OXFORD SEMICONDUCTOR

OX16PCI954

Integrated Quad UART with PCI interface.

APPLICATION NOTES

This application note presents guidelines for hardware implementation and configuration of the Oxford Semiconductor OX16PCI954 Quad UART with PCI interface in various PC add-in card configurations.

Version 1.2 (11 January 2000)

PCI LOCAL BUS

Designed for Microsoft® Windows NT® Windows®98

CONTENTS

1 INTRODUCTION

This document describes example applications for the OX16PCI954 Integrated Quad UART and PCI interface. It provides design engineers with sufficient information and examples to integrate the device into a serial/parallel peripheral application such as a PC add-in card. Examples are given of all aspects of interfacing and using the device, however for detailed specifications the reader should refer to the data sheet.

In the sample applications, some basic knowledge of serial communications and PCI architecture is assumed, but any features particular to the OX16PCI954 are explained in detail.

The document is structured in such a way as to build up familiarity with the device, and provide a useful reference manual for electronic design engineers. In Section 2, a brief overview of the device is given. Section 3 continues with a description of how to access the various functions of the chip via the PCI interface. Section 4 provides examples of how to interface the device with associated interconnects and components, and then Section 4.6 provides whole application examples for many typical applications using this chip. Section 6 deals with programming the four OX16C950 UARTs which comprise the device's major function, and finally, Section 8 describes programming of the advanced PCI-specific features available.

2 Device Overview

The OX16PCI954 is a single chip solution for PCI-based serial and parallel expansion cards. It has various modes of operation, which can provide a combination of four high-performance serial ports, a bidirectional parallel port, and an 8/32 bit Local expansion Bus, operated from a multifunction PCI interface. The device provides a simple means of designing PCI serial boards with up to 20 serial ports, low-cost 4-port PCI serial boards, or combo serial/parallel boards.

The device is a dual-function PCI target, where function 0 offers four high-peformance OX16C950 UARTs, and function 1 is configurable to offer either an 8/32 bit pass-through Local Bus or a bi-directional parallel port. The desired functions are selected via two ‘Mode’ pins, and then operation is performed through standard I/O or memory mapping on the PCI interface. For many applications, the OX16PCI954 and serial line drivers will be the only components necessary; however for more complex solutions many of the device default registers are reconfigurable from an optional serial EEPROM.

Therefore, to design a combo parallel / four-port serial card, the designer need to enable the four UARTs and the parallel port, and make the necessary connections to the PCI bus and line drivers. If the parallel port is not required, the pins can be reconfigured to assign the PCI Subsystem ID and Subsystem Vendor ID.

Alternatively, boards with up to 20 serial ports can be produced using the local bus function. In this case the designer should enable the four internal UARTs, and then add external UART devices onto the local bus, which can be configured to either Intel or Motorola-type operation.

In one further mode of operation, the internal UARTs can be disabled to allow their pins to be reconfigured to provide a full 32-bit pass through interface.

3 Configuration and Operation

3.1 Mode selection

The Mode[1:0] pins are used to select which of the logical functions are enabled behind the PCI interface. Table 1 describes the configuration mapping used.

Mode [1:0]	Configuration
00	Function 0 is Quad UART, Function 1 is 8-bit Local Bus
01	Function 0 is Quad UART, Function 1 is bidirectional parallel port
10	Function 0 is Quad UART, Function 1 is unusable as the local bus pins are used to assign Subsystem ID and Subsystem Vendor ID to Function 0.
11	Function 0 is unusable, Function 1 is 32-bit Local Bus

Table 1: Mode configuration

3.2 Accessing logical functions

Operation of the UARTs, local bus and parallel port is implemented through standard I/O and memory address mapping. The configuration space for each function contains a number of Base Address Registers (BARs), which will be initialized by the system BIOS or Plug and Play operating system. Figure 1 (on page 8) shows the BAR mapping for the UARTs, 8-bit local bus and parallel port. The Plug and Play architecture automatically sets up the BARs in all PCI devices so that no address ranges overlap. Once they are set up, PCI I/O or memory reads to the specified locations will be directed to the respective function. Drivers need to read the value from configuration space (or the interface in the operating system) and direct accesses to the correct address.

Each function has its own unique address, however the local configuration registers can be accessed through either function. This enables multi-port drivers to easily snoop accesses and/or adjust the parameters for either function.

Mapping for the local bus can be reconfigured to suit the external devices connected. The size of the block is variable between 4 and 256 bytes of I/O space. The number of active address lines (maximum 8) depends on the block size selected. For 8-bit local bus, the memory block size is fixed at 4kb, however this can be increased to 16kb in 32-bit mode, where more address lines are available.

Table 2 shows which functions accept byte, word and dword accesses via I/O and memory mapping.

PCI Function	I/O accesses			Memory accesses		
	Byte	Word	Dword	Byte	Word	dword
UARTs	✓	X	X	✓	X	X
8-bit Local Bus	✓	X	X	✓	X	X
32-bit Local Bus	✓	✓	✓	✓	✓	✓
Parallel port	✓	X	X	N/A	N/A	N/A
Local config registers	✓	X	X	✓	✓	✓

Table 2: Accessing logical functions

Complete examples of applications and device register mapping are given in Section 4.6, since access to the local bus devices will vary between applications. Access to the local configuration registers is constant with application and which logical function is used, so a register map is given overleaf in Table 3.

Register	Offset from Base Address 2 in I/O space (hex)	Offset from Base Address 3 in memory space (hex)
LCC	0x00	0x00
MIC	0x04	0x04
LT1	0x08	0x08
LT2	0x0C	0x0C
URL	0x10	0x10
UTL	0x14	0x14
UIS	0x18	0x18
GIS	0x1C	0x1C

Table 3: Accessing local configuration registers

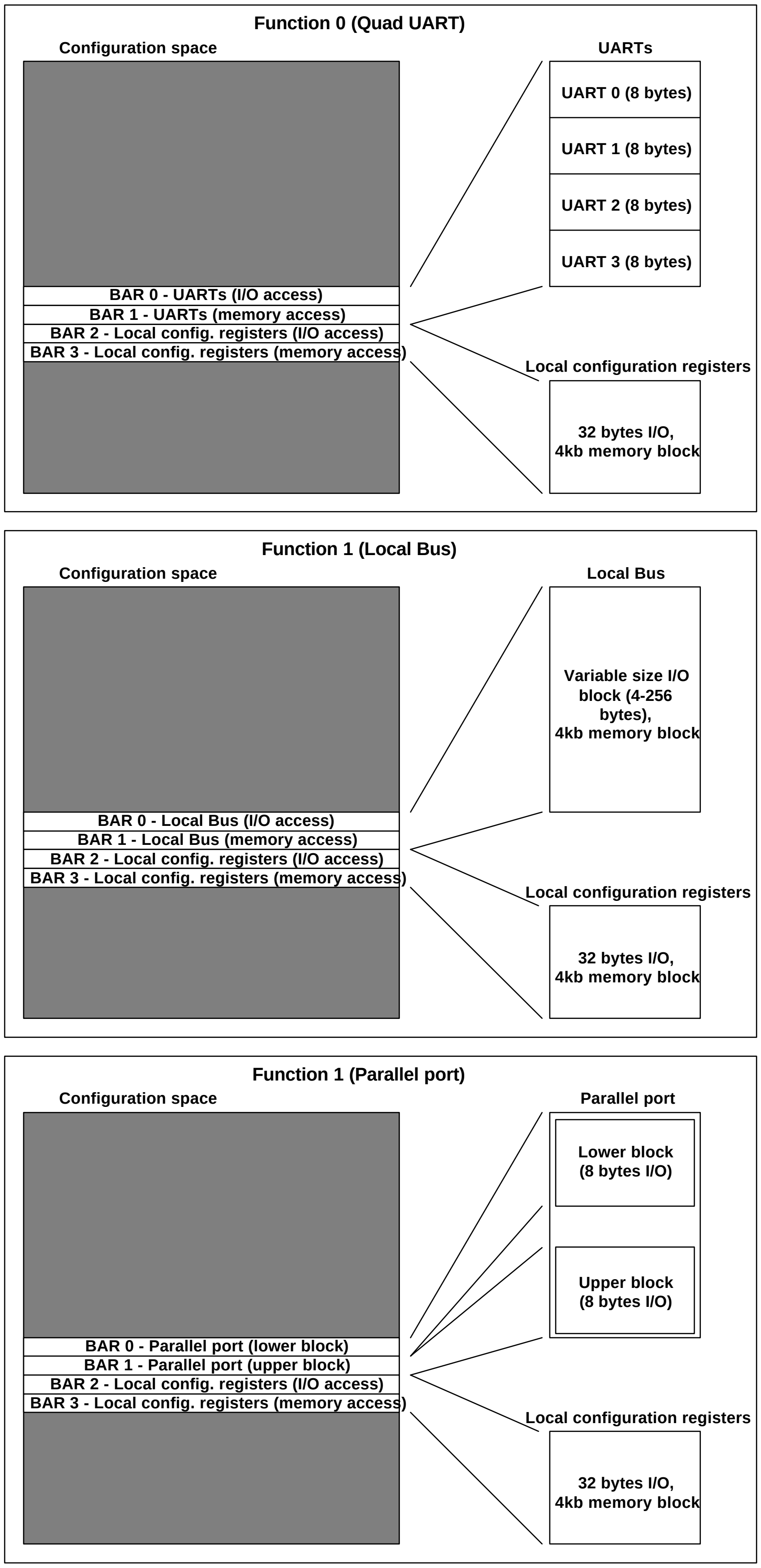

Function 0 (Quad UART)
Configuration space
UARTs
UART 0 (8 bytes)
UART 1 (8 bytes)
UART 2 (8 bytes)
UART 3 (8 bytes)
BAR 0 - UARTs (I/O access)
BAR 1 - UARTs (memory access)
BAR 2 - Local config. registers (I/O access)
BAR 3 - Local config. registers (memory access)
Local configuration registers
32 bytes I/O,
4kb memory block
Function 1 (Local Bus)
Configuration space
Local Bus
Variable size I/O block (4-256 bytes),
4kb memory block
BAR 0 - Local Bus (I/O access)
BAR 1 - Local Bus (memory access)
BAR 2 - Local config. registers (I/O access)
BAR 3 - Local config. registers (memory access)
Local configuration registers
32 bytes I/O,
4kb memory block
Function 1 (Parallel port)
Configuration space
Parallel port
Lower block (8 bytes I/O)
Upper block (8 bytes I/O)
BAR 0 - Parallel port (lower block)
BAR 1 - Parallel port (upper block)
BAR 2 - Local config. registers (I/O access)
BAR 3 - Local config. registers (memory access)
Local configuration registers
32 bytes I/O,
4kb memory block

Figure 1: Base address mapping

3.3 PCI interrupts

PCI interrupts are level-sensitive and can therefore be shared. One PCI add-in card is permitted to use all four interrupt pins, but if it uses fewer than this it must take INTA# first, then INTB#, INTC#, INTD# in order. A PCI platform with expansion slots usually has the form of interrupt layout shown in . Using this architecture, if all the add-in cards require only their INTA#, all card will have their own interrupt. Sharing will occur if a device uses more than one interrupt, or there are more than four devices on the bus.
To assert an interrupt, the device drives the signal low. To deassert, it allows it to float (open-drain connection). The pull-up is located on the system board.

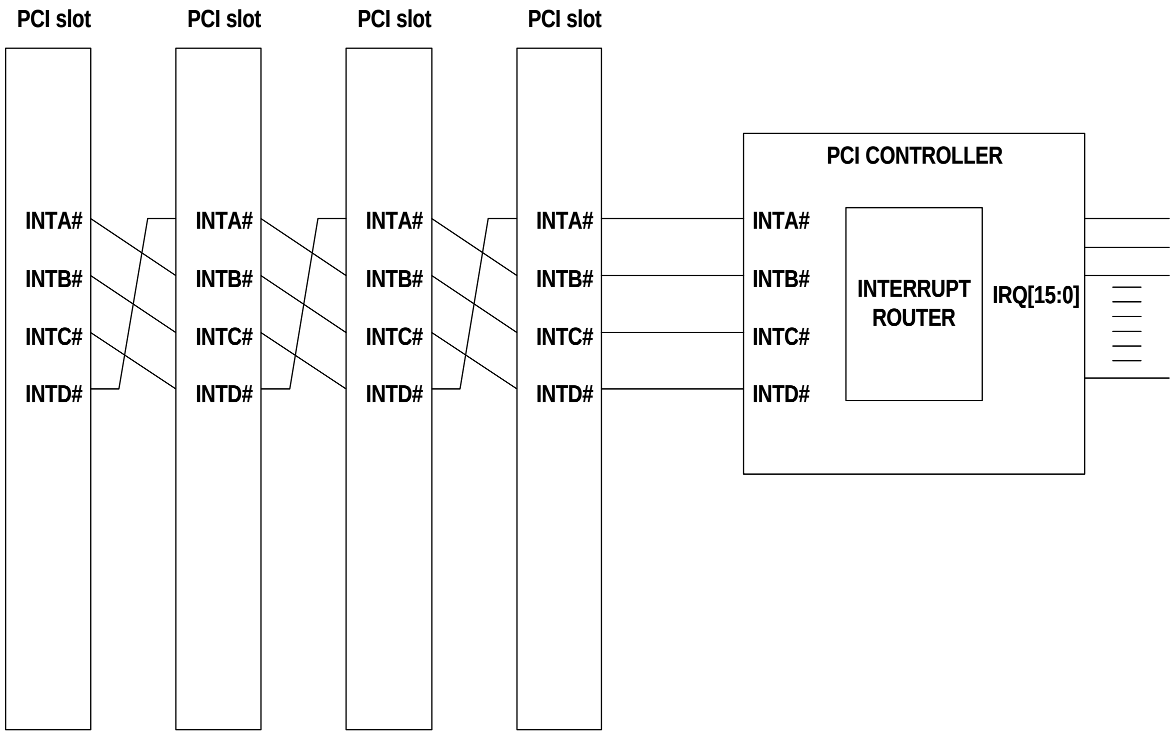


Figure 2: PCI interrupt routing in PC systems

The OX16PCI954 can make use of one or two interrupts (INTA# and INTB#). The default routing is for Function 0 to assert interrupts on INTA# and Function 1 to assert interrupts on INTB#. This can be changed using the serial EEPROM; the only other permissible configuration is for both functions to use INTA#.

4 Interfacing the OX16PCI954

4.1 Pin descriptions

This section gives a functional description of each of the OX16PCI954 device pins. Device pin listings are given in Sections 3 and 4 of the data sheet; the information given below serves as an extension to the data sheet descriptions.

4.1.1 Power supply pins

VDD and GND	Supply power to the device. When devices have multiple power and ground pins, all instances of VDD should be connected to +5V DC and all instances of GND should be connected to the 0V rail. Attention should also be given to local high-frequency decoupling due to the high-frequency switching that occurs in the device during normal operation. Note that there are two separate power and ground rails in the device, AC and DC. The DC rails supply power to all elements in steady state, while the AC rails supply current in switching state. This precaution reduces the effects of simultaneous sitching outputs and undesirable EMI interference.

4.1.2 PCI pins

AD[31:0]	32-bit multiplexed PCI address/data bus.
C/BE[3:0]#	4-bit bus command during PCI address phase, byte enable during data phase. The command encoding is as described in the PCI specification. Since addressing is always DWORD aligned, the Byte Enables ensure that only the correct bytes in an addressed DWORD are accessed.
CLK	PCI system clock – speeds up to 33MHz are supported by the OX16PCI954.
FRAME#	Signifies the start of a PCI transaction
DEVSEL#	Output by the target device to acknowledge that it is the recipient of the current transaction. The OX16PCI954 supports medium-speed decoding, ie it will assert DEVSEL# two clock cycles after FRAME# is activated if the address matches any of its BARs.
IRDY#	Output from the Bus Master (Initiator) to signify that it is ready to complete the data transfer.
TRDY#	Output from the target to signify that it is ready to complete the data transfer
STOP#	Target stop request. Output from the target to terminate the data transfer. The OX16PCI954 asserts this with TRDY# to signify disconnect-with-data on every transfer. Therefore burst accesses are not supported.
PAR	PCI transfer parity pin.
SERR#	System error – output by the OX16PCI954 to report an error that may jeopardize system or data integrity.
PERR#	Parity error – output by any PCI device to report a parity error
IDSEL	Used by the PCI controller to select a device for a configuratin access.
RST#	Active-low PCI device reset
INTA#, INTB#	PCI interrupts. These are active-low, level sensitive, shared interrupts. They should always be connected to the INTA# and INTB# pins respectively on the connector finger. The system board will then map these pins to any of the four PCI interrupts.
PME#	Power Management Event – used by the OX16PCI954 to report a power management event. Used to request that the device driver wake-up the device from a low-power state. If Power Management functionality is required, use of an isolator is recommended to prevent the OX16PCI954 activating PME# when powered down (see p25).

4.1.3 Serial port pins

FIFOSEL	FIFO Size Select input (common to all UART channels). This input determines the power up FIFO size. If it is tied low, the default FIFO size is 16 bytes. If it is tied high, the default is 128 bytes. The FIFO size is also settable in software, overriding the state of this pin. (See Section 7.2.2)
SOUT[3:0]	Serial data output pin (for the respective UART channel). This pin also provides modulated IrDA output when IrDA mode is enabled
SIN[3:0]	Serial data input pin (for the respective UART channel). This pin is also a modulated IrDA signal input when IrDA mode is enabled
CTS[3:0]#	Clear to send input (for the respective UART channel). This pin's state is reflected in bit 4 of the Modem Status Register. It is generally used for hardware flow control handshaking and is connected to the RTS# signal of the remote receiver. When automatic CTS flow control is enabled and this pin is de-asserted, the transmitter will be disabled after completion of the current character. This allows the remote receiver to moderate the flow of data from the transmitter and hence prevent FIFO overrun. A change of state of this pin will cause bit 0 of MSR to be set (delta CTS) and can be configured to generate an interrupt. This pin may also be used as a general-purpose input.
DSR#	Data Set Ready input (for the respective UART channel). This pin's state is reflected in bit 5 of the Modem Status Register. Similar to CTS# It is generally used for hardware flow control handshaking and is connected to the DTR# signal of the remote receiver. When automatic DSR flow control is enabled, de-asserting this pin has the same affect as de-asserting CTS described above. A change of state of this pin will cause bit 1 of MSR to be set (delta DSR) and can be configured to generate an interrupt. This pin may also be used as a general-purpose input.
RI[3:0]#	Ring Indicator input (for the respective UART channel). This pin's state is reflected in bit 6 of the Modem Status Register. This pin is generally connected to the RI output of a modem. This goes low when the modem is called from a remote device. A falling edge on this pin will cause bit 2 of MSR to be set (trailing edge RI) and can be configured to generate an interrupt. This pin may also be used as a general-purpose input. The RI pin may be configured as a receiver 1x clock source for isochronous operation. See Sections 4.3.5 and 7.3.8.
DCD[3:0]#	Data Carrier Detect input (for the respective UART channel). This pins state is reflected in bit 7 of the Modem Status Register. This pin is generally connected to the DCD output of a modem, which is asserted when a valid data carrier signal is present on the line. A change of state of this pin will cause bit 3 of MSR to be set (delta DCD) and can be configured to generate an interrupt. This pin may also be used as a general-purpose input.
RTS[3:0]#	Ready to send output (for the respective UART channel). This pin's state is set by bit 1 of the Modem Control Register. It is generally used for hardware flow control handshaking and is connected to the CTS# input of the remote transmitter. When automatic RTS flow control is enabled this pin is de-asserted and asserted in accordance with pre-defined flow control trigger levels in the receiver FIFO. This allows the receiver to disable and enable the remote transmitter according to how much data is held in the receiver FIFO (as long as it is also using CTS/RTS flow control) and hence prevents FIFO overrun. This pin may also be used as a general-purpose output.
DTR[3:0]#	Data Terminal Ready output (for the respective UART channel). This pin's state is set by bit 0 of the Modem Control Register. It is generally used for hardware flow control handshaking and is connected to the DSR# input of the remote transmitter. When automatic DTR flow control is enabled this pin is de-asserted and asserted in accordance with pre-defined flow control trigger levels in the receiver FIFO, as with RTS. This pin may also be used as a general-purpose output. DTR may also be configured as an RS-485 buffer enable pin (see Section 4.2.2.1) or a 1x clock source (Section 4.3.5).
XTLO	This pin provides an output driver for a crystal oscillator circuit (see Section 4.3). It is not used when an alternative TTL level clock input is applied to XTLI, and can be left unconnected.
XTLI	Main system clock input. This pin is used as a direct TTL level clock input or in conjunction with XTLO in a crystal oscillator circuit configuration (see Section 4.3).

4.1.4 Local Bus Pins

LBA[7:0]	Local Bus address signals. In 32-bit mode the UART's pins are redefined to provide LBA[11:0]
LBD[7:0]	Local Bus Data signals. In 32-bit mode the UART's pins are redefined to provide LBD[31:0]
LBCS[3:0]	Local Bus chip-select signals. The local bus address block can be subdivided into one, two or four chip-select regions. If it is defined as one whole block, LBCS0# will be activated on all accesses. If there are two regions, the lower half will be controlled by LBCS0# and the upper half by LBCS1#. If four regions, the block is split equally between the four chip-select signals. See Section 4.2.4 for information on defining chip-select regions. In motorola-type mode these pins are redefined as Data strobe (LBDS#) pins.
LBWR#	Local Bus write strobe. In Intel-type mode this initiates a write access to local bus peripherals. In Motorola-type mode this pin is redefined to Read-not-Write control (LBRDWR#).
LBRD#	Local Bus read strobe. In Intel-type mode this initiates a read access to local bus peripherals. In Motorola-type mode this pin is redefined to permanent high impedance.
LBRST	Local Bus active-high reset
LBRST#	Local Bus active-low reset.
LBDOUT	Local Bus data-out enable. This pin can be used by optional external transceivers; it is high when LBD[7:0] are in output mode and low when they are in input mode. This pin also applies to the parallel port.
UART_Clk_Out	Buffered UART oscillator output. This clock is a buffered version of the signal present on the XTLI pin. It can be used to drive external oscillators from one single oscillator circuit. See Section 8.4 for information on this pin.
LBCLK	Buffered PCI Clock. Local bus operation is synchronised to this clock. See Section 8.4 for information on this pin.

4.1.5 Parallel port pins

STANDARD PARALLEL PORT (SPP) MODE:

PD[7:0]	Bi-directional 8-bit parallel data, bits 7 (MSB) to 0 (LSB).
INIT#	Active-Low Initialise input/output. This pin is used to send an initialisation signal to a peripheral. The pins state is set using 'Peripheral Control Register' PCR[2]. PCR[2] = 0 sets the output low (active) PCR[2] = 1 sets it high.
ERR#	Peripheral Error Input. This pin is held low by a peripheral to assert an error condition. The state of this pin is reflected in bit 3 of the 'Peripheral Status Register' PSR. This is a user defined pin in EPP mode.
SLCT#	Peripheral Selected Input. This pin is held low by a peripheral when it is selected. The state of this pin is reflected in bit 4 of the 'Peripheral Status Register' PSR. This is a user defined pin in EPP mode.
PE	Paper Empty input. This pin is held high by a peripheral to assert an 'out of paper' condition. The state of this pin is reflected in bit 5 of the 'Peripheral Status Register' PSR. This is a user defined pin in EPP mode.
ACK#	Peripheral Acknowledge Input. PSR[6] PSR[2] (INT#) cleared on rising edge of this pin
AFD#	Auto Feed input/output. The state of this pin is set and read using bit 1 of the 'Peripheral Control Register' PCR bit 1 (0 = inactive, 1 = active).
BUSY	Peripheral Busy Input. This pin is set high by a peripheral when it is not ready to receive data. Its state is reflected in 'Peripheral Status Register' PSR bit 7.

SLIN#	Peripheral Select input/output. This output is asserted to attempt to select a peripheral. The state of this pin is set by 'Peripheral Control Register' PCR bit 3 (0 = inactive, 1 = active). The input state of the pin is also readable from this bit.
STB#	Data Strobe input/output. The peripheral uses this line to latch the data currently available on the PD[7:0] data lines. The state of this pin is set by 'Peripheral Control Register' PCR bit 0 (0 = inactive, 1 = active). The input state of the pin is also readable from this bit.

ENHANCED PARALLEL PORT (EPP) MODE:

For more information on EPP parallel port signals, refer to the IEEE 1284 EPP specification. Most pins remain the same as defined for SPP mode, however the following pins are redefined:

INTR# (ACK#)	Functionality is the same as SPP-mode ACK#.
WAIT# (BUSY#)	Active-low handshake output. This pin is driven by the internal EPP controller.
DATASTB# (AFD#)	Active-low Data Strobe output. This pin is driven by the internal EPP controller and provides data read/write data strobe signals.
ADDRSTB# (SLIN#)	Active-low address strobe output. This pin is driven by the internal EPP controller, and provides the address read / write strobe signals.
WRITE# (STB#)	Write / not read output. This pin is driven by the internal EPP controller, identifying write cycles when low and read cycles when high.

4.1.6 Subsystem ID and Subsystem Vendor ID pins

Sub_ID[15:0]	In Mode '10', these pins are provided to hard-wire the Subsystem ID of Function 0
Sub_V_ID[15:0]	In Mode '10', these pins are provided to hard-wire the Subsystem Vendor ID of Function 0

4.1.7 Multi-purpose Input and output pins

MIO[11:3]	These pins can drive high or low, or be used as inputs to generate a PCI interrupt.
MIO2	The function of this pin is dependent on the setting of LCC[7] in the local configuration registers. When LCC[7]=0, this pin has the same MIO function as MIO[11:3]. When LCC[7] is set, it is used as an input to generate a Power management event for function 1.
MIO1	The function of this pin is dependent on the setting of LCC[6:5]. When LCC[6:5]=00, it has the same MIO function as MIO[11:3]. When LCC[6:5]≠00, this pin is defined as permanent high-impedance.
MIO0	The function of this pin is dependent on the mode of operation of the OX16PCI954. When in Mode '01' this pin is defined as permanent high-impedance. In other modes it has the same MIO function as MIO[11:3].

4.1.8 Serial EEPROM pins

EE_CK	EEPROM clock. If the optional serial EEPROM is used, this pin should be connected to CK. This pin drives high or low depending on the setting of LCC[24] in the Local Configuration Registers
EE_CS	EEPROM active-high chip-select. If the optional serial EEPROM is used, this pin should be connected to CS. This pin drives high or low depending on the setting of LCC[25] in the Local Configuration Registers.
EE_DI	EEPROM data in. If the optional serial EEPROM is used, this pin should be connected to DO, and pulled high with an external resistor (value 1k-10k).

EE_DO EEPROM data out. If the optional serial EEPROM is used, this pin should be connected to DI. This pin drives high or low depending on the setting of LCC[26] in the Local Configuration Registers.

Note: after a reset, these pins are controlled by the EEPROM interface controller, and will download any valid memory contents to the OX16PCI954.

4.1.9 Miscellaneous pins

TEST This pin must be connected to GND.

MODE[1:0] These pins select the mode of operation of the device. See Section 3.1.

4.2 Standard Connectivity

This section describes briefly how to interface the OX16PCI954 modules to other components of a serial / parallel port application. Table 4 provides a quick reference, and each module is described seperately in more detail below.

Pin	Mode	Description	Action when used	Action when not used
VDD	All	Power supply	Connect directly to +5V DC	N/A
GND	All	Power supply	Connect directly to 0V DC	N/A
TEST	All	Manufacturing test	Connect directly to 0V DC	N/A
Mode[1:0]	All	Mode selector	Select mode as per Table 1 (p6).	N/A
AD[31:0]	All	PCI Multiplexed Address / Data pins	Connect to AD[31:0] on the PCI bus	N/A
C/BE[3:0]#	All	PCI Command / byte enable	Connect to C/BE[3:0] on the PCI bus	N/A
CLK, FRAME#, DEVSEL#, IRDY#, TRDY#, STOP#, PAR, SERR#, PERR#, IDSEL, RST# INTA#, INTB#, PME#	All	PCI Control signals	Connect to respective signals on the PCI bus	N/A
FIFOSEL	00,01,10	FIFO select	Connect to GND for 16-deep FIFOs, to VDD for 128-deep FIFOs	Tie high or low
SOUT	00,01,10	Serial data output	Connect to a suitable line driver (See Section 4.2.2)	Leave unconnected
SIN	00,01,10	Serial data input	Connect to a suitable line receiver (See Section 4.2.2)	Tie high
RTS#	00,01,10	Request-To-Send Modem signal output	Connect to a suitable line driver (See Section 4.2.2)	Leave unconnected
CTS#	00,01,10	Clear-To-Send Modem signal input	Connect to a suitable line receiver (See Section 4.2.2)	Tie high
DTR#	00,01,10	Data-Terminal-Ready Modem signal output	Connect to a suitable line driver (See Section 4.2.2)	Leave unconnected
DSR#	00,01,10	Data-Set-Ready Modem signal input	Connect to a suitable line receiver (See Section 4.2.2)	Tie high
DCD#	00,01,10	Data-Carrier-Detect Modem signal input	Connect to a suitable line receiver (See Section 4.2.2)	Tie high
RI#	00,01,10	Ring-Indicator Modem signal input	Connect to a suitable line receiver (See Section 4.2.2)	Tie high

Pin	Mode	Description	Action when used	Action when not used
UART_Clk_Out	All	Buffered UART clock	Connect to XTLI on external UARTs	Leave unconnected
LBRST	00,11	Local Bus reset	Connect to active-high reset on external UARTs	Leave unconnected
LBRST#	00,11	Local Bus reset	Connect to active-low reset on external UARTs	Leave unconnected
LBDOUT	00,01,11	Local Bus Data out enable	Connect to direction pin of external transceiver	Leave unconnected
LBCLK	00,11	Buffered PCI clock	Connect to clock reference on external devices	Leave unconnected
LBCS# LBDS#	00,11 (I) 00,11 (M)	Local Bus chip-select Local Bus Data strobe	Connect to chip-select on external Intel-mode devices Connect to data-strobe on external Motorola-mode devices	Leave unconnected
LBWR# LBRDWR#	00,11 (I) 00,11 (M)	Local Bus write strobe Local Bus Read-not-write control	Connect to WR# on external Intel-mode devices Connect to RD/WR# on external Motorola-mode devices	Leave unconnected
LBRD#	00,11 (I)	Local Bus read strobe	Connect to RD# on external Intel-mode devices	Leave unconnected (always N/C when local bus is Motorola-type)
LBA[7:0] LBA[11:0]	00 11	Local Bus address	Connect to address pins on external devices or to decode logic	Leave unconnected
LBD[7:0] LBD[31:0]	00 11	Local Bus data	Connect to data pins on external devices	Leave unconnected
ACK#, PE, BUSY, SLCT, ERR#, SLIN#, INIT#, AFD#, STB#	01	Parallel port control signals	Connect to parallel port connector and pull up with external resistor (1k-10k)	Leave unconnected
PD[7:0]	01	Parallel port data signals	Connect to data transceiver or parallel port connector.	Leave unconnected
Sub_ID[15:0]	10	Subsystem ID	Tie pins to VDD or GND as per required subsystem ID	N/A
Sub_V_ID[15:0]	10	Subsystem Vendor ID	Tie pins to VDD or GND as per required subsystem Vendor ID	N/A
MIO[11:0]	All	Multi-purpose Input/Output	Configure function in Local configuration registers, connect as required	Tie low (or high as required) with 10k resistor
EE_CS	All	EEPROM chip select	Connect to EEPROM CS pin	Leave unconnected
EE_CK	All	EEPROM clock	Connect to EEPROM CK pin	Leave unconnected
EE_DI	All	EEPROM Data in	Connect to EEPROM DO pin and pull up with external resistor (1k-10k)	Leave unconnected
EE_DO	All	EEPROM Data out	Connect to EEPROM DI pin	Leave unconnected

Table 4: Device Pins

4.2.1 Common connectivity example

Figure 3 shows how to interface the common elements of a solution using the OX16PCI954.

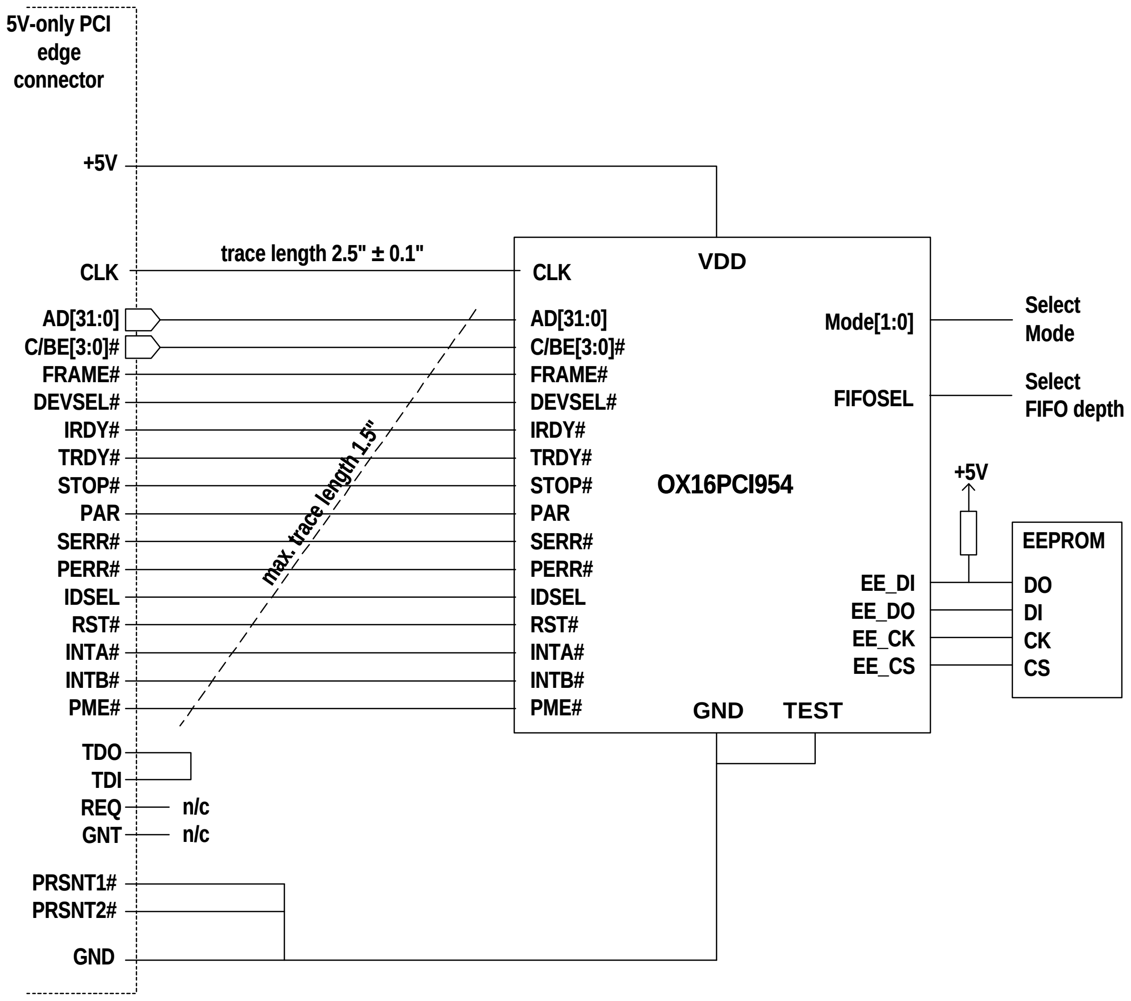


Figure 3: Interfacing the OX16PCI954

Note: All PCI pins should be connected to their respective signals on the PCI bus / edge connector, in strict adherence to the layout guidelines provided in the PCI specification.

4.2.2 Serial port interfacing

4.2.2.1 Connectivity examples

The examples below show how to interface any of the serial ports to RS232, RS422 and RS485 half-duplex line drivers/receivers. Either clock option is valid; use a crystal with suitable passive components, or connect directly to a clock source.

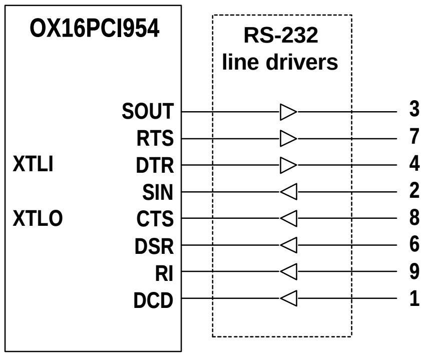


Figure 4: Interfacing an RS-232 port

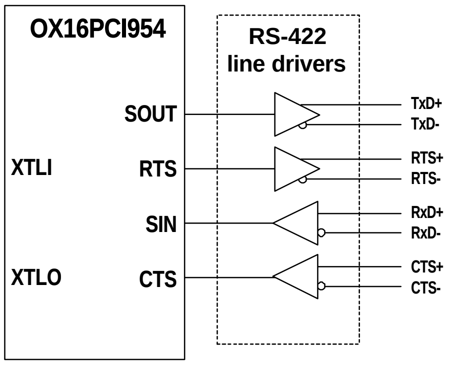


Figure 5: Interfacing an RS-422 port

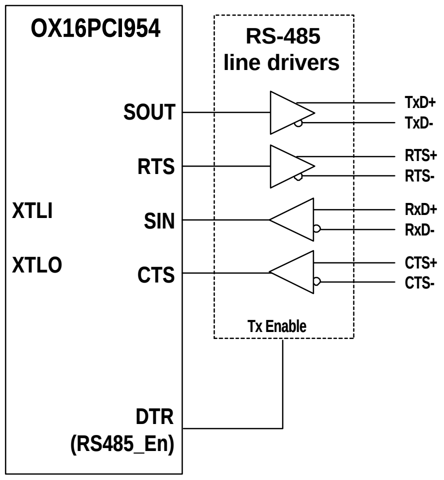


Figure 6: Interfacing an RS-485 port for half-duplex operation

4.2.2.2 RS-232 Line Drivers

RS-232 is the industry standard for PC serial communications. It employs single ended line driving at +/- 12V logic levels and can cover distances of up to 30 metres. Most standard RS-232 line drivers are capable of speeds up to 115.2 Kbps and can be used in conjunction with cost effective bundled multi-wire cables. Some high-speed line drivers are also available, capable of speeds up to 1Mbps.

4.2.2.3 RS-422 Line Drivers

RS-422 allows a single transmitter to communicate with up to 10 separate receivers. Differential signals are used requiring the use of a twisted pair cable over long distances. Each signal requires a pair of cables instead of the single cable used for RS-232.

Data rates of up to 10 Mbps per second are achievable over short distances (around 10 metres), with lower rates, up to 100 Kbps, being possible over distances as high as 1.2 Km.

4.2.2.4 RS-485 Line Drivers

RS-485 is similar to RS-422. It allows up to 32 transmitters to communicate with up to 32 receivers on a common data bus. One device will transmit data at any given time, while the remaining devices are able to simultaneously receive it. RS-485 line drivers are equipped with drive enable inputs that allow them to either drive the bus, or to switch into a high impedance tri-state mode.

This standard operates under the same distance / speed restrictions as RS-422 and has the same cable requirements.

4.2.2.5 Suggested Line Drivers

The following table gives a suggested line transceiver for each of the above standards. Suitable parts mentioned are available from *MAXIM*™ (see Table 5), although there are a wide variety of line drivers and receivers available from a range of other manufacturers, such as National Semiconductor™, Analog Devices™, Linear Technology™ and many others.

Protocol	Maximum Speed	Line Driver Description	MAXIM ™ Part No.
RS-232	115.2 Kbps	4 Drivers / 5 Receivers, 28-pin surface mount	MAX241
RS-232	1 Mbps	2 Drivers / 2 Receivers, 20-pin surface mount	MAX3225
RS-422 / 485	10 Mbps	Single Driver / Receiver, 10-pin surface mount	MAX1484

Table 5: Suggested Line Transceiver IC's

4.2.3 Local bus examples

Figure 7 shows an example of a Quad UART device connected to the local bus in Intel mode (to make an 8-port serial card). Figure 8 shows 3 devices connected in Motorola mode; this would comprise a 16-port serial card. Note, for glueless implementation of serial cards with more than 8 ports Motorola mode is necessary as there are only four chip selects available.

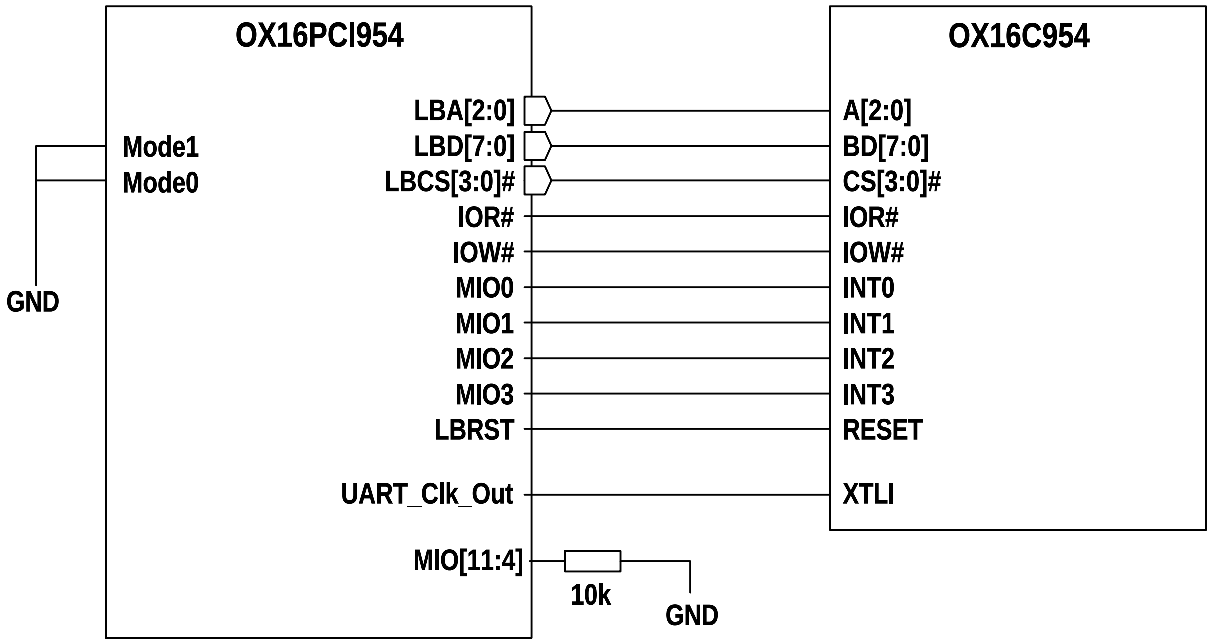


Figure 7: Interfacing the local bus in Intel Mode

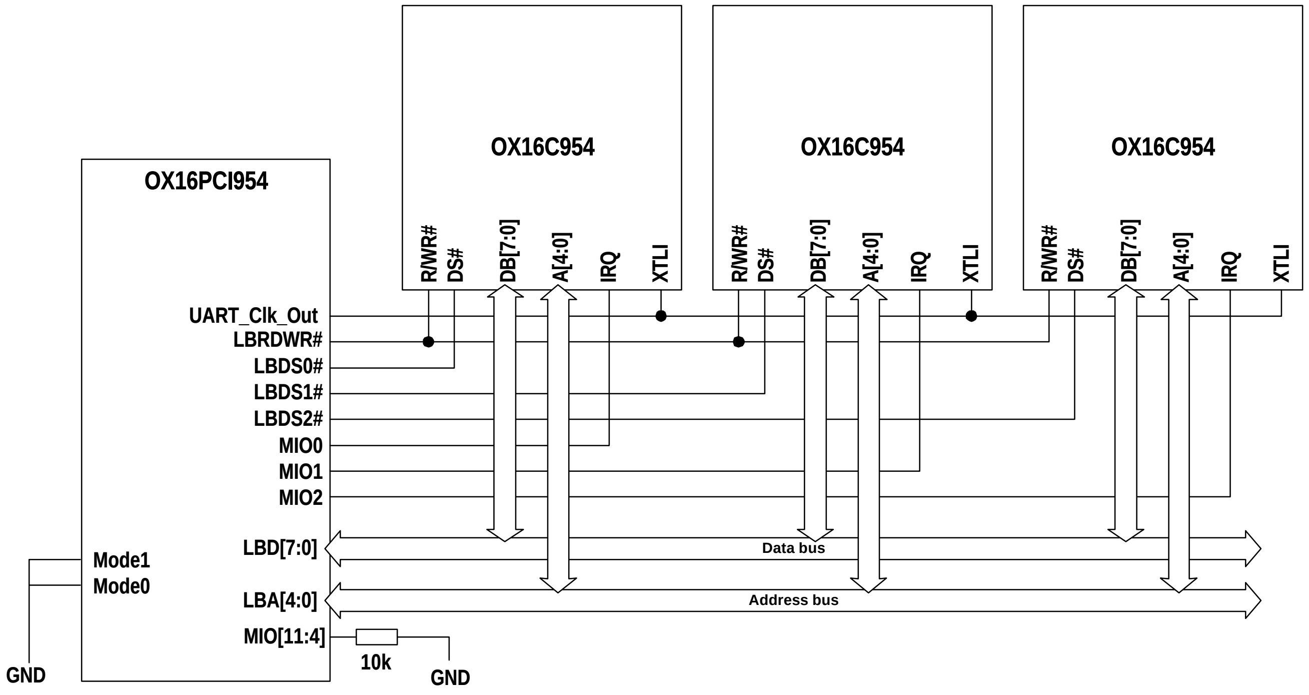


Figure 8: Interfacing the local bus in Motorola Mode

4.2.4 Local bus addressing

The local bus address space can be set in the region of 4 bytes – 256 bytes of I/O space, and 4K bytes of memory space (can be increased to 16K bytes in 32-bit bridge mode). In memory space, the addressable block is always divided into four chip-select regions, however in I/O space it can be divided into one, two or four regions according to the requirements of the peripheral devices connected.

The block size is defined by LT2[22:20] in the local configuration registers, as described in the data sheet. For block sizes less than 256 bytes, the upper address lines which are not needed will always be zero; for example if the default size of 32 bytes is used, this requires Local Bus address pins LBA[4:0], therefore LBA[7:5] will be zero.

The chip-select regions are defined using LT2[26:23]. This bits are named "Lower-Address-CS-Decode" and define which of the LBA[7:0] pins represents a chip-select region boundary. Shows the value which must be programmed into LT2[26:23] to select the required number of chip-select regions.

Number of chip-select regions	I/O space block size (bytes)						
	4	8	16	32	64	128	256
One	A2	A3	A4	A5	A6	A7	A8
Two		A2	A3	A4	A5	A6	A7
Four			A2	A3	A4	A5	A6

Table 6: Lower-address-Chip-select decode values

4.3 UART clock options

If the UARTs are required in the application, a clock signal must be applied to the XTLI pin. The speed of this clock determines the maximum baud rate at which the device can receive and transmit serial data. This maximum baud rate is equal to one sixteenth of the frequency of the system clock (Increasing to one quarter of this value if TCR=4 is used, see Section 7.3.1).

The industry standard system clock for PC COM ports is 1.8432 MHz, limiting the maximum baud rate to 115.2 Kbps. The OX16PCI954 supports system clocks up to 60MHz, and its flexible baud rate generation hardware means that almost any frequency can be optionally scaled down for compatibility with standard devices.

Designers have the option of using either TTL clock modules or crystal oscillator circuits for system clock input, with minimal additional components. The following two sections describe how each can be connected.

NOTE: For very low power applications, use of a low power TTL clock module is recommended

NOTE: Please see Section 4.3.4 on high speed operation when considering applications requiring baud rates in excess of 1Mbps.

4.3.1 TTL Clock Module

Using a TTL module for the system clock simply requires the module to be supplied with +5v power and GND connections. The clock output can then be connected directly to XTLI. XTLO should be left unconnected.

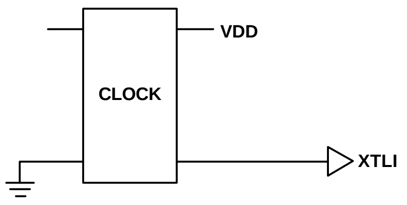


Figure 9: TTL Clock Module Connectivity

4.3.2 Crystal Oscillator Circuit

The OX16PCI954 provides the XTLO output pin to drive a crystal oscillator circuit. The circuit is shown below with suggested component values. Owing to the nature of such circuits, some variation in these values may be required to ensure stable oscillation at different frequencies. The total load capacitance (C1 and C2 in series) however, should be approximately that stated by the crystal manufacturer (nominally 16pF).

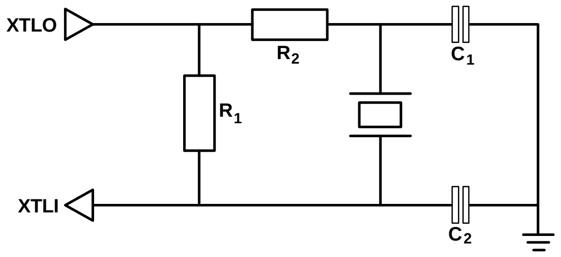


Figure 10: Crystal Oscillator Circuit

Frequency Range (MHz)	C1 (pF)	C2 (pF)	R1 (Ω)	R2 (Ω)
1-8	68	22	220K	470R
8-60	33 – 68	33 - 68	220K – 2M2	470R

4.3.3 Suggested Clock Frequencies

Table 7 below shows a range of standard serial communication clock frequencies, starting at 1.8432 MHz, the standard used in almost all PC COM ports. With each frequency is given a pair of maximum baud rates. One using the standard 16x over-sampling clock (i.e. where the maximum baud rate is restricted to 1/16 of the input clock frequency), and one using a 4x over-sampling clock (configured by writing to the TCR register). All baud rates assume that the prescaler is bypassed and hence no pre-division of the input clock is used. See Section 7.3.1 for more information on TCR and the prescaler.

Input Clock Frequency	Maximum standard baud rate	Maximum baud rate with 4x over-sampling clock
1.8432 MHz	115.2 Kbps	460.8 Kbps
3.6864 MHz	230.4 Kbps	921.6 Kbps
4.9152 MHz	307.2 Kbps	1.23 Mbps
7.3728 MHz	460.8 Kbps	1.843 Mbps
8.192 MHz	512 Kbps	2.048 Mbps
14.7456 MHz	921.6 Kbps	3.686 Mbps
18.432 MHz	1.152 Mbps	4.608 Mbps
32.768 MHz	2.048 Mbps	8.192 Mbps

Table 7: Example maximum baud rates for various input clock frequencies

The standard baud rate divisor word registers allow any divisor of these maximum values in the range 1 to 65535 (0x0001 to 0xFFFF).

4.3.4 High speed operation

Designers using these devices in high speed applications (UART System clock > 10MHz) are advised to follow the guidelines for high speed digital design, paying particular attention to the following:

- Keeping PCB tracks carrying high speed signals and return currents as short and direct as possible
- Correct termination of high-speed traces
- Use of mutli-layered PCB with separate power and ground planes
- Adequate decoupling as near as possible to all high-frequency components

4.3.5 Isochronous Clock Mode

Figure 11 below illustrates the clock configuration required to operate OX16C95x devices in Isochronous mode. The 950 can be configured to output its own 1x clock on the DTR pin and accept a receiver clock on the DSR pin. In the configuration shown, it is therefore possible to receive and transmit at different baud rates if different system clocks are employed at either end. Note that this figure does not apply to OX16C952 and OX16C954 devices – see relevant application notes for information.

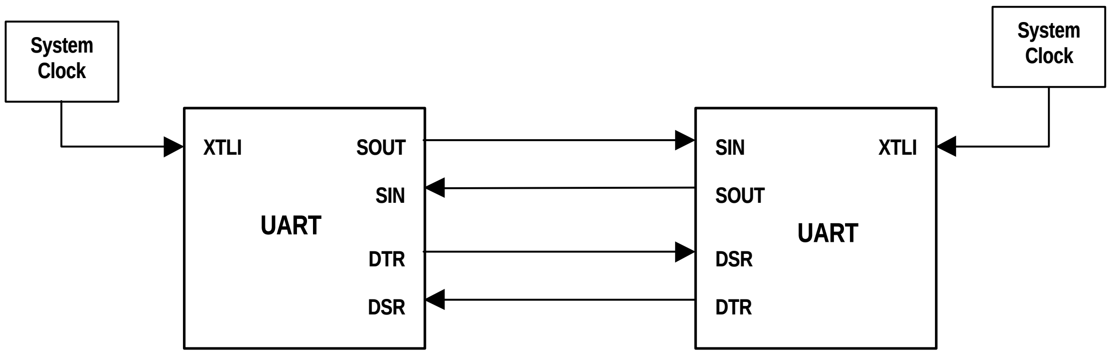


Figure 11: 950 Isochronous Configuration

For information on configuring Isochronous mode operation in the UART, see Section 7.3.8.

4.4 Multi-purpose input / output pins (MIO)

Twelve MIO pins are provided (MIO[11:0]). These can be used to provide drive high or low signals, or can be used to pass external sources of interrupts to the PCI INTA#/INTB# pins. The function of the pins is controlled using the MIC register. Each pin can be reconfigured seperately. When configured as an input, an active signal present on the MIO pin will be passed through to the PCI interrupt pin if the interrupt mask in the GIS register is set.

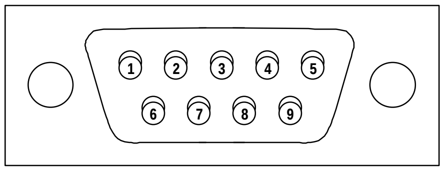
If there are unused MIO pins in the application, care should be taken that these do not cause spurious interrupts. Therefore they should either be masked off in the GIS register, or tied inactive using pull-down (or pull-up resistors). A direct connection to VDD or GND is sufficient; however there is risk of the user inadvertently configuring the pin to drive high or low.

4.5 External connections

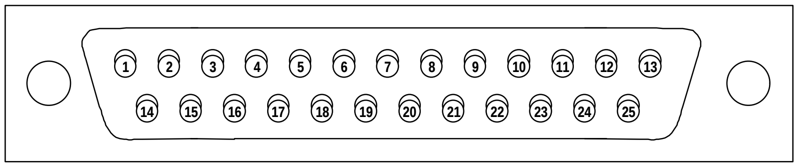
4.5.1 Serial ports

Standard PC COM ports operate using the RS232 standard and use either a 9-Pin or a 25-Pin male D-Type connector. The pin numbering for these connectors is given in the following diagram. Note that normally, only 9 pins on the 25 pin connector are actually used. It is used for compatibility with other devices such as modems, which also have 25 pin connectors.

Table 8 details the signal pin out for both of these connectors



9-Pin D Type Connector



25-Pin D Type Connector

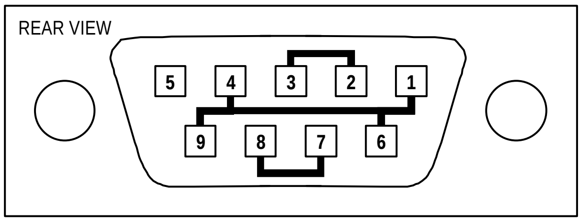
Figure 12: D Type connector pin numbering (Front view)

Signal		9-PIN CONNECTOR	25-Pin Connector
Data Carrier Detect	DCD	1	8
Received Data	RxD	2	3
Transmitted Data	TxD	3	2
Data Terminal Ready	DTR	4	22
Ground	GND	5	7
Data Set Ready	DSR	6	6
Request To Send	RTS	7	4
Clear To Send	CTS	8	5
Ring Indicator	RI	9	20

Table 8: Serial port D Type connector pin-outs

4.5.2 RS-232 Loopback Connector

Serial loopback connectors are often used for testing serial ports. Such connectors simply feed the output signals and data back in to their corresponding inputs on the same port. The diagram below illustrates the connectivity of an RS-232 loopback plug for a standard 9-Pin PC COM port. This is the rear view (solder side) of a female 9-pin D connector. It connects TxD to RxD, RTS to CTS and DTR to DSR, RI & DCD. The ground pin has no connection.



9-Pin RS-232 Serial Loopback Connector

4.5.3 PC Parallel Port Connections

The diagram below illustrates the pin numbering of the standard PC parallel port. This is a standard 25 pin D-Type connector and is shown here as if from the rear of the PC. The following table lists the standard simple parallel port pin-out for this connector.

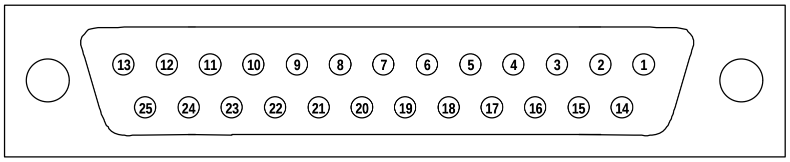


Figure 13: Standard SPP PC Parallel port connector

Pin #	Signal
1	STB#
2-9	PD0 – 7
10	ACK#
11	BUSY
12	PE
13	SLCT
14	AFD#
15	ERR#
16	INIT
17	SLIN
18-25	GND

Table 9: Parallel port connector pinout

4.6 Power Management

The OX16PCI954 support PCI Power Management states D0, D2 and D3. However, when in power state $D3_{cold}$, the device is powered down, and in this condition the PME# line presents a low-impedance path to GND through clamping diodes. In some cases this can cause the PC to wake up from Power State $D3_{cold}$. Therefore, care should be taken when designing an add-in card to be used in fully-ACPI compliant systems.

If Power Management functionality is not required, the PME# should be treated as no-connect

If Power management will be used, use of an isolator FET is recommended so that the PME# pin remains Hi-Z even when the OX16PCI954 is powered down. A suitable circuit is shown in Figure 14.

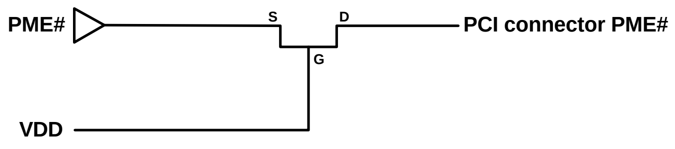


Figure 14: PME# isolator circuit

5 Typical applications

This section provides examples of typical serial / parallel applications using the OX16PCI954 device. Several typical application examples are given; for each one a description is given of the components required, connectivity needed and register/logical function mapping across the PCI interface. Detail is only given for connections particular to the application; it is assumed that the PCI interface, TEST etc. are connected as described in Section 4.

5.1 Low cost four-port serial card.

This application uses the device in Mode '10'; because neither the local bus or parallel port are required the pins can be used to assign the Subsystem Vendor ID and Subsystem ID of function 0.

5.1.1 Interfacing

The only components needed are the OX16PCI954 and RS232/422 line drivers. A block diagram is given in Figure 15.

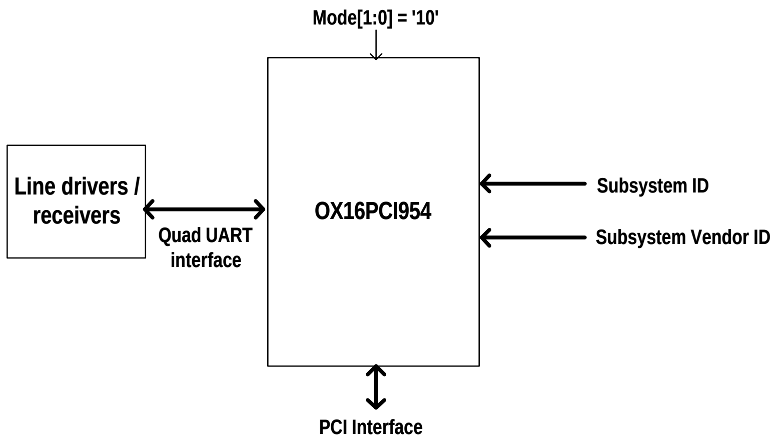


Figure 15: Low-cost four-port serial card

OX16PCI954 pins	Connection
Mode[1:0]	10
PCI control / data pins	PCI interface connections
UART signals	RS232/422/485 Line drivers
Sub_V_ID[15:0]	16-bit Subsystem Vendor ID
Sub_ID[15:0]	16-bit Subsystem ID

Table 10: Connections required

5.1.2 Operation

The UARTs are accessed through BAR0 and BAR1 of function 0, as shown in Table 11. Note that since the memory space occupies a 4kb block, there are a number of aliases of the UARTs in the allocated region. The local configuration registers can be accessed as described in Section 3.2.

UART address (bin)	Offset from Base Address 0 in I/O space (hex)				Offset from Base Address 1 in memory space (hex)			
	UART0	UART1	UART2	UART3	UART0	UART1	UART2	UART3
000	00	08	10	18	00	20	40	60
001	01	09	11	19	04	24	44	64
010	02	0A	12	1A	08	28	48	68
011	03	0B	13	1B	0C	2C	4C	6C
100	04	0C	14	1C	10	30	50	70
101	05	0D	15	1D	14	34	54	74
110	06	0E	16	1E	18	38	58	78
111	07	0F	17	1F	1C	3C	5C	7C

Table 11: Access to UARTs in I/O and memory space

5.2 Combo four-port serial, one-port parallel card.

This application uses the device in Mode '01'. Note that the serial EEPROM is not necessary for operation; however it is necessary to set Subsystem IDs to non-zero values for full PCI 2.2 compliance.

5.2.1 Interfacing

The only components needed are the OX16PCI954 and RS232/422 line drivers. A block diagram is given in Figure 16.

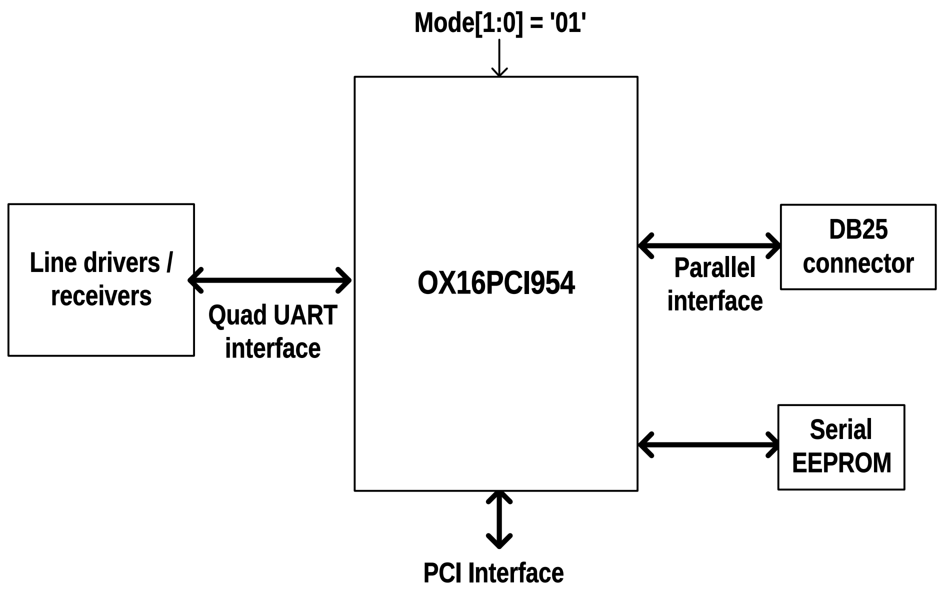


Figure 16: Four-port serial plus parallel port card

OX16PCI954 pins	Connection
Mode[1:0]	01
PCI control / data pins	PCI interface connection
UART signals	RS232/422/485 Line drivers
EEPROM signals	Serial EEPROM
MIO[11:0]	GND (via 10k resistor)
Parallel port control / data pins	Parallel port connector

Table 12: Connections required

5.2.2 Operation

The UARTs are accessed through BAR0 and BAR1 of function 0, in the same fashion as for the previous example (see Table 11). The parallel port register map is divided into two block, lower and upper. In legacy parallel ports the upper block is always placed at an address 400h above the lower block. If this relationship is set in the BARs for the OX16PCI954, generic drivers can be used for SPP and EPP mode operation. A register map for the parallel port is shown in Table 13, with the assumption that the 400h relationship is present.

Parallel port address (hex)	Offset from Base Address 0 (hex)	Offset from Base Address 1 (hex)
000	00	N/A
001	01	N/A
002	02	N/A
003	03	N/A
004	04	N/A
005	05	N/A
006	06	N/A
007	07	N/A
400	N/A	00
401	N/A	01
402	N/A	02
403	N/A	03

Table 13: Access to Parallel Port in I/O space

5.3 8-port serial card

In this configuration, the card designer requires a combination of OX16PCI954, an external Quad UART (recommended OX16C954) and line drivers. Again, the optional EEPROM is recommended to provide maximum flexibility. Figure 17 shows a block diagram.

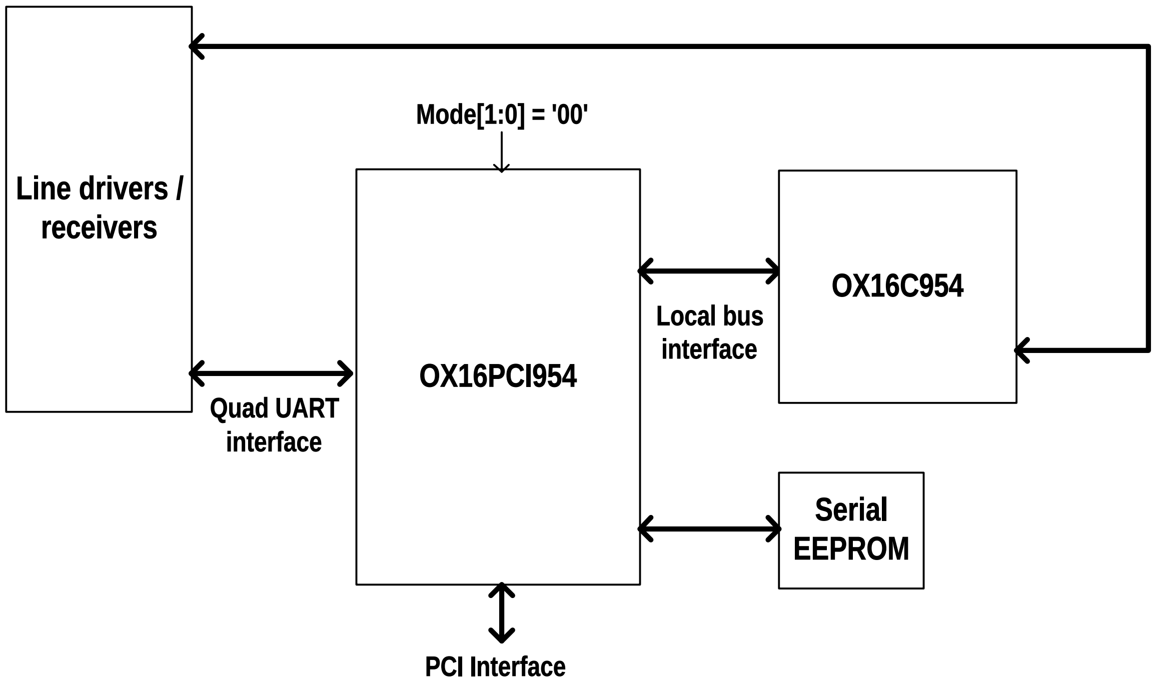


Figure 17: 8-port serial card.

OX16PCI954 pins	Connection
Mode[1:0]	00
PCI control / data pins	PCI interface connection
UART signals	RS232/422/485 Line drivers
EEPROM signals	Serial EEPROM
LBD[7:0]	OX16C954 DB[7:0]
LBA[2:0]	OX16C954 A[2:0]
LBRD#	OX16C954 RD#
LBWR#	OX16C954 WR#
LBCS[3:0]#	OX16C954 CS[3:0]#
MIO[3:0]	OX16C954 INT[3:0] + pull-down resistor
MIO[11:4]	GND (via 10k resistor)
Parallel port control / data pins	Parallel port connector

Table 14: Connections required

5.3.1 Operation

The internal UARTs are accessed through BAR0 and BAR1 of function 0, in the same fashion as for the example in Section 5.1.2 (see Table 11). Similarly, the local bus UARTs are accessed through BAR0 and BAR1 of function 1. When the 8-bit local bus is accessed in memory space, AD[9:2] is asserted on LBA[7:0], and the 4kb block is always divided into four chip-select regions. Table 15 shows a register map for the local bus UARTs in I/O and memory space. Note that since the memory space occupies a 4kb block, there are a number of aliases of the UARTs in the allocated region.

UART address (bin)	Offset from Base Address 0 in I/O space (hex)				Offset from Base Address 1 in memory space (hex)			
	UART0	UART1	UART2	UART3	UART0	UART1	UART2	UART3
000	00	08	10	18	00	400	800	C00
001	01	09	11	19	04	404	804	C04
010	02	0A	12	1A	08	408	808	C08
011	03	0B	13	1B	0C	40C	80C	C0C
100	04	0C	14	1C	10	410	810	C10
101	05	0D	15	1D	14	414	814	C14
110	06	0E	16	1E	18	418	818	C18
111	07	0F	17	1F	1C	41C	81C	C1C

Table 15: Access to Local Bus UARTs in I/O and memory space

5.4 12,16,20-port cards

Up to four external Quad UARTs can be addressed from the 8-bit local bus, however for glueless implementation of serial cards with port counts higher than 8, the local bus should be configured to operate in Motorola mode; there are only four chip-selects available hence only one can be used for each peripheral device. The I/O space block size will also have to be changed; therefore for these reasons the serial EEPROM is mandatory for these applications. Figure 18 shows a block diagram for a 20-port card, although the example can easily be altered to lower port-count applications by removing the extra OX16C954 devices and changing the respective addressing parameters.

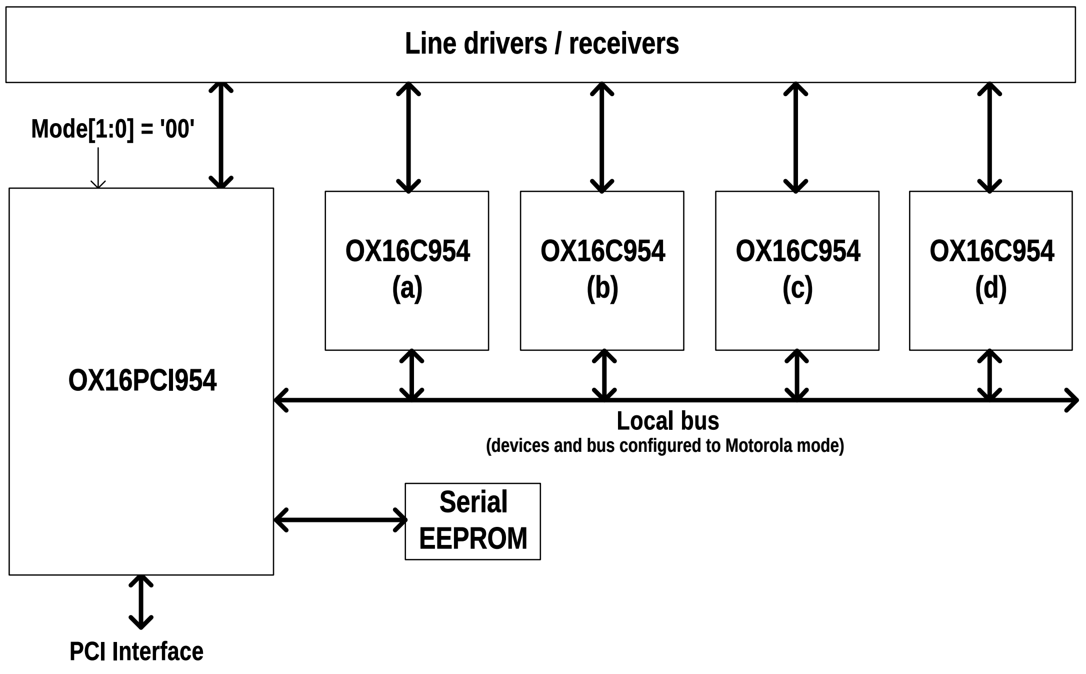


Figure 18: 20-port serial card

OX16PCI954 pins	Connection
Mode[1:0]	00
PCI control / data pins	PCI interface connection
UART signals	RS232/422/485 Line drivers
EEPROM signals	Serial EEPROM
UART_Clk_Out	OX16C954 XTLI
LBD[7:0]	OX16C954 DB[7:0]
LBA[4:0]	OX16C954 A[4:0]
LBRDWR#	OX16C954 R/W#
LBDS0#	OX16C954(a) DS#
LBDS1#	OX16C954(b) DS#
LBDS2#	OX16C954(c) DS#
LBDS3#	OX16C954(d) DS#
MIO0	OX16C954(a) IRQ# + pull-up resistor
MIO1	OX16C954(b) IRQ# + pull-up resistor
MIO2	OX16C954(c) IRQ# + pull-up resistor
MIO3	OX16C954(d) IRQ# + pull-up resistor
MIO[11:4]	GND (via 10k resistor)
Parallel port control / data pins	Parallel port connector

Table 16: Connections required

Table 17 shows the necessary fields that will need to be reprogrammed by the EEPROM. Other values, such as identification registers, class codes etc. may also be desired.

OX16PCI954 register	Value	Description
LCC[2]	1	Enable UART_Clk_Out
MIC[7:0]	55h	MIO[3:0] = inverting inputs
LT1[15:12]	4h	Write-cycle Read-not-Write deassertion = 4 cycles
LT1[19:16]	1h	Read-cycle Data-strobe assertion = 1 cycle
LT1[23:20]	3h	Read-cycle Data-strobe deassertion = 3 cycles
LT1[27:24]	1h	Write-cycle Data-strobe assertion = 1 cycle
LT1[31:28]	3h	Write-cycle Data-strobe deassertion = 3 cycles
LT2[22:20]	110	Function 1 I/O space block size= 128 bytes
LT2[26:23]	0011	Divide into four chip-select regions (Lower-address-CS-decode = A5)
LT2[31]	1	Set to Motorola mode

Table 17: EEPROM configuration

5.4.1 Operation

The internal UARTs are accessed through BAR0 and BAR1 of function 0, in the same fashion as for the example in Section 5.1.2 (see Table 11). Similarly, the local bus UARTs are accessed through BAR0 and BAR1 of function 1. In this example, the I/O space block size is 128 bytes; for similar applications with fewer ports it simply needs to be large enough to address the requisite number of UARTs. The CS-decode parameter will also need to be adusted to divide up the I/O space block into the correct number of Chip-select regions (one for each external UART chip). When the 8-bit local bus is accessed in memory space, AD[9:2] is asserted on LBA[7:0], and the 4kb block is always divided into four chip-select regions. Table 15 shows a register map for the serial ports in I/O and memory space. Note that since the memory space occupies a 4kb block, there are a number of aliases of the UARTs in the allocated region.

Port number	Physical device	PCI function	I/O space addressing (hex offset from BAR0)	Memory space addressing (hex offset from BAR1, DWORD aligned)
0	Internal UART0	0	00-07	00-1C
1	Internal UART1	0	08-0F	20-3C
2	Internal UART2	0	10-17	40-5C
3	Internal UART3	0	18-1F	60-7C
4	LB Chip1, UART0	1	00-07	00-1C
5	LB Chip1, UART1	1	08-0F	20-3C
6	LB Chip1, UART2	1	10-17	40-5C
7	LB Chip1, UART3	1	18-1F	60-7C
8	LB Chip2, UART0	1	20-27	400-41C
9	LB Chip2, UART1	1	28-2F	420-43C
10	LB Chip2, UART2	1	30-37	440-45C
11	LB Chip2, UART3	1	38-3F	460-47C
12	LB Chip3, UART0	1	40-47	800-81C
13	LB Chip3, UART1	1	48-4F	820-83C
14	LB Chip3, UART2	1	50-57	840-85C
15	LB Chip3, UART3	1	58-5F	860-87C
16	LB Chip4, UART0	1	60-67	C00-C1C
17	LB Chip4, UART1	1	68-6F	C20-C3C
18	LB Chip4, UART2	1	70-77	C40-C5C
19	LB Chip4, UART3	1	78-7F	C60-C7C

Table 18: Access to serial ports in I/O and memory space

6 PCB layout testing

Most pins on the OX16PCI954 can be placed into tristate or input mode, to facilitate PCB testing. This extra function is enabled by applying the following vectors to the pins:

```
Hold TEST=1, MODE[1:0]=’10’, RST#=0
Sequence 0-1-0 on PCI_CLK and hold 0
```

This application will place all the device’s pins in tristate/input mode with the following exceptions:

Number	Name	State
64	XTLO	PERMANENT OUTPUT
71	UART_CLK_OUT	PERMANENT OUTPUT
123	LBRST#	PERMANENT OUTPUT
102	LBDOUT	PERMANENT OUTPUT
109	LBCLK	PERMANENT OUTPUT
112	LBWR#	PERMANENT OUTPUT
113	LBRD#	PERMANENT OUTPUT
41	EE_CK	PERMANENT OUTPUT
39	EE_CS	PERMANENT OUTPUT
40	EE_DO	PERMANENT OUTPUT

7 Programming the OX16C95x UART family

The aim of this section is to build up a library of simple functions (written in C) for accessing the internal OX16C950 UARTs and configuring the various features. All the source code presented in this document is available on disk; please see Section 10 at the end of this document for contact information.

7.1 Fundamental I/O Operations

Each UART consists of 35+ independent registers, yet to maintain backward compatibility with earlier devices, it has 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

7.1.1 Standard Register Access

This section gives details on how to access the OX16C95x standard register-set (550 compatible registers).

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 19: 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 structure's 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 7.2.2 for definition of compatibility modes). 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).

7.1.2 650 Compatible Register Access

This group of registers is used solely for configuring automatic flow control, with the exception of EFR bit 4 which is used to enable Enhanced Mode. Table 20 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 20: 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);
}
```

7.1.3 950 Specific Register Access

This register set consists of four registers, as outlined in Table 21. 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 (see the following section for description).

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 21: 950 Specific Register Set

*Note: Only the bottom two bits of ASR are writable

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;
}
```

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

7.1.4 950 Indexed Control Register Set Access

As its name suggests, this register set is index controlled. 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 17 of these locations are used. 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
0x0E	MDM	Modem Disable Mask	R/W
0x0F	RFC	Readable FIFO Control	R
0x10	GDS	Good-Data Status	R

Table 22: 950 Indexed Control Registers

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 Index 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);
}
```

7.2 Getting Started

7.2.1 Identifying the OX16C950 UART

Identifying an OX16C95x UART 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.). Table 23 shows an example of this, in this case the OX16PCI954 internal UARTs.

Device	ID1	ID2	ID3	REV
OX16C950 Revision B	0x16	0xC9	0x50	0x01

Table 23 : 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;
}
```

7.2.2 Mode Selection

Each UART can be configured into one of several modes to provide backward compatibility with previous UART devices (16C450/550/654 and 750). 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 24: Device Mode Configuration Options

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

NOTE: The FIFOSEL pin is used to select default FIFO depth. When it is tied low and FCR[0] is set, the FIFO depth will be 16. When it is high and FCR[0] is set, the FIFO depth is 128.

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[0] is set:

```
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.

7.2.3 Basic Operation & Configuration

This section highlights the basic configuration required to operate the UARTs. 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 7.3.1). 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, 8 or 9)*
- 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 configurable in a separate register (see Section 7.3.7).

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 25: 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;
}
```

7.2.4 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: For the internal OX16C950 UARTs, the interrupt line is permanently enabled and MCR[3] does not affect it. However, it is usually good practice to set this bit as the driver can then be used for other UARTs, perhaps connected on the local bus of the OX16PCI954.

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");
}
```

7.2.5 Interrupts

The UARTs can be configured to generate interrupts on the events listed below.

- 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 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.

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);
```

7.2.6 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 Level 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 26: 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 7.3.2.

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);
```

7.3 Using Enhanced Features

7.3.1 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. However, the OX16C950 UARTs offer extended flexibility of baud rates 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 7.1.4 for details on how to set ICR registers).

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

Where TCR is the value in the TCR register (4-16) and PRESCALER is the value in the CPR register (1 to 31.875).

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 bit rate 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).

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 MCR[7] is reset to provide compatibility with legacy UARTs

The reset state of the CPR register is 0x20 (divide by 4). Therefore when the prescaler is enabled (MCR[7] is set) the UART can use a 7.3728 MHz clock in place of a 1.8432MHz device, and maintain compatibility with existing software.

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)	CPR value	Effective crystal frequency	Error from 1.8432MHz (%)	Max. Baud rate with CPR = 1, TCR = 16	Max. Baud rate with CPR = 1, TCR = 4
1.8432	0x08 (1)	1.8432	0.00	115,200	460,800
7.3728	0x20 (4)	1.8432	0.00	460,800	1,843,200
14.7456	0x40 (8)	1.8432	0.00	921,600	3,686,400
18.432	0x50 (10)	1.8432	0.00	1,152,000	4,608,000
32.000	0x8B (17.375)	1.8417	0.08	2,000,000	8,000,000
33.000	0x8F (17.875)	1.8462	0.16	2,062,500	8,250,000
40.000	0xAE (21.75)	1.8391	0.22	2,500,000	10,000,000
50.000	0xD9 (27.125)	1.8433	0.01	3,125,000	12,500,000
60.000	0xFF (31.875)	1.8824	2.13	3,750,000	15,000,000

Table 27: Example clock options and their associated maximum baud rates

Using The Times Clock Register (TCR)
The TCR register is used to set the number of channel (internal) clocks per serial bit. The OX16PCI954 internal UARTs allow any value in the range 4-16 for flexible high baud rate generation.

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 offset 2 of the Indexed Control Register set, see Section 7.1.4).

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.

7.3.2 Using 950 Trigger Levels

The UARTs have fully configurable trigger levels for receiver and transmitter interrupts as well as configurable flow control XON and XOFF thresholds. These trigger levels are set using a group of four ICR 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 28: 950 Trigger Level Registers

1. Interrupts must be enabled for these to be asserted on the interrupt pin (see Section 7.2.5)
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 7.1.4. 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.

7.3.3 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.

7.3.4 Using Automated Out-of-band Flow Control

The UARTs can be configured to automatically generate 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 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 7.3.2). For thresholds in other modes, refer to the device data sheet.

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.

7.3.5 Using Automated In-band Flow Control

The UARTs also support 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 29: 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 character should be written to the appropriate registers prior to enabling in-band flow control. For more information on setting flow control characters, see Section 7.1.2.

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.

7.3.6 Using Special Character Detection

The UARTs offer 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 no have to be valid for a special character to be recognised
2. More advanced special character detection is available in nine-bit data mode. For more information see the next section.

7.3.7 Transmitting and Receiving Nine-bit Data

The 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.

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 which 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.

7.3.8 Data Transfer Using an Isochronous Clock

The UART allows for an 'Isochronous' mode of operation whereby data can be received and transmitter using a 1x clock (i.e. baud rate = clock frequency).

The CKS register in the ICR set is used for clock configuration. For a full description of this register, refer to Section 7.11.8 of the data sheet. The UART 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 connecting the clock to the receiving device in parallel with the data line. 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);
}
```

7.3.9 Configuring Automatic RS-485 Buffer Enabling

In systems using the RS-485 protocol, the UARTs 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);
}
```

7.3.10 Enabling Sleep-Mode

The UART supports sleep mode operation 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 (alternate sleep mode). Note that this function is NOT the same as PCI Power Management function, although sleep-mode conditions are the same. 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: Each channel can be put into sleep mode independently. Therefore, to achieve minimal power consumption, all channels should be put into sleep mode.

8 Enhanced features

This section describes the extra features available to developers, such as clock references, power management, reconfiguration using the serial EEPROM and performance-enhancing features like shadowed UART registers.

8.1 PCI Power Management

The OX16PCI954 is compliant with the PCI-Power Management specification v1.0. Using this function, the device can be set to low-power mode whenever its logical functions are idle, to the extent that it no longer needs the PCI clock signal to be active. Low-power state is set by the device driver using a subset of the Configuration space registers. The device can also request to be woken up by asserting the PME# pin.

The Power Management ability of Function0 (internal UARTs) is enabled by setting a filter time in LCC[6:5] and ensuring the interrupt is enabled (GIS[21] set). Then, when all four UARTs are idle, the function will wait for the specified time, and assert the interrupt. Device driver software should then set the PMCSR register to low-power state, causing the device to power down. Activity on any of the UARTs can cause the device to assert an event on the PME# pin. The driver should then set the device back to normal operating state.
There are two “low power” states, D2 and D3 (D0 is normal operating state). In power state D3, only activity on one of the RI# lines will assert a PME# event. In power state D2, the device can be configured to respond to any or all modem and data events.

The Power Management ability of Function1 (Local Bus) is also enabled through the local configuration registers. The functionality will be application-specific, and will require a driver to respond to power-down interrupts in the same way as for function0. If LCC[7] is set, the MIO2 pin is redefined as a PME input, ie when driven active (high or low as defined in MIC[4]), Function 1 will set the PME_Status bit in its PMCSR register and assert the PME# pin.

8.2 Enhanced performance features

The OX16PCI954 provides a set of features that can further enhance the efficiency of a device driver. The Local configuration registers contain shadow copies of the Interrupt status Register and Transmit/Receive FIFO fill levels of the internal UARTs. In addition to this, the GIS register reflects the status of interrupts of the internal UARTs and all the MIO pins. These features can be used in various ways to reduce latency during interrupt service routines, and hence occupy less CPU time servicing the UARTs.

8.2.1 Use of GIS registers

This method involves reading the GIS register on every interrupt to determine the source(s) of the interrupt. In this way it is not necessary to step through every active port and determine whether or not it is in need of servicing. On arrival in the Interrupt Service Routine (ISR), the driver reads the GIS[15:0], and services only the ports with interrupt status set. This is most likely to increase efficiency if there are a large number of ports with reasonably low data rates.

8.2.2 Use of shadowed fill levels and Good-data status

This method is most likely to be efficient if only the internal UARTs are in operation, rather than local bus UARTs. This is because the internal UARTs have the fill levels shadowed in the local configuration registers, and they can therefore be read in a single DWORD access without the overhead of enabling access to the 950-specific register set. If the four UARTs are all in operation at high data rates, it is likely that this method will provide a large increase in efficiency. The suggested procedure is given below:

1) Read FIFO levels from Local configuration registers
2) Read Global Good-Data Status from UIS register (bits 31:27)
3) If bit 31 is set, the contents of the receiver FIFO contain no errors, and therefore the number of bytes indicated in step (1) can be read in a burst access. Also, the transmitter FIFO can be ‘topped up’ from the software buffer. This step can be performed on all four UARTs.
4) If bit 31 is not set, the UARTs must be treated individually as indicated by UIS[30:27]. If, for a given UART, the Good-Data Status bit is not set, the data must be read byte-by-byte in similar fashion to legacy UARTs; however if the correct flow control and interrupt thresholds are set in the Indexed Control registers and the connection is good this should not be a regular occurrence. In any case the access is more efficient than legacy serial ports due to the increased bandwidth of the PCI interface.

8.3 Serial EEPROM

Many of the registers in the OX16PCI954 can be reconfigured with an optional serial EEPROM. A full description of which registers are redefinable is given in the data sheet; however an example program is developed here. Oxford Semiconductor has developed oxprom.exe, an EEPROM programming utility for Windows 95.

8.3.1 Operation

The four interface lines to the serial EEPROM are directly connected to the LCC register (bits 24:27). By toggling these bits in the appropriate manner it is possible to program and read the contents of the EEPROM directly from the PC system used. Please refer to the National Semiconductor® Memory data booklet for control / data specification.

8.3.2 Example program

A valid EEPROM program is defined in four zones. The definition of these zones is shown in Table 30. The program shown in Figure 19 gives an example of a program which alters a set of registers in all three zones. If only one or two zones is set, the contents of Zone 0 (the header) should be set to 0x950n, where n is defined in Table 31.

Data Zone	Size (words)	Description
0	One	Header
1	One or more	Local configuration registers
2	1 - 4	Identification registers
3	Two or more	PCI configuration space registers

Table 30: EEPROM data format

n	Zones to be altered
1	1 only
2	2 only
3	1 and 2
4	3 only
5	1 and 3
6	2 and 3
7	All zones

Table 31: Zone definition

Zone	Address	Value	Description
0	0x01	0x9507	Valid EEPROM header – all EEPROM Zones enabled
1	0x02	0x80E0	LCC[7:2] = '111000'. Another zone1 word follows.
1	0x03	0x8830	LT1[7:0] = 0x30. Another zone1 word follows.
1	0x04	0x8920	LT1[15:8] = 0x20. Another zone1 word follows.
1	0x05	0x8A31	LT1[23:16] = 0x31. Another zone1 word follows.
1	0x06	0x8B20	LT1[31:24] = 0x31. Another zone1 word follows.
1	0x07	0x8E70	LT2[23:20] = '0111'. Another zone1 word follows.
1	0x08	0x0F42	LT2[31:30] = '01', LT2[28:24] = '00010'. End of zone1 data
2	0x09	0x82A7	Subsystem Vendor ID[7:0] = 0xA7. Another zone2 word follows.
2	0x0A	0x0339	Subsystem Vendor ID[15:8] = 0x39. End of zone2 data
3	0x0B	0x8000	Function Header – select function 0. Data follows
3	0x0C	0x8600	Function0 Config: Extended Capabilities: Status[4]='0'. Another function0 word follows.
3	0x0D	0xAC9D	Function0 Config: Subsystem ID[7:0] = 0x9D. Another function0 word follows.
3	0x0E	0x2D84	Function0 Config: Subsystem ID[15:8] = 0x84. End of function0 config data
3	0x0F	0x8001	Function Header – select function 1. Data follows
3	0x10	0x8243	Function1 Config: DeviceID[7:0]=0x43. Another function1 word follows.
3	0x11	0x8379	Function1 Config: DeviceID[15:8]=0x79. Another function1 word follows.
3	0x12	0xAC0B	Function1 Config: Subsystem ID[7:0]=0x0B. Another function1 word follows.
3	0x13	0x2D6E	Function1 Config: Subsystem ID[7:0]=0x6E. End of EEPROM data

Figure 19: Example EEPROM data program

8.4 Clock reference signals

There are two clock reference pins which can be enabled in the Local Configuration Registers and used by local bus devices.
The UART_Clk_Out pin provides a buffered UART clock (as input on the XTLI pin). Using this, a multi-port serial application can be designed using only one oscillator circuit. To enable the UART_Clk_Out pin, set the LCC[2] bit in the Local Configuration registers.
The LBCLK pin provides a buffered PCI clock, and should be used as a PCI clock reference instead of a direct connection to the PCI interface, which would violate layout requirements and may cause erroneous operation. To enable the LBCLK pin, set the LT2[30] bit in the Local Configuration registers.

9 Performance enhancements

A set of performance measurements have been done to demonstrate the increased efficiency of the OX16PCI954 over equivalent solutions using a PCI bridge and Quad UART. These were performed in Windows NT4 Workstation, using the Oxford Semiconductor reference driver version 2.21 and a serial port bandwidth utility (ComTach) from Equinox®. Using the Windows System Monitor it is possible to measure the CPU bandwidth occupied while performing file transfers on a number of ports. The results obtained are shown in Figure 20.

Number of ports / **Baud rate**	**1**	**4 (internal)**	**4 (local bus)**	**8 (4/4)**	**16 (8/8)**
115.2k	1 ***2***	5 ***7***	5 ***8***	11 ***15***	21 ***30***
230.4k	2 ***3***	10 ***13***	10 ***15***	19 ***30***	41 ***61***
460.8k	6 ***7***	18 ***27***	19 ***31***	37 ***59***	84 ***-***
921.6k	10 ***14***	37 ***55***	38 ***61***	78 ***-***	▨

Figure 20: Performance of OX16PCI954

Pentium II, 400MHz CPU usage (%)
Pentium, 200MHz CPU usage (%)

Similar tests on a PLX9050 + OX16C954 solution yielded the following results:-

Number of ports	**Description**	**CPU usage (%)**
2	PLX PCI9050 + OX16C954	10
2	OX16PCI954 internal UARTs	5
4	PLX PCI9050 + OX16C954	20
4	OX16PCI954 internal UARTs	11

Figure 21: Performance compared with discrete solution

10 CONTACT INFORMATION

For further information please contact:

Oxford Semiconductor Ltd.
69 Milton Park
Abingdon
Oxfordshire
OX14 4RX
United Kingdom

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

11 DISCLAIMER