODE_EN);
}
```

NOTE: On the OX16C954 multi channel device, each channel can be put into sleep mode independently. Therefore, to achieve minimal power consumption, all channels should be put into sleep mode.

Contact Details

Oxford Semiconductor Ltd.
25 Milton Park
Abingdon
Oxfordshire
OX14 4SH
United Kingdom

Telephone:	+44 (0)1235 824900
Fax:	+44 (0)1235 821141
Sales e-mail:	sales@oxsemi.com
Web site:	http://www.oxsemi.com

DISCLAIMER