# LatticeMico UART

The LatticeMico UART is a universal asynchronous receiver-transmitter used to interface to RS232 serial devices. The UART has many characteristics similar to those of the 16450 UART. To preserve FPGA resources, the LatticeMico UART is not identical to the 16450, so it is not source-code-compatible.

## Version

This document describes the 3.8 version of the LatticeMico UART.

## Features

The UART includes the following features:

- WISHBONE B.3 interface. Data busses are 8-bit wide.
- Similar to the NS16450 UART. The optional 16-word-deep FIFO is implemented in the UART when FIFO mode is selected.
- Insertion or extraction of standard asynchronous communication bits (start, stop, and parity) to or from the serial data
- Holding and shifting registers, which eliminate the need for precise synchronization between the CPU (WISHBONE interface) and serial data
- A common interrupt line for all internal UART data and error events. Interrupt conditions include receiver line errors, receiver buffer available, transmit buffer empty, and detection of status flag change.
- Fully prioritized interrupt system control
- Optional modem function (not presently supported)

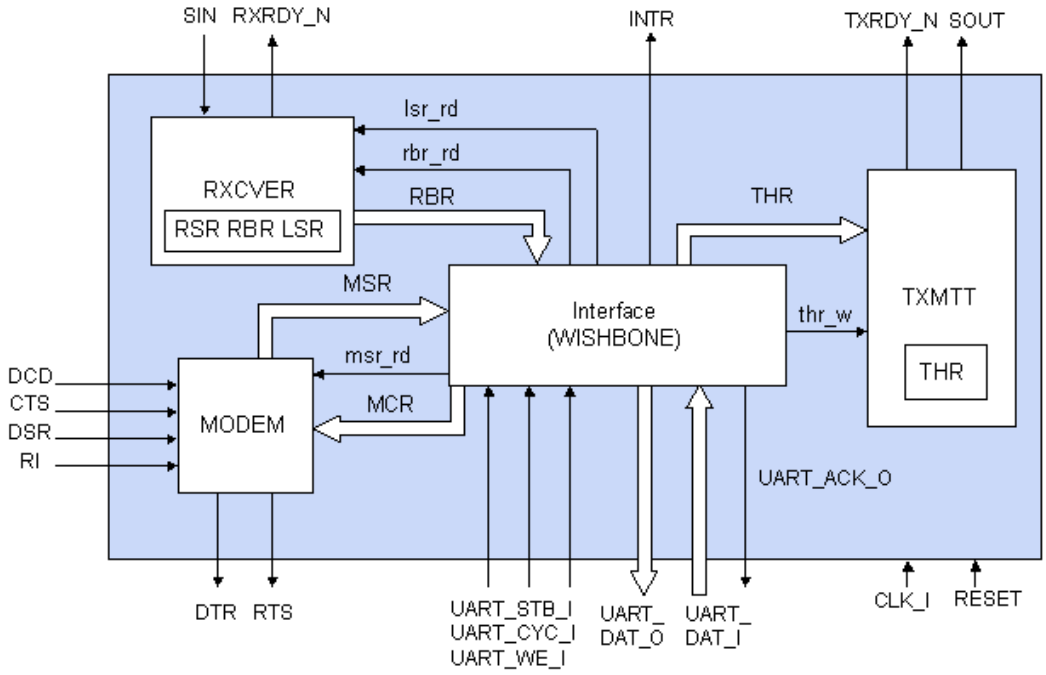
- Support for instantiating the UART multiple times
- Fully programmable serial interface characteristics:
  - 5-, 6-, 7-, or 8-bit characters
  - Even-, odd-, or no-parity bit generation and detection
  - 1-, 1.5-, or 2-stop bit generation and detection
  - False-start bit detection
  - Line-break generation and detection
  - Interactive control signaling and status reporting capabilities

For additional details about the WISHBONE bus, refer to the *LatticeMico8 Processor Reference Manual* or the *LatticeMico32 Processor Reference Manual*.

# Functional Description

The LatticeMico UART provides an interface between the LatticeMico WISHBONE system bus and an RS232 serial communication channel. Figure 1 shows the major blocks implemented in the UART in non-FIFO mode. When you select FIFO mode, RBR in the RXCVER block and THR in the TXMTT block become 16-word-deep FIFOs. In non-FIFO mode, these are simple registers.

**Figure 1: UART Usage Diagram**



# UART Clock Frequencies

The UART only has a single clock input. The UART uses the WISHBONE CLK_I input frequency to transfer data between the LatticeMico embedded microcontroller family and the UART control and status registers. CLK_I is also used to transfer and receive data on the UART's SOUT abd SIN pins.

# Receiver

In non-FIFO mode, the serial receiver (RXCVER) section contains an 8-bit receiver buffer register (RBR) and receiver shift register (RSR). In FIFO mode, the RBR is a 16-word-deep FIFO.

Since the serial frame is asynchronous to the receiving clock, a high-to-low transition of the SIN pin is treated as the start bit of a frame. However, to avoid receiving incorrect data because of SIN signal noise, false-start bit detection is implemented. The UART requires the start bit to be low at least 50 percent of the baud rate clock. The UART samples SIN for half the bit duration, and if a sample is low, a start bit is detected.

Once a valid start bit is received, the data bits and the parity bit are sampled at the RS232 baud rate. If the start bit is exactly equal to the bit duration, each of the following bits will be sampled at the center of the bit itself.

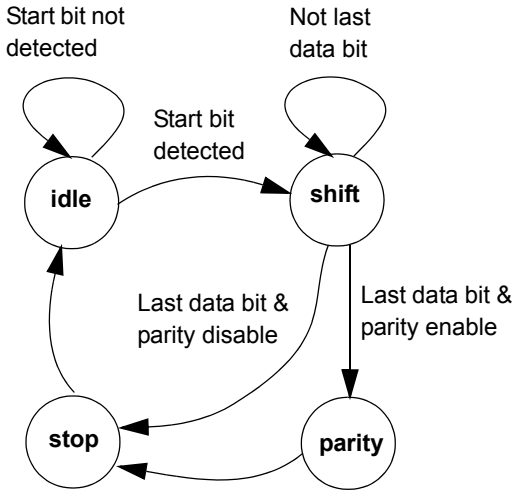
The receive logic monitors the incoming data stream and reports the state of the incoming line in the line status register (LSR). The LSR is used to indicate overrun, parity, and framing errors. It also indicates when a line break has been received.

Whenever a framing error is detected, the UART assumes that the error was due to the start bit of the following frame and tries to resynchronize it. To do this, it samples the start bit twice. If both samples of the SIN are low, the UART resynchronizes and accepts the data following the second start bit. The resynchronization does not occur for a framing error caused by a break character.

Once every bit is received, the data in the receive shift register is transferred to the receive buffer register (RBR). The data-ready bit in the LSR is set to 1 to indicate to the CPU that a byte of data has been received. The external rxrdy_*n* output is asserted (that is, logic 0) simultaneously with the data-ready bit. You can also configure the UART to generate an interrupt upon successful transfer from the RSR to the RBR.

The behavior of the receiver is controlled by the finite state machine (FSM), as shown in Figure 2.

**Figure 2: Receiver State Machine**




The receiver state machine includes the following states:

- idle

  When reset or after the stop state, the receiver FSM is reset to this state. When in this state, it waits for SIN to be changed from high to low and stay low for half a bit duration to be considered a valid start bit. Once a valid start bit is detected, the FSM switches to the “shift” state.

- shift

  When the FSM is in this state, it waits one bit duration for each data bit to shift into the RSR. After the last data bit is shifted in, the FSM switches to the “parity” state if parity is enabled. Otherwise, it switches to the “stop” state.

- parity

  When the FSM is in this state, it waits for one bit duration and then samples the parity bit. Once the parity bit is sampled, the FSM switches to the “stop” state.

- stop

  Whether the stop-bit length is configured to be 1, 1.5, or 2 bits long, the FSM always waits for one bit duration and then samples the stop bit. As long as a logic 1 is sampled at the stop bit, the framing error flag (FE) in LSR is not set. The receiver does not check whether the stop bit is in the right length as configured. The FSM switches back to the “idle” state after sampling the stop bit.

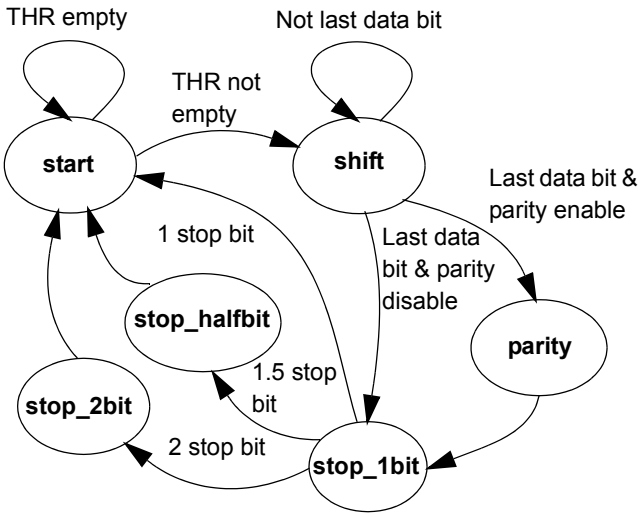
# Transmitter

The serial transmitter (TXMITT) section consists of an 8-bit transmitter hold register (THR) and transmitter shift register (TSR) when the UART is in non-FIFO mode. When the UART is in FIFO mode, THR is a 16-word-deep FIFO. The UART provides two methods of indicating the status of THR: a txrdy_*n* output signal or a transmit holding register ready (THRR) flag in the line status register (LSR). If UART is in non-FIFO mode when THR is empty, the txrdy_*n* pin becomes low active, and the THR empty flag in LSR is set to a logic 1. A write to the THR interferes with a transmission in progress if THRE is active or trdy_*n* is deasserted. After the data is loaded in THR, the THR empty flag in LSR is reset to logic 0, and the txrdy_*n* pin goes inactive high. If the UART is in FIFO mode, when THR FIFO is not full, the txrdy_n pin becomes low active, and the THR Ready flag in LSR is set to logic 1.

The serial data transmission is automatically enabled after the data is loaded into THR. First, a start bit (logic 0) is transmitted and the data in THR is automatically parallel-loaded to TSR. The data bits are shifted out of TSR, followed by the parity bit, if parity is enabled. Finally, the stop bit (logic 1) is generated to indicate the end of the frame. After a frame is fully transmitted, another frame is transmitted immediately if THR is not empty. This automatic sequencing causes the frames to be transmitted back to back, which increases the transmission bandwidth. The SOUT pin is held high when no transmission is in progress.

The behavior of the transmitter is controlled by the finite state machine (FSM), as shown in Figure 3.

**Figure 3: Transmitter State Machine for Non-FIFO Mode**




The transmitter state machine includes the following states:

- start

  When the UART is reset, the FSM transmitter is reset to this state. When in this state, the transmitter waits to assert the start bit. A start bit is

asserted as soon as THR is not empty. Once a low SOUT (start bit) is asserted, the FSM switches to the “shift” state.

- shift

  When the FSM is in this state, it waits for the last (most significant) data bit to be shifted out. After the last data bit is shifted out, the FSM switches to the “parity” state if parity is enabled. Otherwise, it switches to the “stop_1bit” state.

- parity

  When the FSM is in this state, the last data bit is still in transmission. When the transmission is complete, the FSM asserts the parity bit. Once the parity bit is asserted, the FSM switches to the “stop_1bit” state.

- stop_1bit

  Whether the stop bit is configured to be 1, 1.5, or 2 bits long, the FSM always switches to this state, waits for a baud clock cycle, and then asserts the stop bits. For one stop bit, the FSM switches back to the “start” state and waits to assert the start bit of another frame. For 1.5 stop bits, it switches to the “stop_halfbit” state and stays there for just half a baud clock cycle before switching to the “start” state. For two stop bits, it switches to the “stop_2bit” state and then switches back to the “start” state. The stop bit is asserted at the time that the FSM leaves the “stop_1bit” state.

- stop_halfbit

  This state is for 5-bit data bits with a 1.5 stop bit. The FSM stays in this state for only half a baud clock cycle and then switches to the “start” state.

- stop_2bit

  When the FSM is in this state, the first stop bit is in transmission. It waits for a baud clock cycle and then asserts the second stop bit and switches to the “start” state.

# Interrupt

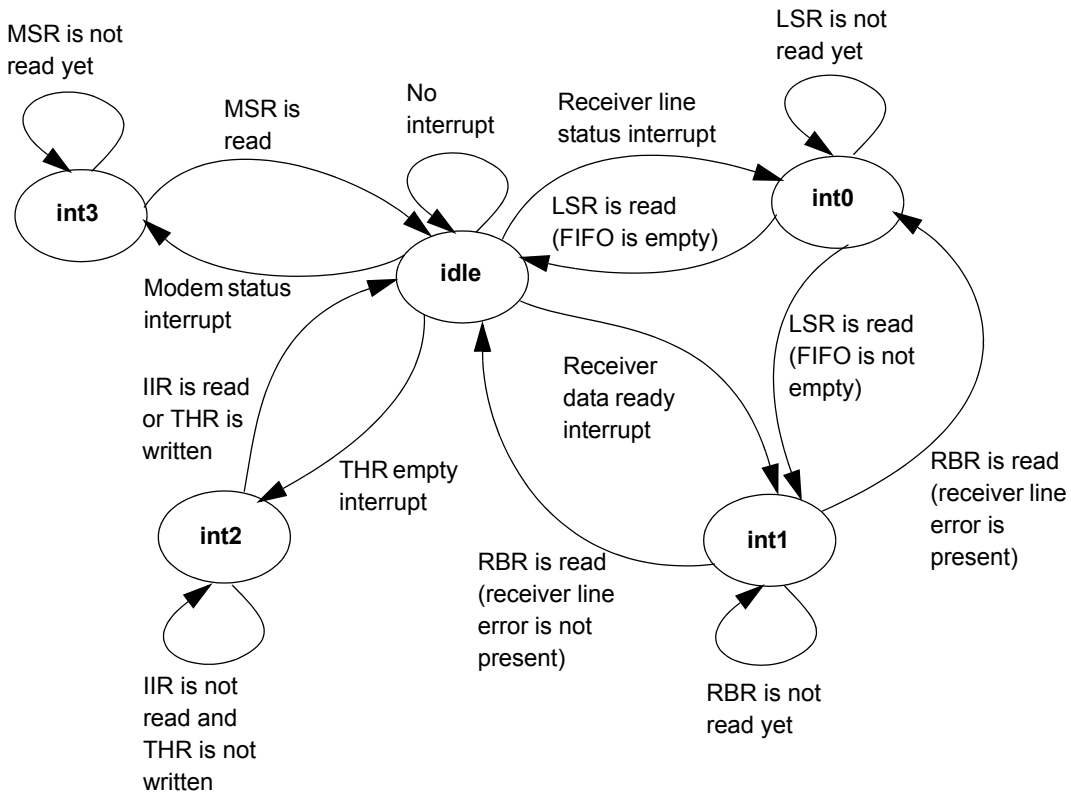
The common interrupt request pin (INTR) goes to high active when any interrupt conditions are matched and enabled by the interrupt enable register (IER), which is described in Table 5.

The UART prioritizes interrupts into four levels to minimize external software interaction, and it records these in the interrupt identification register (IIR), which is described in Table 6. The four levels of interrupt conditions in order of priority are receiver line status, received data ready, transmitter holding register empty, and modem status, which is described in Table 7.

Performing a read cycle on IIR freezes all interrupts and indicates the highest priority pending interrupt to the CPU. No other interrupts are acknowledged until the pending interrupt is serviced. Whenever the IIR is read, the current pending interrupt is cleared. Any pending lower-priority interrupt becomes visible in the IIR after the previous IIR read.

The behavior of the interrupt is controlled by the FSM, as shown in Figure 4.

**Figure 4: Interrupt Behavior for non-FIFO Mode**




The interrupt pin state machine includes the following states:

- idle

  When the UART is reset, the interrupt FSM is reset to this state. When in this state, it waits for the enabled interrupt conditions to be true, and then it switches to the interrupt state with the highest priority.

- int0

  The FSM switches to this state when the highest-priority level interrupt occurs. It stays at this state until the LSR is read.

- int1

  The FSM switches to this state when the second-priority level interrupt occurs. It stays at this state until the RBR is read.

- int2

  The FSM switches to this state when the third-priority level interrupt occurs. It stays at this state until the IIR is read or after THR is written.

- int3

  The FSM switches to this state when the fourth-priority level interrupt occurs. It stays at this state until the MSR is read.

The interrupt continues to be generated as long as the corresponding enable bit in IER is set and the corresponding interrupt condition is matched.

# WISHBONE Interface

The LatticeMico processors interface to the UART control and status registers by using the WISHBONE bus. The registers can only be accessed via the WISHBONE classic bus cycles.

All the control registers are aligned to a byte boundary. Refer to Table 4 on page 12 for more details on the register memory map. The SEL_I input is ignored. All control registers, with the exception of the baudrate divisor register, are 8 bits wide. Therefore, any access to a control register is a byte transfer (C char). The baudrate divisor register is 16 bits wide and can be accessed using two single-byte transfers.

When the microprocessor initiates a bus cycle, it starts by asserting STB_I and CYC_I. The UART responds to the cycle with UART_ACK_O at the first rising edge following the the STB_I and CYC_I assertion. The UART_ACK_O stays active for a single CLK_I cycle, terminating the transfer and accepting or returning data.

# Configuration

The following sections describe the graphical user interface (UI) parameters, the hardware description language (HDL) parameters, and the I/O ports that you can use to configure and operate the LatticeMico UART.

## UI Parameters

Table 1 shows the UI parameters available for configuring the LatticeMico UART through the Mico System Builder (MSB) interface. For the data bit and the stop bit, refer to the LCR register description.

**Table 1: UART UI Parameters**

| Dialog Box Option | Description | Allowable Values | Default Value |
|---|---|---|---|
| Instance Name | Specifies the name of the UART instance. | Alphanumeric and underscores | uart |
| Base Address | Specifies the base address for configuring the UART. The minimum boundary alignment is 0x80. | 0X80000000–0XFFFFFFFF | 0X80000000 |
| **UART Configuration** | | | |
| Baud Rate | Specifies the speed in bits per second. | 9600, 19200, 38400, 57600, 115200 | 115200 |
| Data Bits | Specifies the number of data bits in each transfer. | 6, 7, 8 | 8 |
| Stop Bits | Specifies the number of stop bits in each transfer. | 1, 2 | 1 |
| Parity Enable | Enables the feature of making the number of "one" bits between any start/stop pair odd or even. | 1, 0 | 0 |
| Odd Parity | Ensures that the number of "one" bits between any start/stop pair is always odd. If unselected, the number of "one" bits is always even. | 1, 0 | 0 |
| Stick Parity | Sets the parity to always "one" or always "zero" | 1, 0 | 0 |
| Set Break | Enables the ability to insert a break in transmission. | 1, 0 | 0 |
| Tx/Rx FIFO Enabled | Enables the Tx and Rx FIFO in the UART. | 1, 0 | 0 |
| **Sideband Signals** | | | |
| Receiver Ready | Component generates an active-low signal that indicates data is available in RBR. | 1, 0 | 0 |
| Transceiver Ready | Component generates an active-low signal that indicates that THR is empty and new data is ready to be sent. | 1, 0 | 0 |
| **Software Settings** | | | |

**Table 1: UART UI Parameters (Continued)**

| Dialog Box Option | Description | Allowable Values | Default Value |
|---|---|---|---|
| Use interrupt | Specifies whether the software driver transmits or receives data using interrupts. If you select this option, the software driver transmits or receives data using interrupts. If you do not select it, the software driver polls the UART registers for transmission or reception of data. | 1, 0 | 1 |
| Block on transmit | Specifies whether the driver returns until it is able to queue data for transmission. If true, the driver does not return until it is able to queue data for transmission. | 1, 0 | 1 |
| Block on receive | Specifies whether the driver returns until data is received. If true, the driver does not return until data is received. | 1, 0 | 1 |
| Rx Buffer Size | Specifies the size of the incoming data buffer implemented by the software driver, in bytes. This buffer is used only when operating in interrupt mode. In non-interrupt mode (polling mode), this data buffer is not used; however, the data associated with it is still allocated. | 2-32 | 4 |
| Tx Buffer Size | Specifies the size of the outgoing data buffer implemented by the software driver, in bytes. This buffer is used only when operating in interrupt mode. In non-interrupt mode (polling mode), this data buffer is not used; however, the data associated with it is still allocated. | 2-32 | 4 |
| **Transmit Settings for Simulation** | | | |
| Print Transmit Character in RTL Simulation | Prints the Transmit characters to the console window in the RTL simulator. | 0, 1 | 0 |
| Emulate Transmit Operation | Emulates the Transmit of a character rather than physically transmitting it bit-by-bit on the TX line of the UART. This option is available only when the Transmit characters need to be printed to the console window of RTL simulator and is used when a shorter simulation time is required. | 0, 1 | 0 |

# HDL Parameters

Table 2 lists the parameters that appear in the HDL.

**Table 2: UART HDL Parameters**

| Parameter Name | Description | Allowable Values |
|---|---|---|
| BAUD_RATE | Specifies the speed in bits per second. | 9600, 19200, 38400, 57600, 115200 |
| FIFO | Specifies the use of a 16-word-deep FIFO in the UART | 1 (Yes), 0 (No) |

# I/O Ports

Table 3 describes the input and output ports of the LatticeMico UART.

**Table 3: UART I/O Ports**

| I/O Port | Active | Direction | Initial State | Description |
|---|---|---|---|---|
| RESET | High | I | | Reset signal, active high |
| CLK_I | — | I | | Clock signal 16 times the receiving/transmitting baud rate clock frequency |
| UART_ADR_I [4:0] | — | I | | Address from WISHBONE interface |
| UART_DAT_I [7:0] | — | I | | Data input from WISHBONE interface |
| UART_DAT_O [7:0] | — | O | | Data output to WISHBONE interface |
| UART_SEL_I | — | I | | Select input array, which indicates where the valid data is expected on a data bus |
| UART_STB_I | High | I | | Strobe input indicating that the slave is selected |
| UART_CYC_I | High | I | | Cycle signal indicating that a valid bus cycle is in progress |
| UART_WE_I | — | I | | Write enable input<br>0 = read<br>1 = write |
| UART_BTE_I | High | I | | Burst-type extension |
| UART_ACK_O | High | O | 0 | Acknowledge output, indicating the termination of a normal bus cycle |
| INTR | High | O | 0 | Interrupt request, active high, used to request service from the CPU whenever one of the following four conditions happens:<br>▶ Receiver line errors<br>▶ Receiver buffer available<br>▶ Transmit buffer empty<br>▶ Modem status flag change detected |
| SIN | High | I | — | RS232 serial data input |
| SOUT | High | O | 1 | RS232 serial data output |

# User Impact of Initial State

After reset, the transmitter idles until transmission begins.

# Register Descriptions

The LatticeMico UART includes the registers shown in Table 4:

- Two data buffering registers, RBR and THR
- Three status registers, IIR, LSR, and MSR
- Three control registers, IER, LCR, and MCR
- Baud-rate generator, DIV

**Table 4: Register Map**

| Register Name | Offset | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Receive buffer register/transmit holding register | 0x0 | Data bit 7 | Data bit 6 | Data bit 5 | Data bit 4 | Data bit 3 | Data bit 2 | Data bit 1 | Data bit 0 |
| Interrupt enable register | 0x1 | 0 | 0 | 0 | 0 | MSI | RLSI | THRI | RBRI |
| Interrupt identification register | 0x2 | 0 | 0 | 0 | 0 | 0 | ID1 | ID0 | STAT |
| Line control register | 0x3 | 0 | SB | SP | EPS | PEN | STB | WLS1 | WLS0 |
| Modem control register | 0x4 | 0 | 0 | 0 | 0 | 0 | 0 | RTS | DTR |
| Line status register | 0x5 | 0 | TEMT | THRR | BI | FE | PE | OE | DR |
| Modem status register | 0x6 | DCD | RI | DSR | CTS | DDCD | TERI | DDSR | DCTS |
| Baud-rate divisor register | 0x8 | Divisor bits [7:0]. | | | | | | | |
| Baud-rate divisor register | 0x9 | Divisor bits [15:8] | | | | | | | |

Table 5 through Table 11 provide details about each register in the LatticeMico UART.

## Interrupt Enable Register

The interrupt enable register provides discrete control over each of the independent interrupt sources provided by the UART. When the interrupt is masked, the INTR output is not activated when the masked interrupt source becomes active. An interrupt is masked when the corresponding register bit is set to 0.

**Table 5: Interrupt Enable Register (IER, Addr = 0x04)**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| RBRI | 0 | Write only | Receiver buffer register interrupt (1 = enable, 0 = disable) |
| THRI | 1 | Write only | Transmitter hold register interrupt (1 = enable, 0 = disable) |
| RLSI | 2 | Write only | Receiver line status interrupt (1 = enable, 0 = disable) |
| MSI | 3 | Write only | Modem status interrupt (1 = enable, 0 = disable) |

# Interrupt Identification Register

The interrupt identification register holds information about which interrupt is currently active. Table 7 shows bit encoding for each interrupt source. The THRE interrupt is cleared by either reading the IIR register or writing to the THR FIFO in FIFO mode or THR register in non-FIFO mode. But if the THR FIFO in FIFO mode or the THR register in non-FIFO mode is empty, the THRE interrupt is again asserted by the UART after reading the IIR register, provided that the THRI bit of the IER register is set.

**Table 6: Interrupt Identification Register (IIR, Addr = 0x08)**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| STAT | 0 | Read only | Interrupt state |
| ID1:0 | 2:1 | Read only | Interrupt identification code (see Table 7) |

**Table 7: Four Prioritized Interrupt Levels and Sources**

| Interrupt Level | ID1 | ID0 | STAT | Source of Interrupt | Interrupt Reset Control |
|---|---|---|---|---|---|
| None | 0 | 0 | 1 | No interrupt pending | None |
| Highest | 1 | 1 | 0 | LSR error flags (OE/PE/FE/BI) | Reading LSR |
| Second | 1 | 0 | 0 | LSR receiver data-ready flag. | Reading RBR in non-FIFO mode. In FIFO mode, when FIFO becomes empty. |
| Third | 0 | 1 | 0 | LSR flag THR empty (THRE) | Reading IIR or writing THR |
| Fourth | 0 | 0 | 0 | Modem status | Reading MSR |

# Line Control Register

The line control register is used to define the data protocol between the interconnected serial devices. It controls the number of data bits sent, parity, the number of stop bits, stick parity, and the transmit break bit.

**Table 8: Line Control Register (LCR, Addr = 0x0C)**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| WLS1: | 1:0 | Write only | Defines the data length:<br>▶ 00 – 5 bit<br>▶ 01 – 6 bit<br>▶ 10 – 7 bit<br>▶ 11 – 8 bit |
| STB | 2 | Write only | Stop-bit definition:<br>▶ 0 – 1 stop<br>▶ 1 –<br>WLS1:0 = 00: 1.5 stop<br>WLS1:0 = 01: 2 stop<br>WLS1:0 = 10: 2 stop<br>WLS1:0 = 11: 2 stop |
| PEN | 3 | Write only | Parity enable:<br>▶ 0 – Parity bit disable<br>▶ 1 – Parity bit enable |
| EPS | 4 | Write only | Parity even:<br>▶ 0 – Odd parity selected<br>▶ 1 – Even parity selected |
| SP | 5 | Write only | Stick parity:<br>▶ 0 – Stick parity disable<br>▶ 1 –<br>When PEN, EPS, or SP is set (that is, 1), parity is sent or checked for 0.<br>When PEN or SP is set, EPS is clear, parity is sent or checked for 1. |
| SB | 6 | Write only | Tx break:<br>▶ 0 – Disable break assertion<br>▶ 1 – Assert break. SOUT is driven low active (break character) as long as this bit is 1. It has no effect on the transmitter logic. |

# Modem Control Register

In Table 9, loopback mode for testing (bit 4) and auxiliary user-defined output Out2 (bit 3) and Out1 (bit 2) are removed, because they can be implemented with in-system programmability (ISP).

**Table 9: Modem Control Register (MCR, Addr=0x10)**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| DTR | 0 | Write only | Controls the data-terminal-ready (DTR*n*) output.<br>▶ DTR=0: force DTR*n* output to a logic 1 (normal default)<br>▶ DTR=1: force DTR*n* output to a logic 0 |
| RTS | 1 | Write only | Controls the request-to-send (RTS*n*) output.<br>▶ RTS=0: force RTS*n* output to a logic 1 (normal default)<br>▶ RTS=1: force RTS*n* output to a logic 0 |

# Line Status Register

Table 10 provides information about the line status register.

**Table 10: Line Status Register (LSR, Addr=0x14)**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| DR | 0 | Read only | Receiver data-ready. DR indicates status of RBR. It is set to logic 1 when RBR data is valid and is reset to logic 0 when RBR is empty. When line errors (OE/PE/FE/BI) occur, DR is also set to logic 1, and RBR is updated to reflect the data bits portion of the frame. Pin RxRDY*n* is the complement of this bit. |
| OE | 1 | Read only | Overrun error. This bit is set when the next character is transferred into RBR before the previous RBR data is read by the CPU. The previous RBR data is lost when the overrun occurs. |
| PE | 2 | Read only | Parity error. This bit is set to logic 1 only when the parity is enabled and the parity bit is not at the logic state it should be. For even parity, the parity bit should be 1 if an odd number of 1s in the data bits is received. Otherwise, the parity bit should be 0. For odd parity, the parity bit should be 1 if an even number of 1s in the data bits is received. Otherwise, the parity bit should be 0. |
| FE | 3 | Read only | Framing error. FE is reset to logic 0 whenever SIN is sampled high at the center of the first stop bit, regardless to how many stop bits the UART is configured. |

**Table 10: Line Status Register (LSR, Addr=0x14) (Continued)**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| BI | 4 | Read only | Break interrupt. BI is set to logic 1 whenever SIN is low for longer than a whole data frame (the time of start bit + data bits + parity bit + stop bits). If SIN is still low after BI is reset to logic 0 by reading LSR, BI is not set to logic 1 again. Since break is also a framing error, FE is also set to 1 when BI is set. |
| THRR | 5 | Read only | THR ready. THRE is set to logic 1 whenever THR is ready, which indicates that the transmitter is ready to accept new data to transmit. Pin TxRDYn is the complement of this bit. |
| TEMT | 6 | Read only | Both THR and TSR are empty. This bit is set to logic 1 when THRE is set to 1 and the last data bit in the TSR is shifted out through SOUT. |

The four error flags (OE, PE, FE, and BI) of LSR are reset to logic 0 after a LSR read.

# Modem Status Register

Table 11 provides information about the modem status register.

**Table 11: Modem Status Register (MSR, Addr=0x18)**

| Register Name | Bit | Access Mode | Description |
|---|---|---|---|
| DCTS | 0 | Read only | Delta CTS indicates that the CTS*n* has changed state since the last time it was read by CPU. |
| DDSR | 1 | Read only | Delta DSR indicates that the DSR*n* has changed state since the last time it was read by CP. |
| TERI | 2 | Read only | Trailing edge of RI indicates that the RI*n* input has changed from a low to a high state. |
| DDCD | 3 | Read only | Delta DCD indicates that the DCD*n* input has changed state. |
| CTS | 4 | Read only | Clear to sent is the complement of the CTS*n* input. |
| DSR | 5 | Read only | Data set ready is the complement of the DSR*n* input. |
| RI | 6 | Read only | Ring indicator is the complement of the RI*n* input. |
| DCD | 7 | Read only | Data carrier detect is the complement of the DCD*n* input. |

Whenever bit 0-3 is set to logic 1, a modem status interrupt is generated if the interrupt is enabled. These four bits are reset to logic 0 whenever CPU reads MSR.

## Baud-Rate Divisor Register

The LatticeMico UART includes a baud-rate divisor register, detailed in Table 12.

**Table 12: Baud-Rate Divisor Register**

| Register Name | Bit | Access | Description |
|---|---|---|---|
| DIV | 15:0 | Write read only | The divisor register is used to divide the CLK_I input to generate the baud-rate clock. |

The software can change the value of the divider. The value written to the divider is determined by the following equation:

```
divisor_value = (clk_in_MHz * 1000 * 1000)/baud_rate;
```

The *clk_in_MHz* frequency is in megahertz.

Here is an example:

```
clk_in_mhz= 25 MHz
desired baud_rate = 115200

divider = (25 * 1000 * 1000)/115200 = 217
```

# EBR Resource Utilization

The LatticeMico UART has no EBRs.

# Usage Model

This section describes the software usage model for the LatticeMico UART. The UART device drivers provide a simple and easy-to-use interface to interact with the physical UART device.

The device driver presents the UART as a simple character device from which a designer can either “get” characters or “put” characters from transmission. The UART can be configured through Mico System Builder (MSB) to operate in an interrupt-driven mode or in a polled mode. In the polled mode, the hardware does not raise an interrupt to indicate whether a character has been successfully transmitted/received, or that the transmission/reception has resulted in an error. Therefore, it is the responsibility of the application layer to keep track of the transmission and reception of characters via the UART. In the interrupt-driven mode, the hardware raises an interrupt for every character that is transmitted/received or for errors that may have occured. Therefore, the application layer does not need to continuously monitor the UART.

Interrupt-driven mode is more suitable to an asynchronous mode of operation, where the foreground thread of operation does not need to be synchronized to the transmission and reception of data through the UART. The interrupt-driven mode relies on receive and transmit buffers that are implemented in the software. The size of these buffers is configurable through MSB.

**Note**

The space required for the buffers, which is a minimum of two bytes for transmission and two bytes for reception, is always allocated irrespective of the mode of operation. These allocated buffers are used in interrupt-driven mode but not in polled mode. The compile-time declaration of buffers eliminates the need for performing run-time memory allocation, which allows better memory usage.

## Error Detection and Handling

The UART reports PE (parity error) or FE (framing error) for the character received. However, the detection of the framing error depends on the logic level detected by the UART when expecting the stop bit. Because the detection of the framing error depends on the baud rate between the two UARTs, as well as on the character transmitted by the other end when the UARTS are using different baud rates, it is not very reliable.

When retrieving a character from the UART, the UART software driver will check the LSR (line status register) for the framing and parity errors only if it is operating in interrupt mode. If the FE or PE flag is set, the UART interrupt will ignore the character. Since the FE and PE flags are in the LSR and are reset on reading the LSR, and since in polled-transmit mode the LSR needs to be read by the transmit function, the UART software driver does not check the FE or PE flags when performing a polled receive.

**Note**

While the device driver by itself does not provide for error detection or correction or flow control of the streaming data, a higher-level software protocol can provide these facilities.

# LatticeMico32 Microprocessor Software Support

This section describes the LatticeMico32 microprocessor software support provided for the LatticeMico UART component. It first describes the basic UART device-driver interface and then describes the UART lookup service and the UART file-mode service.

**Note**

The supporting routines are meant for use in a single-threaded environment. If you use them in a multi-tasking environment, you must provide re-entrance protections.

# Register Map Structure

The structure shown in Figure 5 depicts the register map layout for the UART component. The elements are self-explanatory and are based on the register map shown in Table 4. This structure, which is defined in the MicoUART.h header file, enables you to directly access the UART registers, if desired. It is used internally by the device driver for manipulating the UART.

**Figure 5: UART Register Map Structure**

```
typedef struct st_MicoUart {
   volatile unsigned char rxtx;
   volatile unsigned char ier;
   volatile unsigned char iir;
   volatile unsigned char lcr;
   volatile unsigned char mcr;
   volatile unsigned char lsr;
   volatile unsigned char msr;
   volatile unsigned short div;
}MicoUart_t;
```

# Device Driver

The UART device driver interacts directly with the UART instance. This section describes the limitations, type definitions, structure, and functions of the UART device driver.

## Limitations

The UART device driver assumes that the UART component is used as a basic serial transmission/reception device devoid of hardware flow control.

- Software control: any desired software control must be provided by the application.
- Modem functionality: although the UART component has modem functionality, this feature is not supported in the LatticeMico8 UART component driver.

## Type Definitions

This structure, shown in Figure 6, contains the UART component instance specific information and is dynamically generated in the DDStructs.h header file. This information is largely filled in by the managed build process by extracting the UART component-specific information from the platform specification file. You should not manipulate the members directly, because this structure is for exclusive use by the device driver.

**Figure 6: UART Device Context Structure**

```
typedef struct st_MicoUartCtx_t {
const char* name;
unsigned int base;
unsigned char intrLevel;
unsigned char intrAvail;
unsigned int baudrate;
unsigned int databits;
unsigned int stopbits;
unsigned char rxBufferSize;
unsigned char txBufferSize;
unsigned char blockingTx;
unsigned char blockingRx;
unsigned char *rxBuffer;
unsigned char *txBuffer;
DeviceReg_t lookupReg;
unsigned char rxWriteLoc;
unsigned char rxReadLoc;
unsigned char txWriteLoc;
unsigned char txReadLoc;
unsigned int timeoutMicroSecs;
volatile unsigned char txDataBytes;
volatile unsigned char rxDataBytes;
unsigned int errors;
unsigned char ier;
unsigned int fifoenable;
void * prev;
void * next;
} MicoUartCtx_t;
```

Table 13 describes the parameters of the UART device context structure shown in Figure 6.

**Table 13: UART Device Context Structure Parameters**

| Parameter | Data Type | Description |
|---|---|---|
| name | const char * | Instance-specific component name (entered in MSB) |
| base | unsigned int | MSB-assigned base address for this instance |
| intrLevel | unsigned char | Processor interrupt line to which this instance is connected |
| intrAvail | unsigned char | Indicates whether the UART is to be used in interrupt mode (1) or not (0), as configured in MSB |
| baudrate | unsigned int | Baud rate as configured in MSB. It can be modified at run time. |
| databits | unsigned int | Data bits as configured in MSB. The data bits can be changed at run time by using the MicoUart_dataConfig function call. |
| stopbits | unsigned int | Stop bits as configured in MSB. The stop bits can be changed at run time by using the MicoUart_dataConfig function call. |

**Table 13: UART Device Context Structure Parameters (Continued)**

| Parameter | Data Type | Description |
|---|---|---|
| rxBufferSize | unsigned char | Size in bytes of the receive buffer for interrupt-driven operation, as configured in MSB |
| txBufferSize | unsigned char | Size in bytes of the transmit buffer for interrupt-driven operation, as configured in MSB |
| blockingTx | unsigned char | Indicates whether the device driver must operate in blocking mode (1) or in non-blocking mode (0) for transmission |
| blockingRx | unsigned char | Indicates whether the device driver must operate in blocking mode (1) or in non-blocking mode (0) for reception |
| rxWriteLoc | unsigned char | Used internally for recexive buffer management |
| rxReadLoc | unsigned char | Used internally for receive buffer management |
| txWriteLoc | unsigned char | Used internally for transmit buffer management |
| txReadLoc | unsigned char | Used internally for transmit buffer management |
| timeoutMicroSecs | unsigned int | Used internally to detect device errors on transmission |
| txDataBytes | volatile unsigned char | Used internally for transmit buffer management |
| rxDataBytes | volatile unsigned char | Used internally for receive buffer management |
| errors | unsigned int | Not used (reserved for future use) |
| ier | unsigned char | Used internally as a shadow of the UART IER register |
| fifoenable | unsigned int | Indicates whether the UART has a TX/RX FIFO (1) or not (0). |
| prev | void * | Used internally for device lookup service |
| next | void * | Used internally for device lookup service |
| rxBuffer[4] | unsigned char | Internal use for receive buffer implementation. The size of the array is configurable on a per-instance basis through MSB. |

**Table 13: UART Device Context Structure Parameters (Continued)**

| Parameter | Data Type | Description |
|---|---|---|
| txBuffer[4] | unsigned char | Internal use for transmit buffer implementation. The size of the array is configurable on a per-instance basis through MSB. |
| lookupReg | DeviceReg_t | Used by the device driver to register the UART component instance with the LatticeMico lookup service. Refer to the *LatticeMico Software Developer User Guide* for a description of the DeviceReg_t data type. |

# Functions

This section describes the implemented device-driver-specific functions.

## MicoUartInit Function

```
void MicoUartInit(MicoUartCtx *ctx);
```

This function initializes a LatticeMico UART instance on the basis of the passed UART context structure. This initialization function is responsible for the following:

- Initializing the interrupts or buffer parameters for interrupt-driven operation
- Registering the UART instance with UART service

As a part of the managed build process, the LatticeDDInit function calls this initialization routine for each UART instance that is present in the platform.

Table 14 describes the parameter in the MicoUartInit function syntax.

**Table 14: MicoUartInit Function Parameter**

| Parameter | Description |
|---|---|
| MicoUartCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance. |

## MicoUart_setRate Function

```
unsigned int MicoUart_setRate(MicoUartCtx_t *ctx, unsigned int
baudrate);
```

This function sets the baud rate of a UART instance. The software does not check for the validity of the baud-rate value. The function computes the UART divisor value on the basis of the following formula:

```
divisor_value = divisor = cpu_frequency/baud_rate;
```

*Cpu_frequency* is the frequency specified in hertz.

This result is then loaded into the divisor register. It also programs the desired baud rate in the UART control register.

Table 15 describes the parameters in the MicoUart_setRate function syntax.

**Table 15: MicoUart_setRate Function Parameters**

| Parameter | Description | Notes |
|---|---|---|
| MicoUartCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance. | |
| unsigned int | Baud rate | No software check is performed. Depending on the CPU clock speed, the baud rate might result in loss of communication or excessive errors. Consult the component data sheet for more information. |

Table 16 shows the values returned by the MicoUart_setRate function.

**Table 16: Values Returned by the MicoUart_setRate Function**

| Return Value | Description |
|---|---|
| 0 | Successful |
| MICOUART_ERR_INVALID_ARGUMENT | Failure: baud rate value was zero. |

### MicoUart_dataConfig Function

```
unsigned int MicoUart_dataConfig(MicoUartCtx_t *ctx,
                                 unsigned int dwidth,
                                 unsigned char parity_en,
                                 unsigned char even_odd,
                                 unsigned int stopbits)
```

This function changes the following UART data-reception transmission parameters:

- Data width
- Enable parity generation and detection
- Parity selection if parity generation and detection is enabled
- Stop bits

Table 17 describes the parameters in the MicoUart_dataConfig function syntax.

**Table 17: MicoUart_dataConfig Function Parameters**

| Parameter | Description |
|---|---|
| MicoUartCtx_t *ctx | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance |
| unsigned int dwidth | Data width.<br>Allowed values: 5, 6, 7, 8 |
| unsigned int parity_en | Enable parity<br>▶ 0 = Parity disabled<br>▶ 1 = Parity enabled |
| unsigned int even_odd | Parity type:<br>▶ 0 = Odd parity<br>▶ 1 = Even parity<br>**Note:** This parameter is ignored if parity is disabled. |
| unsigned int stopbits | Stop-bits selection:<br>▶ 1 = 1 stop bit<br>▶ 2 = 1.5 or 2 stop bits, depending on the data width |

Table 18 shows the values returned by the MicoUart_dataConfig function syntax.

**Table 18: Values Returned by the MicoUart_dataConfig Function**

| Return Value | Description |
|---|---|
| 0 | Successful |
| MICOUART_ERR_INVALID_ARGUMENT | Failure: bad argument to the function. |

### MicoUart_setBlockMode Function

```
unsigned int MicoUart_setBlockMode(MicoUartCtx_t *ctx, unsigned
int uiBlock);
```

This function changes the device driver mode of operation to blocking mode if it was operating in non-blocking mode. Blocking and non-blocking behavior is discussed in “MicoUart_putC Function” on page 25 and “MicoUart_getC Function” on page 26.

Table 19 describes the parameters in the MicoUart_setBlockMode function syntax.

**Table 19: MicoUart_setBlockMode Function Parameters**

| Parameter | Description |
|---|---|
| MicoUartCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance |
| unsigned int | 0 – Set operation to non-blocking<br>1 – Set operation to blocking |

Table 20 shows the values returned by the MicoUart_setBlockMode function.

**Table 20: Value Returned by the MicoUart_setBlockMode Function**

| Return Value | Description |
|---|---|
| 0 | Successful |

### MicoUart_putC Function

```
unsigned int MicoUart_putC(MicoUartCtx_t *ctx, unsigned char
ucChar);
```

This function posts a character to the UART THR for transmission. The behavior depends on the mode of operation:

- Interrupt-driven blocking mode – This function attempts to add the incoming byte of data to the software-implemented transmit buffer. If the buffer is full, the function waits until space is available in the buffer. When space is available, the incoming byte of data is posted to the software-implemented transmit buffer, and the function returns.
- Non-interrupt-driven blocking mode – The function directly polls the UART status register and waits for the UART transmit-hold register to become empty. Once empty, the function queues the character and returns. If the transmit-hold-register is empty, the function loads the character for transmission and returns.
- Interrupt-driven non-blocking mode – The function queries the software-implemented transmit buffer. If there is space in the buffer, it queues the character and returns with a success status. If there is no space in the buffer, it returns the MICOUART_ERR_WOULD_BLOCK error code.
- Non-interrupt driven non-blocking mode – The function queries the UART line status register and checks the THRE bit to see if the THR is empty. If the THR is empty, it is loaded with the incoming data byte, and the function returns. If the UART status register indicates that the transmit-hold register is not empty, the function returns immediately with the MICOUART_ERR_WOULD_BLOCK error code.

Table 21 describes the parameters in the MicoUart_putC function syntax.

**Table 21: MicoUart_putC Function Parameters**

| Parameter | Description |
|---|---|
| MicoUserCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance |
| unsigned char | Character to queue for transmission |

Table 22 shows the values returned by the MicoUart_putC function.

**Table 22: Values Returned by the MicoUart_putC Function**

| Return Value | Description |
|---|---|
| 0 | Successful |
| MICOUART_ERR_WOULD_BLOCK | UART operates in non-blocking mode, and the function blocks, indicating that there is no room to queue this character. |
| MICOUART_ERR_DEVICE_ERROR | UART operates in blocking mode, and even after a complete word time, there is no space to queue another character, indicating that this is a device error. |

### MicoUart_getC Function

```
unsigned int MicoUart_getC(MicoUartCtx_t *ctx, unsigned char
*pucChar);
```

This function attempts to retrieve a received character from the UART. The exact behavior depends on the operating mode:

- Interrupt-driven blocking mode – The function returns immediately with a received character if there is one waiting in the receive buffer. If there is no character pending in the receive buffer, the function continuously polls the receive buffer for a new character. The function returns when a character is placed into the software-implemented receive buffer.
- Non-interrupt-driven blocking mode – The function returns immediately with the character in the receive-holding register if the UART status indicates there is a character pending. If the UART status indicates there is no character to read, the function continuously polls the UART status register for a new character. The function returns as soon as the RBR is no longer empty.
- Interrupt-driven non-blocking mode – The function returns immediately if there is a new character waiting in the receive buffer. If there is no character waiting in the receive buffer, the function returns immediately with the MICOUART_ERR_WOULD_BLOCK error code.
- Non-interrupt-driven non-blocking mode – The function returns immediately with the character in the UART receive-hold register if the UART status indicates there is a character pending. If there is no

character pending, the function returns with the MICOUART_ERR_WOULD_BLOCK error status.

Table 23 describes the parameters in the MicoUart_getC function syntax.

**Table 23: MicoUart_getC Function Parameters**

| Parameter | Description |
|---|---|
| MicoUartCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance |
| unsigned char* | Pointer to unsigned character location where the read character should be stored |

Table 24 shows the values returned by the MicoUart_getC function.

**Table 24: Values Returned by the MicoUart_getC Function**

| Return Value | Description |
|---|---|
| 0 | Successful |
| MICOUART_ERR_WOULD_BLOCK | UART is operating in non-blocking mode, and the function would block, indicating that there is no character to be read. |

## Services

The UART component software support includes support for the following two types of services:

- Lookup service – UART device driver registers UART instances with LatticeMico lookup service, using their instance names for device names and “UARTDevice” as the device type.

  For information on the LatticeMico lookup service, refer to the *LatticeMico Software Developer User's Guide*.

- File service – Each UART instance makes itself available as a file device to the LatticeMico file services. File device support for the UART is limited to the standard input and output of characters and does not provide an actual file system support. The UART file service has the following limitations:

  - The UART file service does not support an explicit file system, although it does support standard file operations, such as fopen, fprintf, fgets, and fscanf.
  - The UART file system assumes that there is an appropriate connection to the LatticeMico UART instance that provides a console for it to communicate with, for example, the LatticeMico UART instance connected over a serial link to a PC’s serial port, with the PC hosting a serial terminal program such as Hyperlink or TeraTerm.

- For file-read operations such as fscanf and fgets, the UART file service implementation echoes the characters over the transmit channel. This behavior is not modifiable.
- The UART file service read function implementation treats a \r or a \n as an input delimiter character. This behavior is not modifiable.
- The UART file service allows the UART to be treated as a standard I/O stream, provided that it is explicitly set to be used this way through the C/C++ SPE platform configuration tab or at run time programmatically.
- The UART file system support is limited to single-threaded applications.
- The UART file system sets the UART to operate in blocking mode. This behavior cannot be modified.

For information on LatticeMico file services, refer to the *LatticeMico Software Developer User Guide*.

# UART Code Reduction

You can reduce the impact of the UART device driver and services on the overall code size by following these guidelines:

- If appropriate, reduce the transmit and receive buffer sizes to the minimum. You can do this through the MSB user interface.
- If not required, disable the UART file services by defining the following macro in the C/C++ Software Project Environment (SPE) LatticeMico Preprocessor defines:

  `_MICOUART_FILESUPPORT_DISABLED_`

  This macro disables the UART-specific file system registration function, allowing the linker to eliminate all the UART file service software code. However, you can no longer use any UART instance for standard input/output or file operation.
- If none of the UART instances operate using interrupts, you can eliminate the code for interrupt-driven operation by defining the _MICOUART_NO_INTERRUPTS_ preprocessor macro. This change reduces the code size, but the device driver operates on all UART instances present in the system in poll mode.

# Software Usage Examples

This section provides examples of using the UART as a file device and as a standard I/O device at run time.

## Using a UART as a File Device

The default managed build process initializes the UART instance by invoking the UART initialization function described previously in this document. This initialization function makes the UART available to the LatticeMico lookup service, as well as to the LatticeMico file service.

Because the UART instance is available as a file device, it can be used for any other file operation, as shown in the example in Figure 7. In the example, it is assumed that the platform contains a UART named “uart.”

**Figure 7: UART Used as File Device**

```
#include <stdio.h>

int main(void) {

   /*
    * You have a UART instance named "uart" in your platform
    * and you wish to use it as a file device.
    */

   /*
     * STEP 1: "Open" the UART device, using the device-
     * naming convention.
*/

   FILE *fptr = fopen("\\\\uart\\", "wt");
   if(fptr == 0) {
      /* ERROR: failed to open our UART device */
      return(-1);
   }

   /* STEP 2: "Write" some data to our UART device */
   fprintf(fptr, "i wrote bytes to the uart\r\n");
   /* STEP 3: "Close" our device now that we’re done using it.
   */
   fclose(fptr);

   /*
    * Wait for a second just in case the isr is still servicing
    * the transmit buffer, since the UART baud rate is much
    * slower than the CPU clock speed.
   */

   MicoSleepMilliSecs(1000);
   return(0);
}
```

## Setting UART as a Standard I/O Device at Run Time

The UART implementation allows C/C++ SPE to flag it as a device capable of serving standard input/output streams. You can also enable this functionality at run time. The example shown in Figure 8 shows how to set a UART

instance as a standard input/output device and also demonstrates the lookup capability. The example assumes the presence of a UART component named “uart.”

**Figure 8: UART Used as Standard I/O Device**

```
#include <stdio.h>
#include "DDStructs.h"
#include "LookupServices.h"
#include "MicoFileDevices.h"
#include "MicoUtils.h"


int main(void) {
   int i;
   char cBuffer[256];


   /*
    * You have a UART instance named "uart" in your platform
    * and you wish to use it as a standard I/O device for
    * printf and fgets (and scanf, etc.)
   */

   /* STEP1: Fetch UART context */
   MicoUartCtx_t *pUart = (MicoUartCtx_t *)
   MicoGetDevice("uart");
   if(pUart == 0) {
      /* ERROR: Failed to find a named device */
      return(-1);
   }

   /* Set this device as standard input device. Device "0" is
   the stdin file handle.*/
   i = MicoFileRedirIO(0, pUart->name);
   if(i != 0) {
      /* ERROR: Failed to set this device for std input */
      return(-1);
   }

   /* Set this device as standard output device. Device "1" is
   the stdout file handle.*/
   i = MicoFileRedirIO(1, pUart->name);
   if(i != 0) {
      /* ERROR: failed to set this device for std output */
      return(-1);
   }

   /*
    * Read a line from the UART!
    * NOTE: Do not enter more than 255 characters!
    */
   gets(cBuffer);

   /*
    * "Write" over the UART as standard output!
    */
   printf("You just entered a line using UART: %s\r\n",
   cBuffer);
```

**Figure 8: UART Used as Standard I/O Device (Continued)**

```
    /*
     * Wait for a second just in case the isr is still servicing
     * the transmit buffer, since the uart buad rate is much
     * slower than the CPU clock speed
     */
    MicoSleepMilliSecs(1000);

    return(0);
}
```

## Using the Device Driver API Directly

The “UartEcho” template provided with the LatticeMico System software demonstrates the use of the raw device driver API for manipulating the UART device. This template is located in the following directory:

<*install_path*>\micosystem\utilities\templates\Uart_Echo

# Support for C/C++ "printf" in RTL Simulation

Each UART instance makes itself available as a file device to the LatticeMico file services. When the UART instance is used as a Standard Output and Error console (C/C++ stdout and stderr), all C/C++ "printfs" are redirected to the UART instance and are viewable on the console connected to the UART; for example, a UART instance connected over a serial link to PC's serial port, with the PC hosting a serial terminal program such as Hyperlink or TeraTerm.

**Viewing C/C++ “printfs” During RTL Simulation** It might be desirable for the software programmer to be able to view the C/C++ "printfs" even during RTL simulation. Although the aforementioned console is not available during RTL simulation, the UART component has an alternate mechanism that allows you to view C/C++ "printfs" in the RTL simulator.

**To view characters transmitted by the UART instance:**

- Enable the "Print Transmit Character" dialog box option in the UART GUI.

  This option enables behavioral code within the UART component that will print characters transmitted by the UART instance on its SOUT port during RTL simulation. This behavioral code is invisible to synthesis tools and does not impact the actual design. Therefore the designer does not have to maintain two sets of RTL code, one for simulation and the other for synthesis.

**Speeding up RTL Simulation** The UART transmits each character serially over the SOUT port, which might take hundreds of LatticeMico clock cycles. One way to speed up RTL simulation is to turn off the SOUT logic.

**To speed up RTL simulation:**

1. Enable the "Emulate Transmit Operation" dialog box option in the UART GUI.
2. Define the SIMULATION Verilog macro during RTL simulation. For example, use the `+define+SIMULATION` option while compiling the UART RTL for simulation.

Since the disabling of SOUT logic is predicated on defining the SIMULATION Verilog macro, it is acceptable to keep the "Emulate Transmit Operation" option enabled during synthesis. This means that the designer does not have to maintain two sets of RTL code, one for simulation and the other for synthesis.

**Disabling “Interrupt” to Further Reduce RTL Simulation Time** To reduce RTL simulation time further, the designer can disable the “interrupt” feature of the UART. This feature raises an interrupt whenever a character has been successfully transmitted over the SOUT port.

**To disable the UART “interrupt” feature, take one of the following actions:**

- Disable the "Use Interrupts" dialog box option in the UART GUI. The drawback of this option is that the designer will now need to maintain two sets of RTL code, one for simulation in which interrupts are disabled and another for synthesis in which interrupts are enabled.
- Disable the interrupts by defining the _MICOUART_NO_INTERRUPTS_ preprocessor macro while compiling the C/C++ software application. This macro will disable all software code that monitors and uses the UART's interrupt mechanism to transmit characters over the SOUT port. The advantage of this approach is that the designer need not maintain two sets of RTL code. The designer can leave the "Use Interrupts" option enabled for synthesis as well as simulation. The drawback of this approach is that the designer will have to recompile the C/C++ software application without the _MICOUART_NO_INTERRUPTS_ preprocessor macro whenever the software is tested on the actual FPGA. But given that the software compilation time is invariably much smaller than the time required to synthesis a new FPGA bitstream for the design, this option might be a better match for the software developer than the previous option in which interrupts are disabled in RTL.

# LatticeMico8 Microcontroller Software Support

This section describes the LatticeMico8 microcontroller software support provided for the LatticeMico UART component.

# Device Driver

The UART device driver interacts directly with the UART instance. This section describes the limitations, type definitions, structure, and functions of the UART device driver.

## Limitations

The UART device driver assumes that the UART component is used as a basic serial transmission/reception device devoid of hardware flow control.

- Software control: any desired software control must be provided by the application.
- Although the UART component has modem functionality, this feature is not supported in the LatticeMico8 UART component driver.

## Type Definitions

This section describes the type definitions for the UART device context structure. This structure, shown in Figure 9, contains the UART component instance-specific information and is dynamically generated in the DDStructs.h header file. This information is largely filled in by the managed build process by extracting the UART component-specific information from the platform specification file. As part of the managed build process, designers can choose to control the size of the generated structure, and hence the software executable, by selectively enabling some of the elements in this structure via C preprocessor macro definitions. These C preprocessor macro definitions are explained later in this document. You should not manipulate the members directly, because this structure is for exclusive use by the device driver.

Table 25 describes the parameters of the UART device context structure shown in Figure 9. The table also identifies any C preprocessor 'macro definition' that controls the the generation of the parameter. If the 'state' associated with a C preprocessor 'macro definition' is 'ifdef', then it means that the application must be compiled with this macro definition for the parameter to be generated. If the 'state' associated with a C preprocessor 'macro definition' is 'ifndef', then it means that the application must be compiled without the this macro definition for the parameter to be generated.

## C Preprocessor Macro Definitions

This section describes the C preprocessor macro definitions that are available to the software developer. There are two types of macro definitions: 'object-like' and 'function-like'.

The 'object-like' macro definitions do not take any arguments and are used to control the size of the generated application executable. There are three ways an 'object-like' macro definition can be used by the software developer.

1. Manually adding the -D<*macro name*> option to the compiler's command-line in the application's 'Build Properties'. Refer to the *LatticeMico8 Software Developers User Guide* for more information on how to manually add the macro definition in the the application's 'Build Properties' GUI.

**Figure 9: UART Device Context Structure**

```
typedef struct st_MicoUartCtx_t {
const char *name;
size_t base;
#ifndef __MICO_NO_INTERRUPTS__
#ifdef __MICOUART_INTERRUPT__
unsigned char intrLevel;
unsigned char intrAvail;
unsigned char rxBufferSize;
unsigned char txBufferSize;
#endif
#endif

#ifdef __MICOUART_BLOCKING__
unsigned char blockingTx;
unsigned char blockingRx;
#endif

#ifndef __MICO_NO_INTERRUPTS__
#ifdef __MICOUART_INTERRUPT__
unsigned int fifoenable;
unsigned char *rxBuffer;
unsigned char *txBuffer;
unsigned char rxWriteLoc;
unsigned char rxReadLoc;
unsigned char txWriteLoc;
unsigned char rxReadLoc;
volatile unsigned char txDataBytes;
volatile unsigned char rxDataBytes;
#endif
#endif
} MicoUartCtx_t;
```

**Table 25: UART Device Context Structure Parameters**

| Parameter | Data Type | C Preprocessor Macro Definition Name | C Preprocessor Macro Definition State | Description |
|---|---|---|---|---|
| name | const char * | | | Instance-specific component name (entered in MSB) |
| base | size_t | __MICO_NO_INTERRUPTS__ | ifndef | MSB-assigned base address for this instance |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| intrLevel | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Processor interrupt line to which this instance is connected |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| intrAvail | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Indicates whether the UART is to be used in interrupt mode (1) or not (0), as configured in MSB |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| rxBufferSize | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Size in bytes of the receive buffer for interrupt-driven operation, as configured in MSB |
| | | __MICOUART_INTERRUPTS__ | ifdef | |

**Table 25: UART Device Context Structure Parameters (Continued)**

| Parameter | Data Type | C Preprocessor Macro Definition Name | C Preprocessor Macro Definition State | Description |
|---|---|---|---|---|
| txBufferSize | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Size in bytes of the transmit buffer for interrupt-driven operation, as configured in MSB |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| blockingTx | unsigned char | __MICOUART_BLOCKING__ | ifndef | Indicates whether the device driver must operate in blocking mode (1) or in non-blocking mode (0) for transmission |
| blockingRx | unsigned char | __MICOUART_BLOCKING__ | ifdef | Indicates whether the device driver must operate in blocking mode (1) or in non-blocking mode (0) for reception |
| fifoenable | unsigned int | __MICO_NO_INTERRUPTS__ | ifndef | Indicates whether the UART has a TX/RX FIFO (1) or not (0). |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| rxBuffer[4] | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Internal use for receive buffer implementation. The size of the array is configurable on a per-instance basis through MSB. |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| txBuffer[4] | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Internal use for transmit buffer implementation. The size of the array is configurable on a per-instance basis through MSB. |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| rxWriteLoc | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Used internally for recexive buffer management |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| rxReadLoc | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Used internally for receive buffer management |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| txWriteLoc | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Used internally for transmit buffer management |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| txReadLoc | unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Used internally for transmit buffer management |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| txDataBytes | volatile unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Used internally for transmit buffer management |
| | | __MICOUART_INTERRUPTS__ | ifdef | |
| rxDataBytes | volatile unsigned char | __MICO_NO_INTERRUPTS__ | ifndef | Used internally for receive buffer management |
| | | __MICOUART_INTERRUPTS__ | ifdef | |

2. Automatically adding the -D<*macro name*> option to the compiler's command-line in the application's 'Build Properties' by enabling the 'check-box' associated with the macro definition. Refer to the *LatticeMico8 Software Developers User Guide* for more information on how to set up

the check/uncheck the macro definitions in the application's 'Build Properties' GUI.

3. Manually adding the macro definition to the C code using the following syntax:

```
#define <macro name>
```

It is recommended that the developer use options 1 or 2.

- __MICOUART_INTERRUPTS__

  This preprocessor macro definition enables code and data structures within the device driver that allow the UART to be used in an interrupt-driven mode. It is not defined by default.

- __MICOUART_BLOCKING__

  This preprocessor macro definition enables code and data structures within the device driver that allow the UART to be used in a blocking mode. It is not defined by default.

- __MICOUART_MISC__

  This preprocessor macro definition enables the function that modifies the baudrade of the UART on-the-fly. It is not defined by default.

The 'function-like' macro definitions are used in the LatticeMico8 software drivers to access the component's Register Map in order to perform certain operations. All 'fuction-like' macro definitions take input parameters that are used in performing the operations encoded within the macro. Table 26 describes the 'function-like' macros available in the LatticeMico8 UART driver header file 'MicoUART.h'. Table 26 also shows how each macro can be used by the software developer in his application code.

**Table 26: C Preprocessor Function-like Macros Available to the Software Developer**

| Macro Name | Second Argument to Macro | Description |
|---|---|---|
| MICO_UART_RD_RBR | The 8-bit value read from the RBR register. | This macro reads a character from the Receive Buffer. |
| MICO_UART_WR_THR | The 8-bit value to be written to the THR register. | This macro writes a character to the Transmit Buffer. |
| MICO_UART_RD_IER | The 8-bit value read from the IER register | This macro reads from the Interrupt Enable register. |
| MICO_UART_WR_IER | The 8-bit value to be written to the IER register | This macro writes to the Interrupt Enable register. |
| MICO_UART_RD_IIR | The 8-bit value read from the IIR register | This macro reads from the Interrupt Identification register. |

**Table 26: C Preprocessor Function-like Macros Available to the Software Developer (Continued)**

| Macro Name | Second Argument to Macro | Description |
|---|---|---|
| MICO_UART_WR_IIR | The 8-bit value to be written to the IIR register | This macro writes to the Interrupt Identification register. |
| MICO_UART_RD_LCR | The 8-bit value read from the LCR register | This macro reads from the Line Control register. |
| MICO_UART_WR_LCR | The 8-bit value to be written to the LCR register | This macro writes to the Line Control register. |
| MICO_UART_RD_LSR | The 8-bit value read from the LSR register. | This macro reads from the Line Status register. |
| MICO_UART_RD_MCR | The 8-bit value read from the MCR register | This macro writes to the Modem Control register. |
| MICO_UART_WR_MCR | The 8-bit value to be written to the MCR register | This macro writes to the Modem Control register. |
| MICO_UART_RD_DIV | The 16-bit value read from the DIV register | This macro reads from the Baudrate Divisor register. |
| MICO_UART_WR_DIV | The 16-bit value to be written to the DIV register | This macro reads from the Baudrate Divisor register. |

## Functions

This section describes the implemented device-driver-specific functions.

### MicoUartInit Function

```
void MicoUartInit(MicoUartCtx *ctx);
```

This function initializes a LatticeMico UART instance on the basis of the passed UART context structure. This initialization function is responsible for initializing the interrupts or buffer parameters for interrupt-driven operation. As a part of the managed build process, the LatticeDDInit function calls this initialization routine for each UART instance that is present in the platform.

Table 27 describes the parameter in the MicoUartInit function syntax.

**Table 27: MicoUartInit Function Parameter**

| Parameter | Description |
|---|---|
| MicoUartCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance. |

### MicoUart_putC Function

```
void MicoUart_putC(MicoUartCtx_t *ctx, char aChar);
```

This function posts a character to the UART THR for transmission. The behavior depends on the mode of operation:

- Interrupt-driven blocking mode – This function attempts to add the incoming byte of data to the software-implemented transmit buffer. If the buffer is full, the function waits until space is available in the buffer. When space is available, the incoming byte of data is posted to the software-implemented transmit buffer, and the function returns.
- Non-interrupt-driven blocking mode – The function directly polls the UART status register and waits for the UART transmit-hold register to become empty. Once empty, the function queues the character and returns. If the transmit-hold-register is empty, the function loads the character for transmission and returns.
- Interrupt-driven non-blocking mode – The function queries the software-implemented transmit buffer. If there is space in the buffer, it queues the character and returns.
- Non-interrupt driven non-blocking mode – The function queries the UART line status register and checks the THRE bit to see if the THR is empty. If the THR is empty, it is loaded with the incoming data byte, and the function returns. If the UART status register indicates that the transmit hold register is not empty, the function returns immediately with loading the data byte in the THR register.

Table 28 describes the parameters in the MicoUart_putC function syntax.

**Table 28: MicoUart_putC Function Parameters**

| Parameter | Description |
|---|---|
| MicoUserCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance |
| unsigned char | Character to queue for transmission |

### MicoUart_getC Function

```
unsigned int MicoUart_getC(MicoUartCtx_t *ctx, unsigned char
*pucChar);
```

This function attempts to retrieve a received character from the UART. The exact behavior depends on the operating mode:

- Interrupt-driven blocking mode – The function returns immediately with a received character if there is one waiting in the receive buffer. If there is no character pending in the receive buffer, the function continuously polls the receive buffer for a new character. The function returns when a character is placed into the software-implemented receive buffer.
- Non-interrupt-driven blocking mode – The function returns immediately with the character in the receive-holding register if the UART status indicates there is a character pending. If the UART status indicates there is no character to read, the function continuously polls the UART status

register for a new character. The function returns as soon as the RBR is no longer empty.

- Interrupt-driven non-blocking mode – The function returns immediately if there is a new character waiting in the receive buffer. If there is no character waiting in the receive buffer, the function returns immediately.
- Non-interrupt-driven non-blocking mode – The function returns immediately with the character in the UART receive-hold register if the UART status indicates there is a character pending. If there is no character pending, the function returns without reading tje receive-hold register.

Table 29 describes the parameters in the MicoUart_getC.

**Table 29: MicoUart_getC Function Parameters**

| Parameter | Description |
|---|---|
| MicoUartCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance |
| unsigned char* | Pointer to unsigned character location where the read character should be stored |

### MicoUart_setRate Function

```
unsigned int MicoUart_setRate(MicoUartCtx_t *ctx, unsigned int
baudrate);
```

This function sets the baud rate of a UART instance. The software does not check for the validity of the baud-rate value. The function computes the UART divisor value on the basis of the following formula:

divisor_value = divisor = cpu_frequency/baud_rate;

Cpu_frequency is the frequency specified in hertz.

This result is then loaded into the divisor register. It also programs the desired baud rate in the UART control register.

Table 30 describes the parameters in the MicoUart_setRate function syntax.

**Table 30: MicoUart_setRate Function Parameters**

| Parameter | Description | Notes |
|---|---|---|
| MicoUartCtx_t* | Pointer to a valid MicoUartCtx_t structure representing a valid UART instance. | |
| unsigned int | Baud rate | No software check is performed. Depending on the CPU clock speed, the baud rate might result in loss of communication or excessive errors. Consult the component data sheet for more information. |

# Software Usage Example

This section provides an example of using the UART as a standard I/O device at run time. The example is shown in Figure 10 and assumes the presence of a UART component named "uart.”

**Figure 10: UART used as a standart I/O device**

```
#include "DDStructs.h"
#include "MicoUtils.h"
#include "MicoUART.h"

static unsigned char GetCharacter(MicoUartCtx_t *pUart)
{
  unsigned char c;
  MicoUart_getC(pUart, &c);
  return(c);
}

static void SendCharacter(MicoUartCtx_t *pUart, unsigned char c)
{
  MicoUart_putC(pUart, c);
  return;
}

int main (void)
{
   /*
    * Fetch the UART context for UART named "uart" from DDStructs.h
    */
   MicoUartCtx_t *uart = &uart_core_uart;

   /*
    * Set UART baudrate to 115200
    */
   MicoUart_setRate (uart, 115200);

   /*
    * Echo 255 characters
    */
   unsigned char i = 0, c;
   do {
      c = GetCharacter (uart);

      /*
       * Wait for some time since the UART baud rate is much slower
       * than the CPU clock speed
       */
      MicoSleepMilliSecs (100);

      SendCharacter (uart, c);
   } while (++i < 255);
}
```

# Support for Viewing Characters Transmitted by UART in RTL Simulation

It might be desirable for the software developer to be able to view the transmitted characters during RTL simulation. Normally the transmitted characters would travel over the serial link to PC's serial port and be viewable in serial terminal programs such as Hyperlink or TeraTerm. Although such

consoles are not available during RTL simulation, the UART component has an alternate mechanism that allows the software developer to view transmitted characters in the RTL simulator. To view the characters transmitted, enable the "Print Transmit Character" dialog box option in the UART GUI. This option enables behavioral code within the UART component that will print characters transmitted by the UART instance on its SOUT port during RTL simulation. This behavioral code is invisible to synthesis tools and does not impact the actual design. Therefore the designer does not have to maintain two sets of RTL code, one for simulation and the other for synthesis.

## Speeding up RTL Simulation

The UART transmits each character serially over the SOUT port, which might take hundreds of LatticeMico clock cycles. One way to speed up RTL simulation is to turn off the SOUT logic.

*To speed up RTL simulation:*

1. Enable the "Emulate Transmit Operation" dialog box option in the UART GUI.
2. Define the SIMULATION Verilog macro during RTL simulation. For example, use the +define+SIMULATION option while compiling the UART RTL for simulation.

Since the disabling of SOUT logic is predicated on defining the SIMULATION Verilog macro, it is acceptable to keep the "Emulate Transmit Operation" option enabled during synthesis. This means that the designer does not have to maintain two sets of RTL code, one for simulation and the other for synthesis.

## Disable "Interrupt" to further reduce RTL simulation time

To reduce RTL simulation time further, the designer can disable the “interrupt” feature of the UART. This feature raises an interrupt whenever a character has been successfully transmitted over the SOUT port.

*To disable the UART “interrupt” feature, take one of the following actions:*

1. Disable the "Use Interrupts" dialog box option in the UART GUI. The drawback of this option is that the designer will now need to maintain two sets of RTL code, one for simulation in which interrupts are disabled and another for synthesis in which interrupts are enabled.
2. Disable the interrupts by not defining __MICOUART_INTERRUPT__ preprocessor macro while compiling the C/C++ software application. This macro will disable all software code that monitors and uses the UART's interrupt mechanism to transmit characters over the SOUT port. While the advantage of this approach is that the designer need not maintain two sets of RTL code, the designer will have to recompile the software application with the __MICOUART_INTERRUPT__ preprocessor macro

whenever the software is tested on the actual FPGA. But given that the software compilation time is invariably much smaller than the time required to synthesis a new FPGA bitstream for the design, this option might be a better match for the software developer than the previous option in which interrupts are disabled in RTL.

## Revision History

| Component Version | Description |
|---|---|
| 1.0 | Initial release. |
| 3.0 (7.0 SP2) | Used CPU clock for the MSR update. |
| 3.1 | Modified baud-rate generation.<br>Updated RX and TX path of the UART to a faster clock.<br>Implemented six-word-deep FIFO when the FIFO option is selected. |
| 3.2 | FIFO support was added for the UART driver. |
| 3.3 (8.1 SP1) | Sideband signal options added so that UART generates active-low sideband signals indicating full RBR and empty THR. |
| 3.4 | Transmit option added for printing transmit character to the simulator console window. |
| 3.5 (8.1 SP1) | WISHBONE data bus size changed from 32 bits to 8 bits. WISHBONE address bus size changed from 32 bits to 4 bits. Register map compressed to make all registers byte-addressable and eliminate unused space. |
| 3.6 | Software support added for LatticeMico8. |
| 3.7 | Removes issue that did not utilize FIFO beyond the first entry. |
| 3.7 | Updated document with new corporate logo. |
| 3.8 | Removed unused signals.<br>Component can be used in designs that do not include a processor. |

.

## Trademarks