Generic UART Model
Datasheet

Rev. 1.0.2 – 11/05/2007

Author: Bert van Moll

Table Of Contents

1 Introduction

This document describes a generic UART model. This model was created based on the common functionality that was found in three different UART IPs. For now, the generic UART model contains only the basic functionality of a UART. Only the most important features of an UART are implemented. Additional features can be added to the model later.

The model is created using the SCML, OSCI TLM and NXP GMFL library components on top of the standard SystemC features. The generic UART model was first created using only SystemC, SCML and OSCI TLM features. Features from the NXP GMFL library were added later.

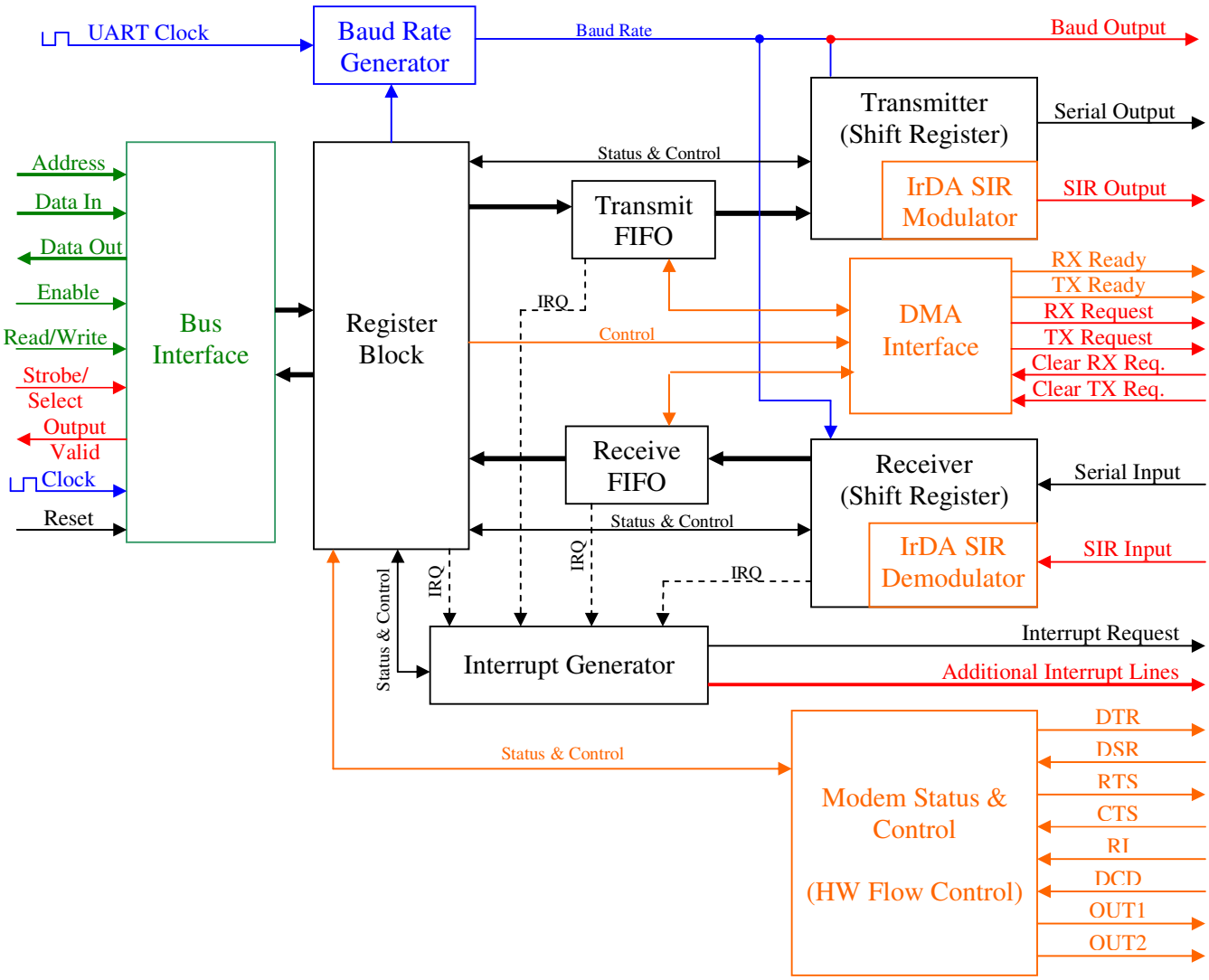
2 UART Features

Figure 1 on the next page shows a schematic diagram of all features that were found in common in the UART specifications that were studied. From all the common features that were found, a selection of the most important features was made and these were implemented in the generic UART model. Table 1 shows the common features that are available in an UART IP, and it indicates if a feature is really required for the operation of an UART, of if it is an optional feature that could be added at a later point in time.

Table 1: *UART Feature Table*

Feature:	***Necessity:***	***Generic UART Model:***
Bus Interface	Required	Implemented.
Register Block	Required	Implemented.
Baud Rate Generator	Optional	Not implemented.
Receive FIFO	Recommended	Implemented. Can be disabled.
Transmit FIFO	Recommended	Implemented. Can be disabled.
Receiver Logic	Required	Implemented.
Transmitter Logic	Required	Implemented.
Interrupt Generator	Recommended	Implemented. Can be disabled.
Modem	Optional	Not implemented.
IrDA Modulation	Optional	Not implemented.
DMA Interface	Optional	Not implemented.
Flow Control	Optional	Not implemented.

Figure 1: *Schematic of all common features found in UARTs*



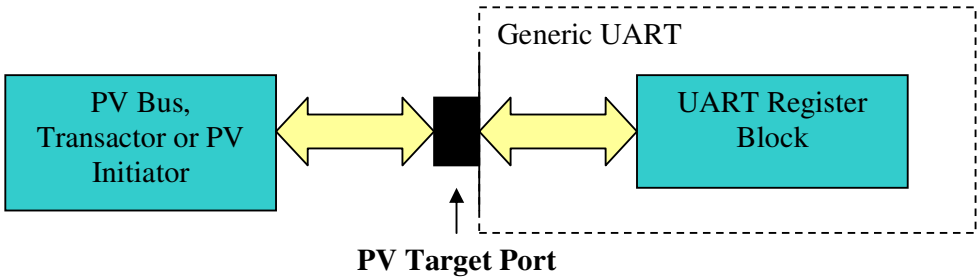
Color	*Meaning*
Black	**Necessary Common Features**
Green	**Common, but implementation specific feature.**
Blue	**Timing Specific Features**
Orange	**Optional Common Features**
Red	**Non-common Features**

The following sections describe each feature in more detail and note how the feature was implemented, or why the feature was left out of the generic UART model.

2.1 Bus Interface

The bus interface is needed to connect the UART to a bus interconnect. This interface is specific for the bus that the UART is connected to. All the UART IP specifications that were studied implemented a different type of bus interface. To make this interface generic for the generic UART model, a SCML PV Target port is used to communicate with the bus. A transactor could be used to connect the generic UART to a specific bus infrastructure.
Figure 2 shows the implementation of the PV Target Port in the generic UART model.

Figure 2: *A PV Target Port is used to communicate between UART and the rest of a system.*



2.2 Register Block

A register block is needed to control the UART functionality through the bus interface. The registers control the operation and indicate the status of the UART. For the generic UART model, only a few registers are needed to control the entire UART. All registers are 32 bits wide. A detailed description of the registers is shown later in this document.

2.3 Baud Rate Generator

Since the generic UART is modelled as a PV model without timing, there is no clock signal in the model. This means that the baud rate generator, which divides the clock frequency to provide the correct baud rate, is not needed in the un-timed generic UART model.

2.4 Receive FIFO

The receive FIFO buffers the incoming serial data. While it is not really necessary to have this FIFO buffer, it is a feature that is nice to have because it reduces the risk that data is lost when a new data frame is received but the previous data has not been read by the system yet.
The generic UART model has a receive FIFO. This FIFO can be disabled, by clearing the FIFO Enable bit in the FCR register. The default size of the receive FIFO is 16 elements, but this can be changed at compile time.
When the interrupt is enabled, an interrupt will be triggered when the contents of the receive FIFO reach the trigger level. This trigger level can be set in the FCR register.

2.5 Transmit FIFO

The transmit FIFO buffers the data that is coming from the system. When the transmit logic is ready, data will be read from the transmit FIFO and sent out as a serial data frame. This FIFO is not vital to the operation of an UART, but it is a nice feature to

have because it allows the system to write multiple data characters to the UART without having to wait before each character is sent.
The generic UART model implements a transmit FIFO buffer. This FIFO can be disabled, by clearing the FIFO Enable bit in the FCR register. The default size of the receive FIFO is 16 elements, but this can be changed at compile time.
When the interrupt is enabled, an interrupt will be triggered when the contents of the transmit FIFO drop below the trigger level. This trigger level can be set in the FCR register.

2.6 Receiver Logic

To receive incoming serial data, some form of serial logic is needed in an UART. In the generic UART model, the receiver logic gets the incoming serial data frame and puts the data in the receive FIFO (if the FIFOs are enabled). When an error in the incoming data frame is detected (Overflow Error, Parity Error, Framing Error or Break Indication), the receive logic will set the line status flags in the LSR register and trigger an interrupt.

2.7 Transmitter Logic

An UART needs transmitter logic to send serial data. The transmit block of the generic UART model reads a data character from the transmit FIFO (or the THR register if FIFOs are disabled), puts it into a serial character frame and sends it out.

2.8 Interrupt Generator

The Interrupt Generator will assert the interrupt line of the UART, when the status of the UART indicates that an interrupt should be given. It is not required for the basic operation of an UART that interrupts are generated, but it is a good feature to have because it allows the UART to notify the system it is connected to when it needs attention.
The generic UART model can indicate three different interrupts. The interrupt status can be read from the IIR register. Each of the three interrupts can be enabled or disabled by writing to the IER register.

2.9 Modem

The Modem block of an UART provides additional hardware signals for synchronising and controlling data flow between two serial devices. Since it is not really necessary for the basic operation of the UART, and because the generic model is modelled at a higher abstraction level, the modem feature is not implemented in the generic UART model.

2.10 IrDA Modulation

Serial Infra-Red (SIR) Data modulation and demodulation allows an UART to generate and receive IrDA pulses for connecting to an infrared transceiver module. This feature is not needed for the operation of an UART, so it is not implemented in the generic UART model.

2.11 DMA Interface

A DMA Interface allows an UART to communicate directly with an external DMA controller to place read and write requests. This is an optional feature and not implemented in the generic UART model.

2.12 Flow Control

Flow control can be used to let the UART control its own data flow. Hardware and/or software flow control allows an UART to send and receive signals from the serial device it is connected to and ‘automatically’ stop or start the sending or receiving of serial data.

No form of flow control is implemented in the generic UART model, since it is not a vital feature for the operation of an UART.

3 Register description

3.1 Register map

Table 2 shows the complete register map of the generic UART model.

Table 2: *Generic UART Memory Map*

Address Index:	*Byte Address:*	*Name:*	*Description:*	*Read/Write:*
0	0x00	RBR	Receiver Buffer Register	Read Only
0	0x00	THR	Transmitter Holding Register	Write Only
1	0x04	IER	Interrupt Enable Register	Read & Write
2	0x08	IIR	Interrupt Identification Register	Read Only
2	0x08	FCR	FIFO Control Register	Write Only
3	0x0C	LCR	Line Control Register	Read & Write
4	0x10	-	Reserved	Read & Write
5	0x14	LSR	Line Status Register	Read & Write

3.2 Receiver Buffer Register

The Receiver Buffer Register (RBR) is located on byte address 0x00. It is read-only; writing to this address will access the THR register. Table 3 shows the bit definition of this register.

Table 3: *Receive Buffer Register bit definition*

Bit Range	Description	R/W/RW	Reset Value
31:8	Reserved	R	0x00
7:0	Data Character	R	0x00

When reading from this register, the UART will check if the FIFOs are enabled (FCR[0] = 1). If FIFOs are enabled, the first element in the FIFO queue is returned. If the FIFOs are not enabled, the value of the RBR register is returned.

3.3 Transmitter Holding Register

The Transmitter Holding Register (THR) is located on byte address 0x00. It is write-only; reading from this address will access the RBR register. Table 4 shows the bit definition of this register.

Table 4: *Transmitter Holding Register bit definition*

Bit Range	Description	R/W/RW	Reset Value
31:8	Reserved	R	0x00
7:0	Data Character	R	0x00

If FIFOs are enabled (FCR[0] = 1), writing to this address will put the data in the transmit FIFO. If FIFOs are not enabled (FCR[0] = 0), data is placed in the THR register instead.

3.4 Interrupt Enable Register

The Interrupt Enable Register (IER) is located on byte address 0x04. Read and write access is possible to the IER register. Table 5 shows the bit definition of this register.

Table 5: *Interrupt Enable Register bit definition*

Bit Range	Description	R/W/RW	Reset Value
31:8	Reserved	RW	0x00
7:3	Reserved for future interrupts	RW	0x00
2	Enable Line Status Interrupt	RW	0x00
1	Enable THR Empty Interrupt	RW	0x00
0	Enable Receive Data Available Interrupt	RW	0x00

The IER masks the Line Status, THR Empty and Receive Data Available interrupts. An interrupt can be enabled, by writing a '1' to the corresponding bit in the IER register. Only enabled interrupts can trigger the UART interrupt line and show up in the IIR register.

3.5 Interrupt Identification Register

The Interrupt Identification Register (IIR) is located on byte address 0x08. It is read-only; writing to this address will access the FCR register. Table 6 shows the bit definition of this register.

Table 6: *Interrupt Identification Register bit definition*

Bit Range	Description	R/W/RW	Reset Value
31:8	Reserved	R	0x00
7:6	Same as FCR[0]	R	0x00
5:4	Reserved for future use	R	0x00
3:1	Interrupt Identification	R	0x00
0	No Interrupt Pending	R	0x01

The IIR register indicates what interrupt, if any, is pending. If bit 0 is '1', no interrupt is pending. If it is '0', bits 3:1 indicate what kind of interrupt is pending. The three possible interrupts are coded with their own interrupt ID and priority as shown in Table 7.

Table 7: *Interrupt ID*

IIR[5:0] ID:	Priority:	Interrupt:	Source:
000001	-	No Interrupt Pending	
000110	1 (high)	Line Status Interrupt	BI, FE, PE or OE set in LSR
000100	2	Receive Data Available	Data received, or receive FIFO trigger level reached.
000010	3 (low)	THR Empty Interrupt	THR Empty, or transmit FIFO trigger level reached.

Line Status Interrupt.
The Line Status Interrupt is triggered when a line status error is detected in the receiver logic. Reading the LSR register will clear this interrupt.

Receive Data Available Interrupt.
If the FIFOs are enabled (FCR[0] = 1), this interrupt is generated when the receive FIFO is filled to its trigger level (see FCR[7:6]). It can then be cleared by reading from the RBR register until the contents of the receive FIFO drop below its trigger level.
If the FIFOs are not enabled (FCR[0] = 0), this interrupt is generated when the RBR register contains data. Reading from the RBR register will clear the interrupt.

THR Empty Interrupt.
If the FIFOs are enabled, this interrupt is generated when the contents of the transmit FIFO drop below its trigger level (see FCR[5:4]). Writing to the THR until the transmit FIFO is above the trigger level will clear the interrupt.
If FIFOs are not enabled, this interrupt indicates that the Transmitter Holding Register is empty. Writing data to the THR will clear the interrupt. Reading the IIR register when this interrupt is pending also clears the interrupt.

3.6 FIFO Control Register

The FIFO Control Register (FCR) is located at byte address 0x08. This register is write-only; when reading from this address, the IIR register is accessed. Table 8 shows the bit definition of this register.

Table 8: *FIFO Control Register bit definition*

Bit Range	Description	R/W/RW	Reset Value
31:8	Reserved	W	0x00
7:6	Receiver FIFO Trigger Level	W	0x00
5:4	Transmitter FIFO Trigger Level	W	0x00
3:1	Reserved for future use	W	0x00
0	Enable FIFOs	W	0x01

The FCR register controls the receive and transmit FIFOs. Bit 0 has to be set to '1' to enable the FIFOs. If FCR[0] = 0, the other bits in the FCR register have no use.
The receiver FIFO trigger level bits (FCR[7:6]) and transmitter FIFO trigger level bits (FCR[5:4]) determine the trigger level at which the UART will generate an interrupt when accessing the FIFOs.
Table 9 and Table 10 show the value of the trigger level bits, and what FIFO level they represent.

Table 9: *Receive FIFO Trigger level control bits*

FCR[7:6]	Trigger Level	Default (FIFO size = 16)
00	1	1
01	¼ of FIFO size	4
10	½ of FIFO size	8
11	FIFO size – 2	14

Table 10: *Transmit FIFO Trigger level control bits*

FCR[5:4]	Trigger Level	Default (FIFO size = 16)
00	1	1

01	¼ of FIFO size	4
10	½ of FIFO size	8
11	FIFO size – 2	14

3.7 Line Control Register

The Line Control Register (LCR) is located at byte address 0x0C. The register allows both read and written access. Table 11 shows the bit definition of this register.

Table 11: *Line Control Register bit definition*

Bit Range	Description	R/W/RW	Reset Value
31:8	Reserved	RW	0x00
7	Reserved for future use	RW	0x00
6	Set Break (SB)	RW	0x00
5	Stick Parity (SP)	RW	0x00
4	Even Parity Select (EPS)	RW	0x00
3	Parity Enabled (PE)	RW	0x01
2	Number of Stop Bits (STB)	RW	0x00
1:0	Word Length Select (WLS)	RW	0x03

The LCR register controls the format of the serial data frame output of the UART.

Bit 6 can be set to force a break condition. Writing a '1' to this bit will cause the start bit of the serial data output frame to be set to '0'.

Bits 5:3 determine the parity options of the UART. Table 12 shows the possible parity mode options:

Table 12: *Parity Mode Options*

PE - LCR[3]	EPS – LCR[4]	SP – LCR[5]	Description
0	**X**	**X**	Parity is disabled. No parity information will be sent or checked.
1	**0**	**0**	Odd parity mode. Data and parity bits will have an odd number of 1s.
1	**1**	**0**	Even parity mode. Data and parity bits will have an even number of 1s.
1	**0**	**1**	Stick 1 mode. Parity is always 1.
1	**1**	**1**	Stick 0 mode. Parity is always 0.

Bit 2 indicates the number of stop bits that should be used when sending a character frame. However, this functionality has not yet been implemented in the generic UART model, so the UART always sends a single stop bit.

Bits 1:0 determine the length of the data word in the serial character frame. Table 13 shows the valid options.

Table 13: *Character Word Length options*

FCR[1:0]:	Character Word Length:

00	5 bits
01	6 bits
10	7 bits
11	8 bits

3.8 Line Status Register

The Line Status Register (LSR) is located on byte address 0x14. This register is read-only. Table 14 shows the bit definition of this register.

Table 14: *Line Status Register bit definition*

Bit Range	Description	R/W/RW	Reset Value
31:8	Reserved	R	0x00
7:6	Reserved for future use	R	0x00
5	Transmitter Holding Register Empty	R	0x01
4	Break Indication (BI)	R	0x00
3	Framing Error (FE)	R	0x00
2	Parity Error (PE)	R	0x00
1	Overrun Error (OE)	R	0x00
0	Data Ready (DR)	R	0x00

If FIFOs are enabled, the Transmitter Holding Register Empty (THRE) bit (LSR[5]) is set to ‘1’ when the contents of the transmit FIFO drop below the trigger level of the transmit FIFO. The bit is cleared when the at least one byte is written to the FIFO. When the FIFOs are disabled, the THRE bit is set to ‘1’ when the THR register is empty and ready to accept new data. It is cleared when data is written to the THR register.

The Break Indication bit (LSR[4]) is set to ‘1’ when the last serial character frame that was received by the UART had its start bit set to ‘0’.

The Framing Error bit (LSR[3]) is set to ‘1’ when the last serial character frame that was received by the UART had its stop bit set to ‘0’.

The Parity Error bit (LSR[2]) is set to ‘1’ when the parity of the last serial character frame that was read by the UART does not match the parity mode set in the LCR register.

The Overrun Error bit (LSR[1]) is set to ‘1’ when FIFOs are enabled and a character frame is received while the receive FIFO is full, or when FIFOs are disabled and a new character frame is received while the previous character has not yet been read from the RBR register. In both cases, the incoming character will be lost.

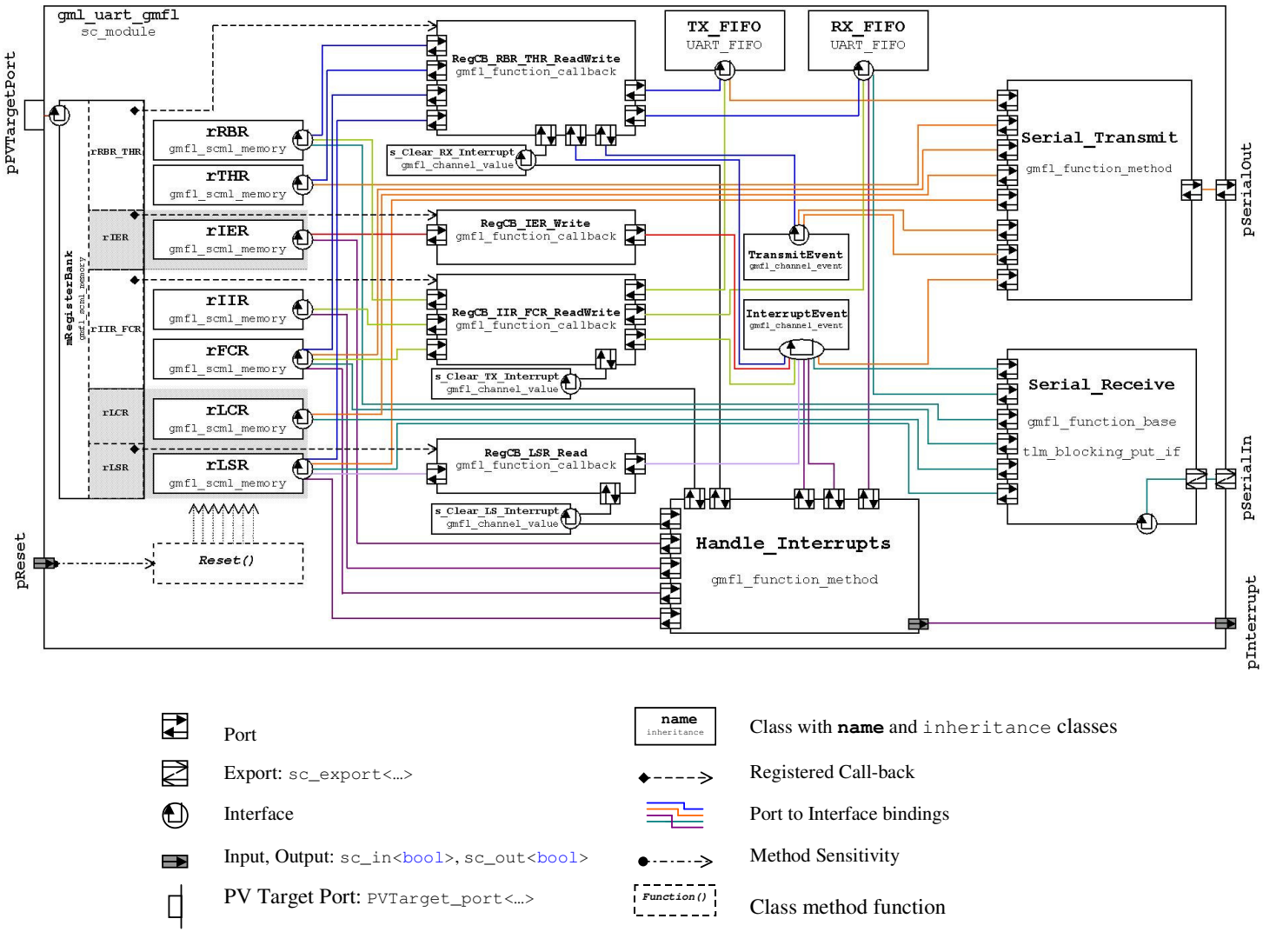
The Data Ready bit (LSR[0]) is set to ‘1’ when there is data available in either the RBR register or the receive FIFO. It is cleared when the RBR register is read or all the contents of the receive FIFO have been read.

4 Architecture and Operation

4.1 Overview

The generic UART model consists of several different module classes. Most of these classes are either instances of, or classes derived from, GMFL feature classes. The different modules are connected together by port-to-interface bindings. Figure 3 shows a block diagram of the generic UART model.

Figure 3: *Block Diagram of the generic UART model.*

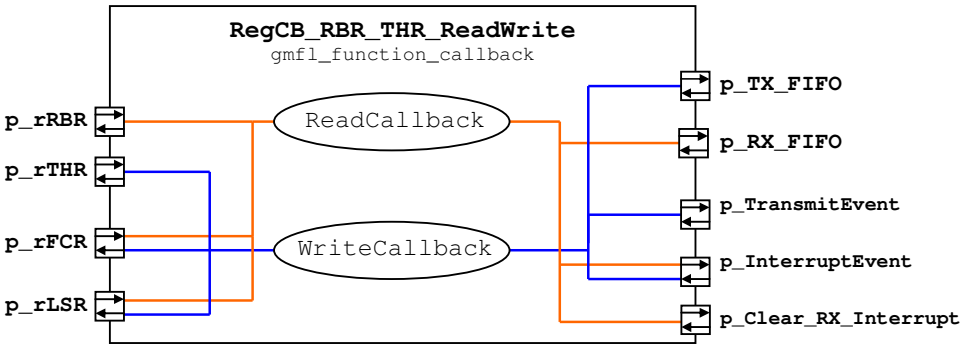


The following sections describe the module class blocks that are not just instances of a GMFL feature class.

4.2 RBR and THR register callback block

The RBR and THR register callback block is a class that derives from *gmfl_function_callback*. This block implements the non-blocking read and write callback functions that are attached to the RBR/THR register alias in the register bank. Figure 4 shows the block diagram of this *RegCB_RBR_THR_ReadWrite* class.

Figure 4: *The RBR and THR register read/write callback class*



4.2.1 Port description

Table 15 lists the ports that are implemented in the RBR and THR register callback block.

Table 15: *RegCB_RBR_THR_ReadWrite class port description*

Name:	Type:	Description:
p_rRBR	gmfl_scml_memory_port<unsigned int>	Connects to the RBR register
p_rTHR	gmfl_scml_memory_port<unsigned int>	Connects to the THR register
p_rFCR	gmfl_scml_memory_port<unsigned int>	Connects to the FCR register
p_rLSR	gmfl_scml_memory_port<unsigned int>	Connects to the LSR register
p_TX_FIFO	FIFO_port<unsigned int>	Connects to the Transmit FIFO
p_RX_FIFO	FIFO_port<unsigned int>	Connects to the Receive FIFO
p_InterruptEvent	gmfl_channel_event_notify_port	Used to notify the Interrupt Event
p_TransmitEvent	gmfl_channel_event_notify_port	Used to notify the Transmit Event
p_Clear_RX_Interrupt	gmfl_channel_value_port<bool>	Signal to clear the receive interrupt.

4.2.2 Function description

```
void WriteCallback(unsigned int WriteData, unsigned int AccessSize,
                                           unsigned int Offset)
```

This function is registered as the non-blocking write callback of the **rRBR_THR** register alias in the **mRegisterBank** memory.
This function first checks if the FIFOs are enabled, by reading the FIFO enable bit from the FCR register via the **p_rFCR** port. If FIFOs are enabled, data is written to the **p_TX_FIFO** port, if FIFOs are diabled, data is written to the **p_rTHR** port.
The THR Empty bit in the LSR register is cleared via the **p_rLSR** port, to indicate that data has been written.
Finally, the events connected to **p_TransmitEvent** and **p_InterruptEvent** are notified to trigger the transmitter and interrupt handler blocks.

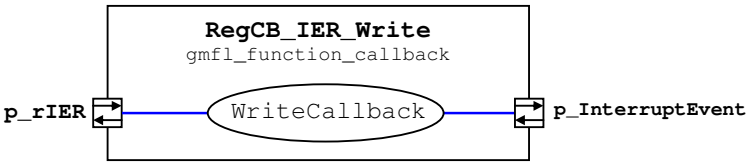
```
unsigned int ReadCallback(unsigned int AccessSize, unsigned int
Offset)
```

This function is registered as the non-blocking read callback of the **rRBR_THR** register alias in the **mRegisterBank** memory.
This function first checks if the FIFOs are enabled, by reading the FIFO enable bit from the FCR register via the **p_rFCR** port. If FIFOs are enabled, data is read from the **p_RX_FIFO** port, if FIFOs are disabled, data is read from the **p_rRBR** port.
If the FIFO is empty after the read, the Data Ready (DR) bit in the LSR register is cleared via the **p_rLSR** port. If the receive FIFO drops below its trigger level or if the FIFOs are disabled, the **p_Clear_RX_Interrupt** port is asserted to clear the receive interrupt.
Finally, the event connected to **p_InterruptEvent** is notified to trigger the interrupt handler.

4.3 IER Register Write Callback block

The IER register write callback block is a class that derives from *gmfl_function_callback*. This block implements the non-blocking write callback function that is attached to the IER register alias in the register bank. Figure 5 shows the block diagram of this *RegCB_IER_Write* class.

Figure 5: *The IER write callback class*



4.3.1 Port description

Table 16 lists the ports that are implemented in the IER register write callback block.

Table 16: *RegCB_IER_Write class port description*

Name:	Type:	Description:
p_rIER	gmfl_scml_memory_port<unsigned int>	Connects to the IER register
p_InterruptEvent	gmfl_channel_event_notify_port	Used to notify the Interrupt Event

4.3.2 Function description

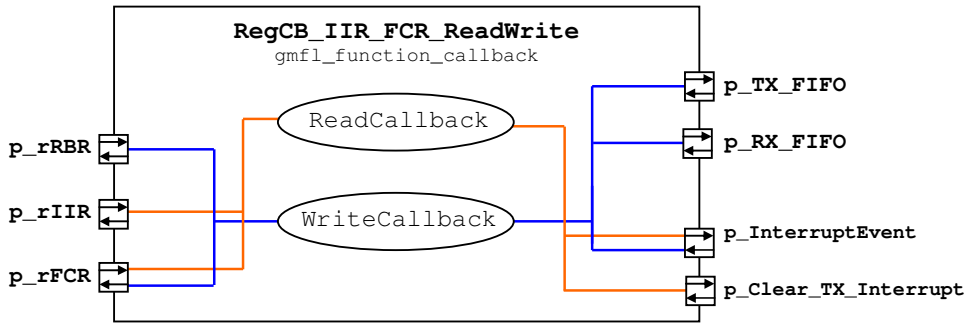
```
void WriteCallback(unsigned int WriteData, unsigned int AccessSize,
                                           unsigned int Offset)
```

This function is registered as the non-blocking write callback of the **rIER** register alias in the **mRegisterBank** memory block.
This function simply writes the data to the **p_rIER** port and notifies the event connected to the **p_InterruptEvent** port to update the interrupts.

4.4 IIR and FCR register callback block

The IIR and FCR register callback block is a class that derives from *gmfl_function_callback*. This block implements the non-blocking read and write callback functions that are attached to the IIR/FCR register alias in the register bank. Figure 6 shows the block diagram of this *RegCB_IIR_FCR_ReadWrite* class.

Figure 6: *The IIR and FCR register read/write callback class*



4.4.1 Port description

Table 17 lists the ports that are implemented in the IIR and FCR register read/write callback block.

Table 17: *RegCB_IIR_FCR_ReadWrite class port description*

Name:	Type:	Description:
p_rRBR	gmfl_scml_memory_port<unsigned int>	Connects to the RBR register
p_rIIR	gmfl_scml_memory_port<unsigned int>	Connects to the IIR register
p_rFCR	gmfl_scml_memory_port<unsigned int>	Connects to the FCR register
p_TX_FIFO	FIFO_port<unsigned int>	Connects to the Transmit FIFO
p_RX_FIFO	FIFO_port<unsigned int>	Connects to the Receive FIFO
p_InterruptEvent	gmfl_channel_event_notify_port	Used to notify the Interrupt Event
p_Clear_TX_Interrupt	gmfl_channel_value_port<bool>	Signal to clear the transmit interrupt.

4.4.2 Function description

```
void WriteCallback(unsigned int WriteData, unsigned int AccessSize,
                                                    unsigned int Offset)
```

This function is registered as the non-blocking write callback of the **rIIR_FCR** register alias in the **mRegisterBank** memory block.
When writing data to the FCR register, this function checks if the FIFOs are being enabled or disabed. If they are being enabled, the trigger levels of the receive and transmit FIFOs are set by writing to the **p_RX_FIFO** and **p_TX_FIFO** ports. If the FIFOs are being disabled, this function moves the top item from the receive FIFO to the RBR register via the **p_rRBR** port, provided the receive FIFO is not empty.
Finally the written data is stored in the FCR register via the **p_rFCR** port and the event connected to **p_InterruptEvent** is notified.

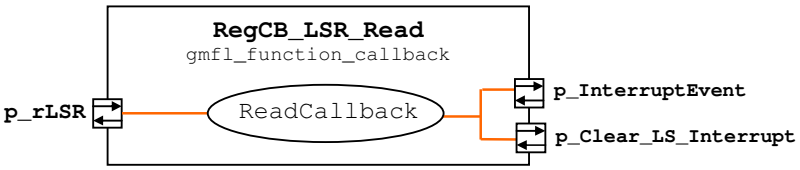
```
unsigned int ReadCallback(unsigned int AccessSize, unsigned int
                                                            Offset)
```

This function is registered as the non-blocking read callback of the **rIIR_FCR** register alias in the **mRegisterBank** memory block.
If this function is called, and the IIR register (read via the **p_rIIR** port) indicates a transmitter empty interrupt, the **p_Clear_TX_Interrupt** port is asserted to clear the THR empty interrupt.
The FIFO Enable bit from the FCR register is read from the **p_rFCR** port, this bit is returned as bit 6 and 7 of the IIR register.
The event connected to **p_InterruptEvent** port is notified to update the interrupts.

4.5 LSR Register Read Callback block

The LSR register read callback block is a class that derives from *gmfl_function_callback*. This block implements the non-blocking read callback function that is attached to the LSR register alias in the register bank. Figure 7 shows the block diagram of this *RegCB_LSR_Read* class.

Figure 7: *The LSR read callback class*



4.5.1 Port description

Table 18 lists the ports that are implemented in the LSR register read callback block.

Table 18: *RegCB_LSR_Read class port description*

Name:	Type:	Description:
p_rLSR	gmfl_scml_memory_port<unsigned int>	Connects to the LSR register
p_InterruptEvent	gmfl_channel_event_notify_port	Used to notify the Interrupt Event
p_Clear_LS_Interrupt	gmfl_channel_value_port<bool>	Signal to clear the line status interrupt.

4.5.2 Function description

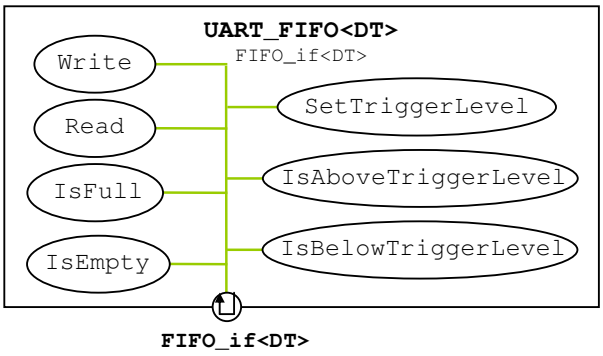
```
unsigned int ReadCallback(unsigned int AccessSize, unsigned int
                                                          Offset)
```

This function is registered as the non-blocking read callback of the **rLSR** register alias in the **mRegisterBank** memory block.
This function returns the data from the LSR register, read via the **p_rLSR** port. It clears the line status interrupt by writing to the **p_Clear_LS_Interrupt** port, and then notifies the event connected to the **p_InterruptEvent** port to update the interrupts.

4.6 Transmit and Receive FIFO blocks

The Transmit and Receive FIFOs are instances of a custom FIFO class called *UART_FIFO*. This class implements the **FIFO_if<DT>** interface. Other modules connecting to the interface of this class should do so using a **FIFO_port<DT>** port.

Figure 8: *The UART_FIFO class*



4.6.1 FIFO Interface description

Table 19 lists the interface methods that are available through the the **`FIFO_if<DT>`** interface. The class template parameter **`DT`** defines the data type of the FIFO. The generic UART model uses unsigned integer as the data type.

Table 19: *FIFO_if<data_type> interface methods*

Name:	Return Type:
`read()`	data_type
`write(data_type v)`	void
`SetTriggerLevel(int level)`	void
`IsEmpty()`	bool
`IsFull()`	bool
`IsAboveTriggerLevel()`	bool
`IsBelowTriggerLevel()`	bool

4.6.2 FIFO Methods description

```
UART_FIFO<DT>(sc_module_name N, int pSize)
```

Constructor of the `UART_FIFO` class. The class parameter `DT` defines the data type of the FIFO. The constuctor arguments `N` and `pSize` set the name and the size of the FIFO, respectively. On construction, the trigger level of the FIFO is set to 1.

```
void write(DT WriteData)
```

If the FIFO is not full, this function pushes the `WriteData` element onto the internal FIFO queue.

```
DT read()
```

If the FIFO is not empty, this function pops the front item from the FIFO queue and returns it.

```
void SetTriggerLevel(int level)
```

This function sets the trigger level of the FIFO. The value has to be between 1 and FIFO Size – 2.

```
bool IsEmpty()
```

Returns true if FIFO is empty.

```
bool IsFull()
```

Returns true if FIFO is full.

```
bool IsAboveTriggerLevel()
```

Returns true if FIFO is above (or at) its trigger level.

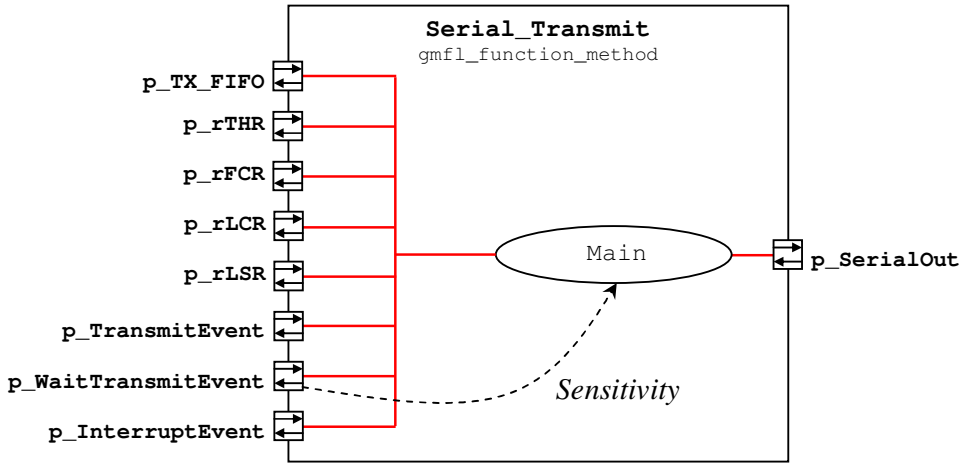
```
bool IsBelowTriggerLevel()
```

Returns true if FIFO is below its trigger level.

4.7 Serial Transmit block

The Serial Transmit block is a class that derives from *gmfl_function_method*. This block implements the serial transmit logic of the UART needed to send serial data. Figure 9 shows the block diagram of this *Serial_Transmit* class.

Figure 9: *The Serial Transmit function method class*



4.7.1 Port description

Table 20 lists the ports that are implemented in the Serial Transmit block.

Table 20: *Serial_Transmit class port description*

Name:	Type:	Description:
p_TX_FIFO	FIFO_port<unsigned int>	Connects to the Transmit FIFO
p_rTHR	gmfl_scml_memory_port<unsigned int>	Connects to the THR register
p_rFCR	gmfl_scml_memory_port<unsigned int>	Connects to the FCR register
p_rLCR	gmfl_scml_memory_port<unsigned int>	Connects to the LCR register
p_rLSR	gmfl_scml_memory_port<unsigned int>	Connects to the LSR register
p_TransmitEvent	gmfl_channel_event_notify_port	Used to notify the Transmit Event
p_WaitTransmitEvent	gmfl_channel_event_wait_port	Used to 'wait' for the Transmit Event
p_InterruptEvent	gmfl_channel_event_notify_port	Used to notify the Interrupt Event
p_SerialOut	sc_port< tlm::tlm_blocking_put_if <SERIAL_DATA_STRUCTURE> >	Serial Data output port.

4.7.2 Function description

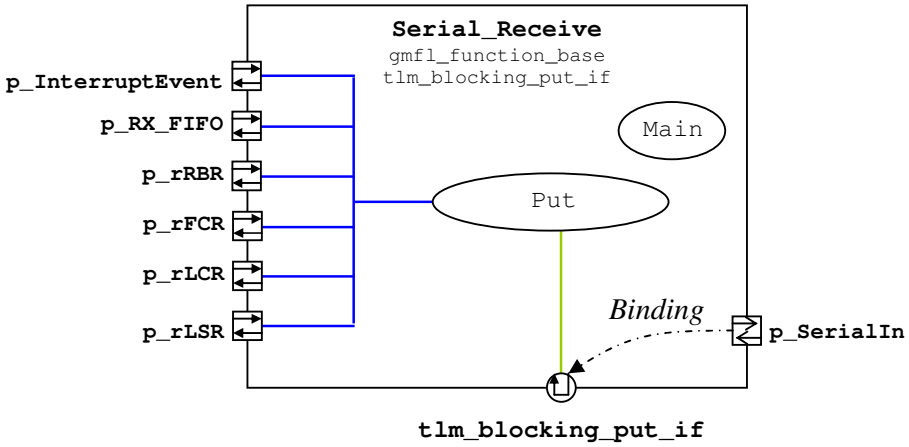
```
void main()
```

This function is made sensitive to the event connected to the **p_WaitTransmitEvent** port.
This function checks if the FIFOs are enabled, by reading from the **p_rFCR** port. If they are enabled and the transmit FIFO is not empty, data is read from the FIFO via the **p_TX_FIFO** port. If FIFOs are disabled, data is read from the THR register via the **p_rTHR** port instead.
If, after reading from the FIFO or the THR, the FIFO drops below its trigger level, or the THR becomes empty, the THR Empty bit in the LSR register is set via the **p_rLSR** port.
After the data has been read, a character frame is created using the settings read from the LCR register via the **p_rLCR** port. After all values of the character frame have been set, the frame is sent to the **p_SerialOut** port using the *put* interface method call of the *tlm_blocking_put_if* interface.
Finally, the events connected to the **p_TransmitEvent** and **p_InterruptEvent** ports are notified to trigger the Serial Transmit block again in case more data is available, and trigger the Interrupt Handler to update the interrupts.

4.8 Serial Receive block

The Serial Receive block is a class that derives from *gmfl_function_base* and *tlm_blocking_put_if*. This block implements the serial receive logic of the UART needed to receive serial data. Figure 10 shows the block diagram of this *Serial_Receive* class.

Figure 10: *The Serial Receive function method class*



4.8.1 Port description

Table 21 lists the ports that are implemented in the Serial Receive block.

Table 21: *Serial_Receive class port description*

Name:	Type:	Description:
p_InterruptEvent	gmfl_channel_event_notify_port	Used to notify the Interrupt Event
p_RX_FIFO	FIFO_port<unsigned int>	Connects to the Receive FIFO
p_rRBR	gmfl_scml_memory_port<unsigned int>	Connects to the RBR register
p_rFCR	gmfl_scml_memory_port<unsigned int>	Connects to the FCR register
p_rLCR	gmfl_scml_memory_port<unsigned int>	Connects to the LCR register
p_rLSR	gmfl_scml_memory_port<unsigned int>	Connects to the LSR register
p_SerialIn	sc_export< tlm::tlm_blocking_put_if <SERIAL_DATA_STRUCTURE> >	Serial data input port, bound to tlm interface

4.8.2 Function description

```
void main()
```

This function is empty and not used.

4.8.3 Interface Method description

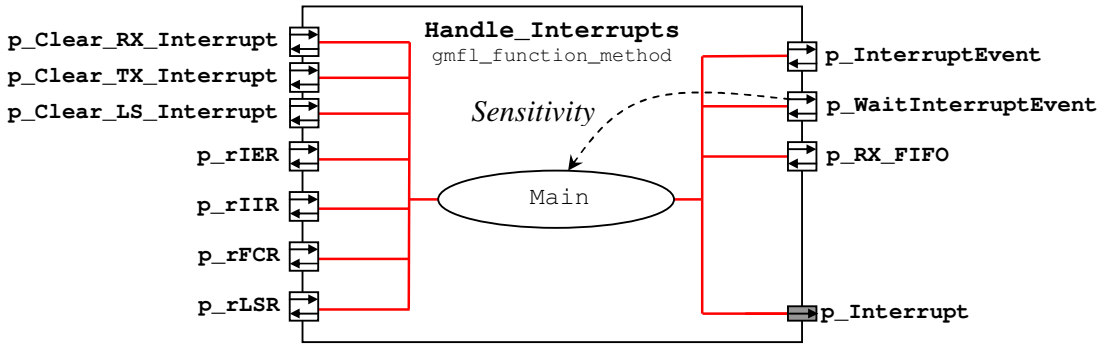
```
void put(const SERIAL_DATA_STRUCTURE &DataStruct)
```

This is the *put* interface method of the *tlm_blocking_put_if* interface, to which the **p_SerialIn** export is bound.
This function checks if the FIFOs are enabled, by reading from the **p_rFCR** port. If they are enabled and the receive FIFO is not full, data from the **DataStruct** structure is written to the FIFO via the **p_RX_FIFO** port. If FIFOs are disabled, the data is written to the RBR register via the **p_rRBR** port.
If there are any Line Status errors, the appropriate Overflow Error, Parity Error, Framing Error or Break Indicator bit is set in the LSR register via the **p_rLSR** port. Finally, the event connected to the **p_InterruptEvent** port is notified to trigger the Interrupt Handler to update the interrupts.

4.9 Handle Interrupts block

The Handle Interrupts block is a class that derives from *gmfl_function_method*. This block checks the status of the UART, and sends an interrupt if needed. Figure 11 shows the block diagram of this *Handle_Interrupts* class.

Figure 11: *The Handle Interrupts function method class*



4.9.1 Port description

Table 22 lists the ports that are implemented in the Handle Interrupts block.

Table 22: *Handle_Interupts class port description*

Name:	Type:	Description:
p_Clear_RX_Interrupt	gmfl_channel_value_port<bool>	Signal to clear the receive interrupt
p_Clear_TX_Interrupt	gmfl_channel_value_port<bool>	Signal to clear the transmit interrupt
p_Clear_LS_Interrupt	gmfl_channel_value_port<bool>	Signal to clear the line status interrupt
p_rIER	gmfl_scml_memory_port<unsigned int>	Connects to the IER register
p_rIIR	gmfl_scml_memory_port<unsigned int>	Connects to the IIR register
p_rFCR	gmfl_scml_memory_port<unsigned int>	Connects to the FCR register
p_rLSR	gmfl_scml_memory_port<unsigned int>	Connects to the LSR register
p_InterruptEvent	gmfl_channel_event_notify_port	Used to notify the Interrupt Event
p_WaitInterruptEvent	gmfl_channel_event_wait_port	Used to 'wait' for the Interrupt Event
p_RX_FIFO	FIFO_port<unsigned int>	Connects to the Receive FIFO
p_Interrupt	sc_out<bool>	Interrupt signal port

4.9.2 Function description

```
void main()
```

This function is made sensitive to the event connected to the **p_WaitInterruptEvent** port.
If any of the **p_Clear_RX_Interrupt**, **p_Clear_RX_Interrupt** or **p_Clear_LS_Interrupt** lines are asserted and there is a matching interrupt pending, the interrupt is cleared and the clear interrupt line is deasserted.
The IER register is read via the **p_rIER** port to see which interrupts are enabled.

If the line status interrupt is enabled, the OE, PE, FE and BI bits are read from the **p_rLSR** port. If any of these are set, the **p_Interrupt** port is asserted.

If the receiver interrupt is enabled the FCR register is checked by reading the **p_rFCR** port, to determine if the FIFOs are enabled. If enabled, the level of the receive FIFO is checked via the **p_RX_FIFO** port and an interrupt is send if the level is above the trigger level. If the FIFOs are not enabled, an interrupt is generated if the Data Ready bit of the LSR register is set.

If the transmitter interrupt is enabled, the THR Empty bit of the LSR register is checked via the **p_rLSR** port. If it is set, an interrupt is generated.

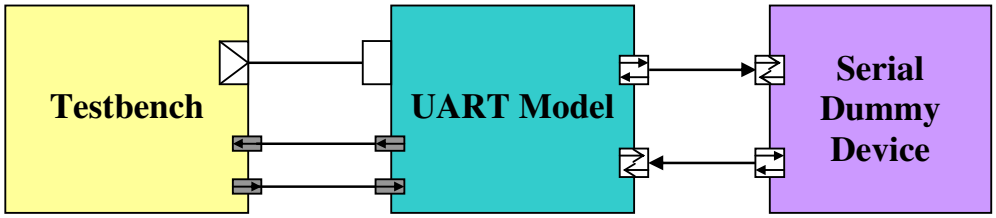
After an interrupt has been generated, the IIR register is updated via the **p_rIIR** port.

If an interrupt with a higher priority than the interrupt currently pending is detected, the interrupt line is held low for one delta cycle before becoming high again.

5 Test Setup

The generic UART model has been tested with a simple test set-up, shown in Figure 12. A testbench is connected to the PV port and reset and interrupt ports of the UART. The serial input and output is connected to a Serial Dummy Device, which does nothing more than loop-back the serial data from the output to the input.

Figure 12: *Test set-up of the UART Model*

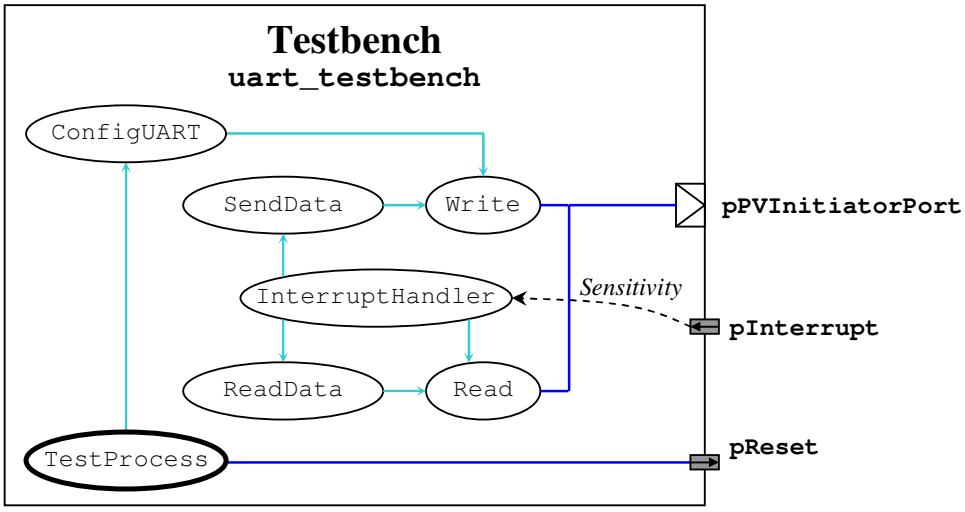


The following sections describe the Testbench and Serial Dummy Device modules in more detail.

5.1 Testbench

The testbench block is an instance of the custom made **uart_testbench** module class. This class was created for the purpose of testing the functionality if the generic UART model. The testbench has one thread and one method, which is sensitive to the interrupt port of the module. The thread (*TestProcess*) configures the UART at the beginning of the test, and validates the data at the end of the test. The method (*InterruptHandler*) does the actual reading and writing of data to the UART, based on the incomming interrupts. Figure 13 shows the schematic of the **uart_testbench** class.

Figure 13: *UART Testbench class*



5.1.1 Port description

Table 23 lists the ports that are implemented in the Testbench block.

Table 23: *uart_testbench class port description*

Name:	Type:	Description:
pPVInitiatorPort	PVInitiator_port<DATA_TYPE, ADDRESS_TYPE>	Used to communicate with the UART at the PV level
PInterrupt	sc_in<bool>	Incoming interrupt signal port
pReset	sc_out<bool>	Reset signal port

5.1.2 Function description

```
void TestProcess()
```

This function is the main thread of the Testbench class. It starts by asserting the **pReset** port to clear the registers of the UART. Then the UART is configured by calling the `ConfigUART()` function. After the UART is configured, the thread waits for 1 ns to make sure no more interrupts are being received. Finally the data that was send to the UART is compared with the data that was received from the UART. If the data is the same, there was no error.

```
void ConfigUART()
```

This function writes data to the FCR and IER registers to configure the UART. By default, the receive and transmit FIFOs are enabled, both with a trigger level of 1. Also, all three interrupt mask bits in the IER register are set. These options can be changed at compile time to test different settings.

```
void InterruptHandler()
```

The `InterruptHandler()` method function provides the main functionality of the Testbench. This method is sensitive to the positive edge of the interrupt signal connected to the **PInterrupt** port. When an interrupt is received, first the IIR register of the UART is read to determine the ID of the interrupt.
In case a Line Status interrupt is detected, the LSR register is read to get the specific line status error and to clear the interrupt. If it concerns an Overflow Error, data is read from the UART. If it is an Parity Error, the SystemC simulation is aborted.
In case of a receive buffer full interrupt, data is read from the UART by calling the `ReadData(…)` function.
In case of a transmit buffer empty interrupt, data is written to the UART by calling the `SendData(…)` function.
If the IIR register indicates no interrupt when this function is called, an error message is displayed.

```
void ReadData(int NrOfReads = 1)
```

This function reads a number of data elements (equal to the `NrOfReads` parameter) from the RBR register address of the UART by making several calls to the `Read(…)` function. All received elements are stored in a buffer.

```
void SendData(int NrOfWrites = 1)
```

This function writes a number of data elements (equal to the `NrOfReads` parameter) to the THR register address of the UART by making several calls to the `Write(…)` function. The elements are read from a buffer.

```
bool Write(PVInitiator_port<DATA_TYPE, ADDRESS_TYPE> *Port,
ADDRESS_TYPE address, DATA_TYPE *data, int BurstCount=1)
```

Generic PV write function. Sends a PV write request with the address, data source and burstcount values copied from the `address`, `*data` and `BurstCount` function arguments to the PV Initiator Port defined by the `*Port` pointer. The function returns true if the response was OK.

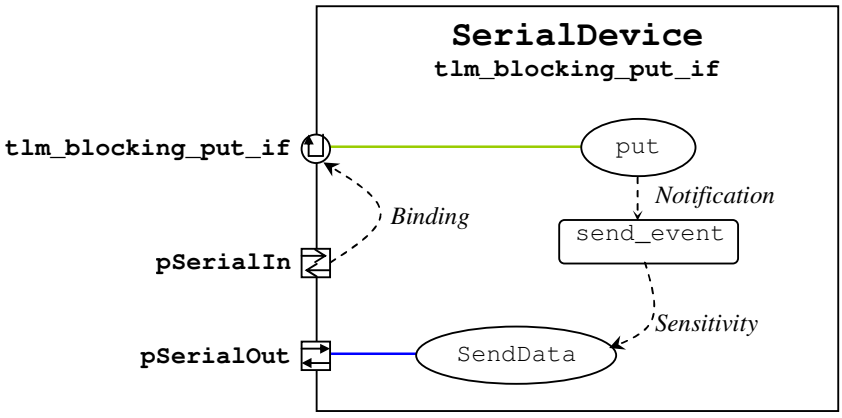
```
bool Read(PVInitiator_port<DATA_TYPE, ADDRESS_TYPE> *Port,
ADDRESS_TYPE address, DATA_TYPE *data, int BurstCount=1)
```

Generic PV read function. Sends a PV read request with the address, data destination and burstcount values copied from the `address`, `*data` and `BurstCount` function arguments to the PV Initiator Port defined by the `*Port` pointer. The function returns true if the response was OK.

5.2 Serial Dummy Device

The Serial Dummy Device is a custum class named **`SerialDevice`**. It implements the *`tlm_blocking_put_if`* interface, which is exported via the export. Data put to the **`pSerialIn`** port, is send back out the **`PSerialOut`** port. Figure 14 shows the diagram of this *`SerialDevice`* class.

Figure 14: *SerialDevice class*



5.2.1 Port description

Table 24 lists the ports that are implemented in the Serial Dummy Device block.

Table 24: *SerialDevice class port description*

Name:	Type:	Description:
pSerialIn	`sc_export< tlm_blocking_put_if <SERIAL_DATA_STRUCTURE> >`	Serial data input port, bound to tlm interface
PSerialOut	`sc_port< tlm_blocking_put_if <SERIAL_DATA_STRUCTURE> >`	Serial data output port

5.2.2 Function description

```
Void SendData()
```

This function is sensitive to the `send_event` event. It sends the current data frame to the **PSerialOut** port.

5.2.3 Interface Method description

```
Void put(const SERIAL_DATA_STRUCTURE &DataStruct)
```

This is the *put* interface method of the *tlm_blocking_put_if* interface, to which the **p_SerialIn** export is bound.
This function stores the **DataStruct** data structure in a local variable, and notifies the `send_event` event.

6 List of Figures

7 List of Tables