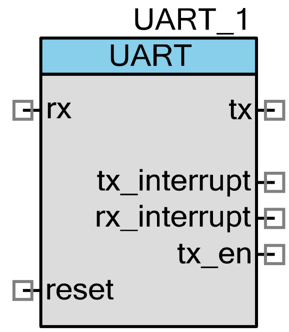
Universal Asynchronous Receiver Transmitter (UART)

2.0

Features



- 9-bit address mode with hardware address detection
- BAUD rates from 110 – 921600 bps or arbitrary up to 3 Mbps
- RX and TX buffers = 1 – 65535
- Detection of Framing, Parity and Overrun errors
- Full Duplex, Half Duplex, TX only and RX only optimized hardware
- 2 out of 3 voting per bit
- Break signal generation and detection
- 8x or 16x oversampling

General Description

The UART provides asynchronous communications commonly referred to as RS-232 or RS-485. The UART component can be configured for Full Duplex, Half Duplex, RX only or TX only versions. All versions provide the same basic functionality differing only in the amount of resources utilized.

To assist with processing of the UART receive and transmit data, independent size configurable buffers are provided. The independent circular receive and transit buffers in SRAM as well as hardware FIFOs help to ensure that data will not be missed while allowing the CPU to spend more time on critical real time tasks rather than servicing the UART.

For most use cases the UART can be easily configured by choosing the BAUD rate, parity, number of data bits and number of start bits. The most common configuration for RS-232 is often listed as "8N1" which is shorthand for 8 data bits, No parity and 1 stop bit which is also the default for the UART component. Therefore in most applications only the BAUD rate must be set. A second common use for UARTs is in multi-drop RS-485 networks. The UART component supports 9-bit addressing mode with hardware address detect, as well as a TX output enable signal to enable the TX transceiver during transmissions.

The long history of UARTs has resulted in many physical layer and protocol layer variations over time including but not limited to RS-423, DMX512, MIDI, LIN bus, legacy terminal protocols, and IrDa. To support the UART variations commonly used, the component provides configuration support for the number of data bits, stop bits, parity, hardware flow control and parity generation and detection.

As a hardware-compiled option, you can select to output a clock and serial data stream that outputs only the UART data bits on the clock's rising edge. An independent clock and data output is provided for both the TX and RX. The purpose of these outputs is to allow automatic calculation of the data CRC by connecting a CRC component to the UART.

When to Use a UART

The UART should be used any time that a compatible asynchronous communications interface is required especially RS-232 and RS-485 and other variations. The UART can also be used to create more advanced asynchronous based protocols such as DMX512, LIN and IrDa or customer/industry proprietary.

A UART should not be used in those cases where a specific component has already been created to address the protocol. For example if a DMX512, LIN or IrDa component were provided, it would have a specific implementation providing both hardware and protocol layer functionality and the UART should not be used in this case (subject to component availability).

Input/Output Connections

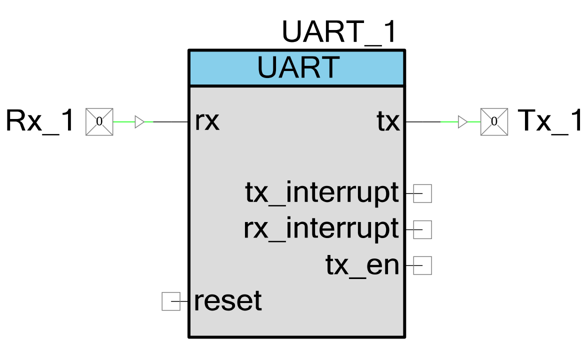
This section describes the various input and output connections for the UART. Some I/Os may be hidden on the symbol under the conditions listed in the description of that I/O.

Input	May Be Hidden	Description
clock	Y	The clock input defines the baud rate (bit-rate) of the serial communication. The baud-rate is 1/8th or 1/16th the input clock frequency depending on **Oversampling Rate** parameter. This input is visible if the **Clock Selection** parameter is set to "External." If the internal clock is selected then you define the desired baud-rate during configuration and the necessary clock frequency is solved by PSoC Creator.
reset	N	Resets the UART state machines (RX and TX) to the idle state. This will throw out any data that was currently being transmitted or received. This input is a synchronous, reset requiring at least one rising edge of the clock.
rx	Y	The rx input carries the input serial data from another device on the serial bus. This input is visible and must be connected if the **Mode** parameter is set to "RX Only", "Half Duplex" or "Full UART (RX & TX)."
cts_n	Y	The cts_n input indicates that another device is ready to receive data. This input is an active-low input indicated by the _n, and indicates when the other device has room for more data to be transmitted to it. This input is visible if the **Flow Control** parameter is set to "Hardware".

Output	May Be Hidden	Description
tx	Y	The tx output carries the output serial data to another device on the serial bus. This output is visible if the **Mode** parameter is set to "TX Only", “Half Duplex” or "Full UART (RX & TX)."
rts_n	Y	The rts output indicates to another device that you are ready to receive data. This output is active-low indicated by the _n, and informs another device when you have room for more data to be received. This output is visible if the **Flow Control** parameter is set to "Hardware."
tx_en	Y	The tx_en output is used primarily for RS-485 communication to indicate that you are transmitting on the bus. This output will go high before a transmit starts and low when transmit is complete indicating a busy bus to the rest of the devices on the bus. This output is visible when the **Hardware TX Enable** parameter is true.
tx_interrupt	Y	The tx_interrupt output is the logical OR of the group of possible interrupt sources. This signal will go high while any of the enabled interrupt sources are true. This output is visible if the **Mode** parameter is set to "TX Only" or "Full UART (RX & TX)."
rx_interrupt	Y	The rx_interrupt output is the logical OR of the group of possible interrupt sources. This signal will go high while any of the enabled interrupt sources are true. This output is visible if the **Mode** parameter is set to "RX Only", “Half Duplex” or "Full UART (RX & TX)."
tx_data	Y	The tx_data output used to shift out the TX data to a CRC component or other logic. This output is visible when the **Enable CRC outputs** parameter is true.
tx_clk	Y	The tx_clk output provides clock edge used to shift out the TX data to a CRC component or other logic. This output is visible when the **Enable CRC outputs** parameter is true.
rx_data	Y	The tx_data output used to shift out the RX data to a CRC component or other logic. This output is visible when the **Enable CRC outputs** parameter is true.
rx_clk	Y	The rx_clk output provides clock edge used to shift out the RX data to a CRC component or other logic. This output is visible when the **Enable CRC outputs** parameter is true.

Schematic Macro Information

The default UART in the Component Catalog is a schematic macro using a UART component with default settings. It is connected to digital input and output Pins components.



Parameters and Setup

Drag an UART component onto your design and double-click it to open the Configure dialog.

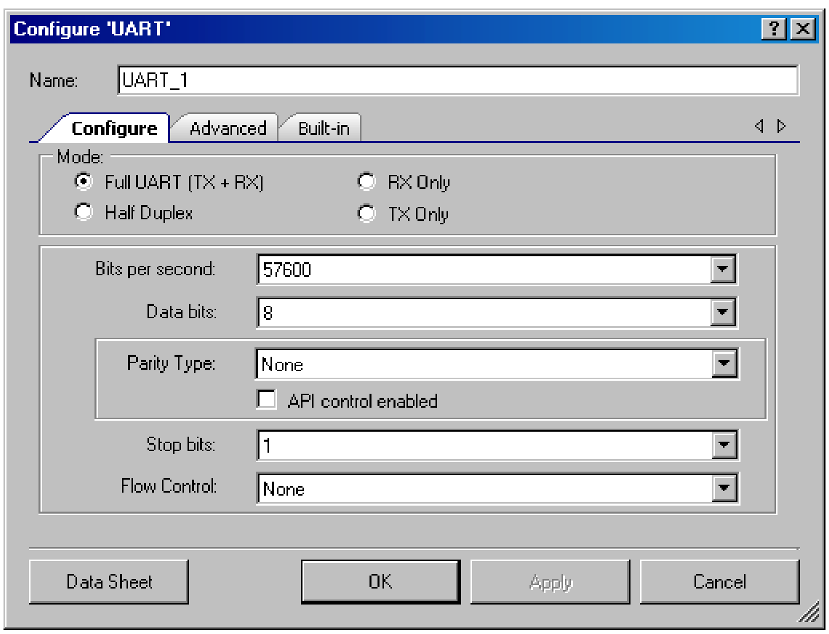
Hardware vs. Software Options

Hardware configuration options change the way the project is synthesized and placed in the hardware. You must rebuild the hardware if you make changes to any of these options. Software configuration options do not affect synthesis or placement. When setting these parameters before build time you are setting their initial value which may be modified at any time with the API provided.

The following sections describe the UART parameters, and how they are configured using the dialog. They also indicate whether the options are hardware or software.

Configure Tab

The dialog is set up to look like a hyperterminal configuration window to avoid incorrect configuration of two sides of the bus, where the PC using hyperterminal is quite often the other side of the bus.



All of these options are hardware configuration options.

Mode

This parameter defines the desired functional components to include in the UART. This can be setup to be a bidirectional Full UART (TX + RX) (default), Half Duplex UART (uses half the resources), RS-232 Receiver (RX Only) or Transmitter (TX Only).

Bits per second

This parameter defines the baud-rate or bit width configuration of the hardware for clock generation. The default is 57600.

If the internal clock is used (set by the **Clock Selection** parameter) the necessary clock to achieve this baud-rate will be generated.

Data bits

This parameter defines the number of data bits transmitted between start and stop of a single UART transaction. Options are 5, 6, 7, 8 (default), or 9.

- 8 data bits is the default configuration sending a byte per transfer.
- 9-bit mode does not transmit 9 data bits; the 9^{th} bit takes the place of the parity bit as an indicator of address using Mark/Space parity. Mark/Space parity should be selected if 9 data bits mode used.

Parity Type

This parameter defines the functionality of the parity bit location in the transfer. This can be set to None (default), Odd, Even or Mark/Space. If you selected 9 data bits, then select Mark/Space as the **Parity Type**.

API control enabled

This check box is used to change parity by using the control register and the UART_WriteControlRegister() function. The parity type can be dynamically changed between bytes without disrupting UART operation if this option selected, but the component will use more resources.

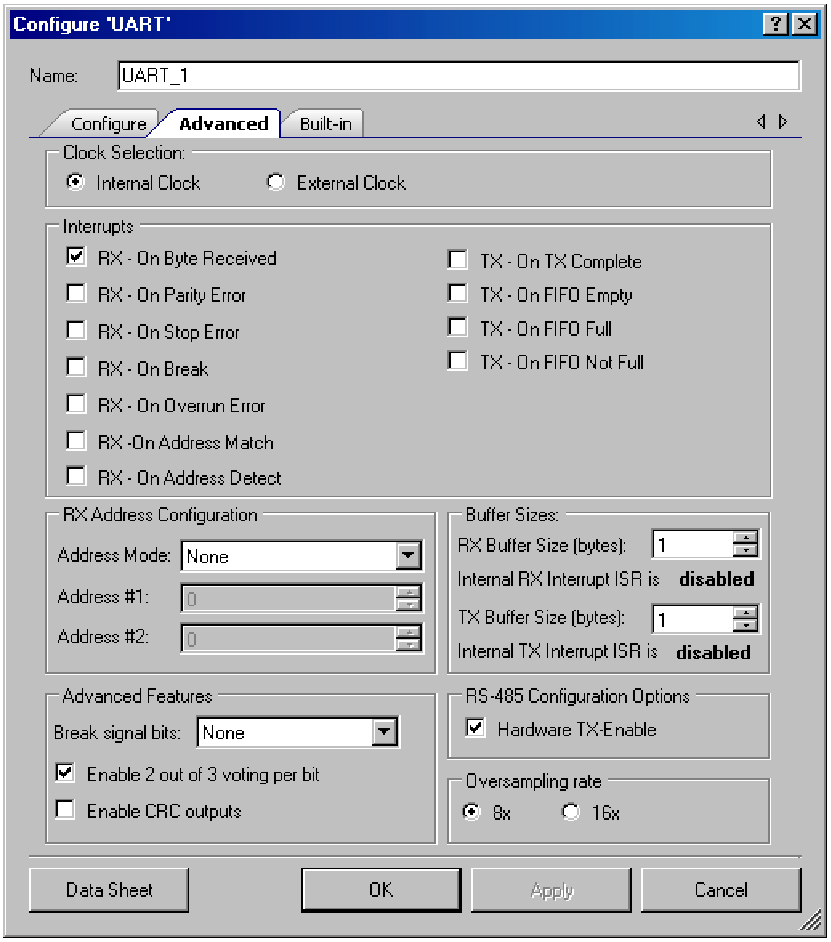
Stop bits

This parameter defines the number of stop bits implemented in the transmitter. This parameter can be set to 1 (default) or 2 data bits.

Flow Control

This parameter allows you to choose between Hardware or None (default). When this parameter is set to Hardware, the CTS and RTS signals become available on the symbol.

Advanced Tab



Hardware Configuration Options

Clock Selection

This parameter allows you to choose between an internally configured clock or an externally configured clock or I/O for the baud-rate generation. When set to the "Internal" setting the required clock frequency is calculated and configured by PSoC Creator. In the "External" mode the component does not control the baud-rate but can calculate the expected baud-rate.

If this parameter is "Internal" then the clock input is not visible on the symbol.

Address Mode

This parameter defines how hardware and software interact to handle device addresses and data bytes. This parameter can be set to the following types:

- Software Byte-by-Byte – Hardware indicates the detection of an address byte for every byte received. Software must read the byte and determine if this address matches the device addresses defined as in the Address #1 or Address #2 parameters
- Software Detect to Buffer – Hardware indicates the detection of an address byte and software will copy all data into the RX buffer defined by the RX Buffer Size parameter.
- Hardware Byte-By-Byte – Hardware detects a byte and forces an interrupt to move all data from the hardware FIFO into the data buffer defined by RX Buffer Size.
- Hardware Detect to buffer – Hardware detects a byte and forces an interrupt to move only the data (address byte is not included) from the hardware FIFO into the data buffer defined by RX Buffer Size.
- None – No RX address detection is implemented.

Hardware TX Enable

This parameter enables or disables the use of the TX-Enable output of the TX UART. This signal is used in RS-485 communications. The hardware provides the functionality of this output automatically, based on buffer conditions.

Advanced Features

- Break signal bits – Break signal bits parameter enables Break signal generation and detection and defines the number of logic 0s bits transmitted. This option will save resources when set to None.
- Enable 2 out of 3 voting per bit – The Enable 2 out of 3 voting per bit enables or disables error compensation algorithm. This option will save resources when disabled. For more information, refer to the Functional Description section of this data sheet.
- Enable CRC outputs – The Enable CRC outputs parameter enables or disables tx_data, tx_clk, rx_data, rx_clk outputs. They used to output a clock and serial data stream that outputs only the UART data bits on the clock's rising edge. The purpose of these outputs is to allow automatic calculation of the data CRC. This option will save resources when disabled.

Oversampling Rate

This parameter allows you to choose clock divider for the baud-rate generation.

Software Configuration Options

Interrupts

The "Interrupt On" parameters allow you configure the interrupt sources. These values are ORed with any of the other "Interrupt On" parameter to give a final group of events that can trigger an interrupt. The software can re-configure these modes at any time; these parameters simply define an initial configuration.

- RX – On Byte Received (bool)
- RX – On Parity Error (bool)
- RX – On Stop Error (bool)
- RX – On Break (bool)
- RX – On Overrun Error (bool)
- RX – On Address Match (bool)
- RX – On Address Detect (bool)
- TX – On TX Complete (bool)
- TX – On FIFO Empty (bool)
- TX – On FIFO Full (bool)
- TX – On FIFO Not Full (bool)

You may handle the ISR with an external interrupt component connected to the tx_interrupt or rx_interrupt output. The interrupt output pin is visible depending on the selected **Mode** parameter. It outputs the same signal to the internal interrupt based on the selected status interrupts.

These outputs may then be used as a DMA request source to the DMA from the RX or TX buffer independent of the interrupt, or as another interrupt dependant upon the desired functionality.

RX Address #1/#2

The RX Address parameters indicate up to two device addresses that the UART may assume. These parameters are stored in hardware for hardware address detection modes described in the RX Address Mode parameter and are available to firmware the software address modes.

RX Buffer Size (bytes)

This parameter defines how many bytes of RAM to allocate for an RX buffer. Data is moved from the receive registers into this buffer.

Four bytes of hardware FIFO are used as a buffer when the buffer size selected is less than or equal to 4 bytes. Buffer sizes greater than 4 bytes require the use of interrupts to handle moving of the data from the receive FIFO into this buffer and the UART_GetChar() or UART_ReadRXData() APIs get data from the correct source without any changes to your top-level firmware.

When the RX buffer size is greater than 4 bytes, the **Internal RX Interrupt ISR** is automatically enabled and the **RX – On Byte Received** interrupt source is selected and disabled for use because it causes incorrect handler functionality.

TX Buffer Size (bytes)

This parameter defines how many bytes of RAM to allocate for the TX buffer. Data is written into this buffer with the UART_PutChar() and UART_PutArray() API commands.

Four bytes of hardware FIFO are used as a buffer when the buffer size selected less than or equal to 4 bytes; otherwise, the RAM buffer is allocated. Buffer sizes greater than 4 bytes require the use of interrupts to handle moving of the data from the transmit buffer into the hardware FIFO without any changes to your top level firmware.

When the TX buffer size is greater than 4 bytes, the **Internal TX Interrupt ISR** is automatically enabled and the **TX – On FIFO EMPTY** interrupt source is selected and disabled for use because it causes incorrect handler functionality.

The TX interrupt is not available in Half duplex mode; therefore, the TX Buffer Size is limited to up to 4 bytes when Half duplex mode is selected.

Internal RX Interrupt ISR

Enables the ISR supplied by the component for the RX portion of the UART. This parameter is set automatically depending on the **RX Buffer Size** parameter, because the internal ISR is needed to handle transferring data from the FIFO to the RX buffer.

Internal TX Interrupt ISR

Enables the ISR supplied by the component for the TX portion of the UART. This parameter is set automatically depending on the **TX Buffer Size** parameter, because the internal ISR is needed to handle transferring data to the FIFO from the TX buffer.

Clock Selection

When the internal clock configuration is selected PSoC Creator will calculate the needed frequency and clock source and will generate the resource needed for implementation. Otherwise, you must supply the clock and calculate the baud-rate at 1/8th or 1/16th the input clock frequency.

The clock tolerance should be maximum ±2%. The warning will be generated if clock could not be generated within this limit. In this case the Master Clock should be modified in the DWR.

Placement

The UART component is placed throughout the UDB array and all placement information is provided to the API through the *cyfitter.h* file.

Resources

Resources	Resource Type				API Memory (Bytes)		Pins (per External I/O)
	Datapath Cells	Macrocells	Status Cells	Control/Count7 Cells	Flash	RAM	
Full UART	3	45	2	2	1976	241	13
Full UART*	2	45	2	3	1976	241	13
Simple UART	3	21	2	1	850	45	3
Half Duplex	1	18	1	2	860	45	3
RX Only	1	12	1	2	353	20	2
TX Only	2	9	1	1	588	30	2
TX Only*	1	9	1	2	588	30	2

* Parameter TxBitClkGenDP = false. (To switch go to Expression View of Configure tab).

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name "UART_1" to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is "UART."

Function	Description
void UART_Start(void)	Initializes and enable the UART operation.
void UART_Stop(void)	Disable the UART operation.
uint8 UART_ReadControlRegister(void)	Returns the current value of the control register.
void UART_WriteControlRegister(uint8 control)	Writes an 8-bit value into the control register.
void UART_EnableRxInt(void)	Enables the internal interrupt irq.
void UART_DisableRxInt(void)	Disables the internal interrupt irq.
void UART_SetRxInterruptMode(uint8 intSrc)	Configures the RX interrupt sources enabled.
uint8 UART_ReadRxData(void)	Returns the data in RX Data register.

Function	Description
uint8 UART_ReadRxStatus(void)	Returns the current state of the status register.
uint8 UART_GetChar(void)	Returns the next byte of received data.
uint16 UART_GetByte(void)	Reads UART RX buffer immediately, returns received character and error condition.
uint8/uint16 UART_GetRxBufferSize(void)	Determine the amount of bytes left in the RX buffer and return the count in bytes.
void UART_ClearRxBuffer(void)	Clears the memory array of all received data.
void UART_SetRxAddressMode(uint8 addressMode)	Sets the software controlled Addressing mode used by the RX portion of the UART.
void UART_SetRxAddress1(uint8 address)	Sets the first of two hardware-detectable addresses.
void UART_SetRxAddress2(uint8 address)	Sets the second of two hardware-detectable addresses.
void UART_EnableTxInt(void)	Enables the internal interrupt irq
void UART_DisableTxInt(void)	Disables the internal interrupt irq.
void UART_SetTxInterruptMode(uint8 intSrc)	Configures the TX interrupt sources enabled
void UART_WriteTxData(uint8 txDataByte)	Sends a byte without checking for buffer room or status
uint8 UART_ReadTxStatus(void)	Reads the status register for the TX portion of the UART
void UART_PutChar(uint8 txDataByte)	Puts a byte of data into the transmit buffer to be sent when the bus is available.
void UART_PutString(uint8* string)	Places data from a string into the memory buffer for transmitting.
void UART_PutArray(uint8* string, uint8/uint16 byteCount)	Places data from a memory array into the memory buffer for transmitting
void UART_PutCRLF(uint8 txDataByte)	Writes a byte of data followed by a Carriage Return and Line Feed to the transmit buffer.
uint8/uint16 UART_GetTxBufferSize(void)	Determine the amount of space left in the TX buffer and return the count in bytes.
void UART_ClearTxBuffer(void)	Clears all data from the TX buffer.
void UART_SendBreak(uint8 retMode)	Transmit a break signal on the bus.
void UART_SetTxAddressMode (uint8 addressMode)	Configures the transmitter to signal the next bytes as address or data.
void UART_LoadRxConfig(void)	Loads the receiver configuration. Half Duplex UART is ready for receive byte.
void UART_LoadTxConfig(void)	Loads the transmitter configuration. Half Duplex UART is ready

Function	Description
	for transmit byte.
void UART_Sleep(void)	Stops the UART operation and saves the user configuration.
void UART_Wakeup(void)	Restores and enables the user configuration.
void UART_SaveConfig(void)	Save the current user configuration.
void UART_RestoreConfig(void)	Restores the user configuration.
void UART_Init(void)	Initializes default configuration provided with customizer.
void UART_Enable(void)	Enables the UART block operation.

Global Variables

Variable	Description
UART_initVar	Indicates whether the UART has been initialized. The variable is initialized to 0 and set to 1 the first time UART_Start() is called. This allows the component to restart without reinitialization after the first call to the UART_Start() routine. It is required for correct operation of the component that the UART is initialized before Send or Put commands are run. Therefore, all APIs that write transmit data must check that the component has been initialized using this variable. If reinitialization of the component is required, then the UART_Init() function can be called before the UART_Start() or UART_Enable() function.
UART_rxBuffer	This is a RAM allocated RX buffer with a user-defined length. This buffer used by interrupts, when RX buffer size parameter selected more then 4, to store received data and by UART_ReadRxData() and UART_GetChar() APIs to convey data to the user level firmware.
UART_rxBufferWrite	This variable is used by the RX interrupt as a cyclic index for UART_rxBuffer to write data. This variable also used by the UART_ReadRxData() and UART_GetChar() APIs to identify new data. Cleared to zero by the UART_ClearRxBuffer() API.
UART_rxBufferRead	This variable is used by the UART_ReadRxData() and UART_GetChar() APIs as a cyclic index for UART_rxBuffer to read data. Cleared to zero by the UART_ClearRxBuffer() API.
UART_rxBufferLoopDetect	This variable is set to one in RX interrupt when UART_rxBufferWrite index overtakes UART_rxBufferRead index. This is pre-overload condition which will affect on UART_rxBufferOverflow when next byte received, or will be set to zero when the UART_ReadRxData() or UART_GetChar() API called. Cleared to zero by the UART_ClearRxBuffer() API.
UART_rxBufferOverflow	This variable is used to indicate overload condition. It set to one in RX interrupt when there isn't free space in UART_rxBufferRead to write new data. This condition returned and cleared to zero by the UART_ReadRxStatus() API as an UART_RX_STS_SOFT_BUFF_OVER bit along with RX Status register bits.

Variable	Description
	Cleared to zero by the UART_ClearRxBuffer() API.
UART_txBuffer	This is a RAM allocated TX buffer with the user defined length. This buffer used by sending APIs when TX buffer size parameter selected more then 4, to store data for transmitting and by TX interrupt to move data into hardware FIFO.
UART_txBufferWrite	This variable is used by the UART_WriteTxData (), UART_PutChar(), UART_PutString(), UART_PutArray(), and UART_PutCRLF() APIs as a cyclic index for UART_txBuffer to write data. This variable is also used by the TX interrupt to identify new data for transmitting. Cleared to zero by the UART_ClearTxBuffer() API.
UART_txBufferRead	This variable is used by the TX interrupt as a cyclic index for the UART_txBuffer to read data. Cleared to zero by the UART_ClearRxBuffer() API.

void UART_Start(void)

Description: This is the preferred method to begin component operation. UART_Start() sets the initVar variable, calls the UART_Init() function, and then calls the UART_Enable() function.

Parameters: void

Return Value: void

Side Effects: If the initVar variable is already set, this function only calls the UART_Enable() function.

void UART_Stop(void)

Description: Disables the UART operation.

Parameters: void

Return Value: void

Side Effects: None

uint8 UART_ReadControlRegister(void)

Description: Returns the current value of the control register.

Parameters: void

Return Value: (uint8) Contents of the control register The following defines can be used to interpret the returned value. Additional information on the control register can be found in the control register description near the end of this document.

Value	Description
UART_HD_SEND	Configures whether the half duplex UART (if enabled) is in RX mode (0), or in TX mode (1).
UART_HD_SEND_BREAK	Set to send a break signal on the bus. This bit is written by the UART_SendBreak() function.
UART_CTRL_MARK	Configures whether the parity bit during the next transaction (in Mark/Space parity mode) will be a 1 or 0.
UART_CTRL_PARITYTYPE_MASK	2 bit wide field configuring the parity for the next transfer if software configurable. The following defines can be used to set the parity type:
UART__B_UART__NONE_REVB	No parity
UART__B_UART__EVEN_REVB	Even parity
UART__B_UART__ODD_REVB	Odd parity
UART__B_UART__MARK_SPACE_REVB	Mark/Space parity
UART_CTRL_RXADDR_MODE_MASK	3 bit wide field configuring the expected hardware addressing operation for the UART receiver. The following defines can be used to set the address mode:
UART__B_UART__AM_SW_BYTE_BYTE	Software Byte-by-Byte address detection
UART__B_UART__AM_SW_DETECT_TO_BUFFER	Software Detect to Buffer address detection
UART__B_UART__AM_HW_BYTE_BY_BYTE	Hardware Byte-by-Byte address detection

UART__B_UART__AM_HW_DETECT_TO_BUFFER	Hardware Detect to Buffer address detection
UART__B_UART__AM_NONE	No address detection

Side Effects:

void UART_WriteControlRegister(uint8 control)

Description: Writes an 8-bit value into the control register

Parameters: (uint8) control: Control Register Value

Value	Description
UART_HD_SEND	Configures whether the half duplex UART (if enabled) is in RX mode (0), or in TX mode (1). Can be set and cleared using the UART_LoadTxConfig() and UART_LoadRxConfig() functions.
UART_HD_SEND_BREAK	Set to send a break signal on the bus. This bit is best written using the UART_SendBreak() function.
UART_CTRL_MARK	Configures whether the parity bit during the next transaction (in Mark/Space parity mode) will be a 1 or 0.
UART_CTRL_PARITYTYPE_MASK	2 bit wide field configuring the parity for the next transfer if software configurable. The following defines can be used to set the parity type:
UART__B_UART__NONE_REVB	No parity
UART__B_UART__EVEN_REVB	Even parity
UART__B_UART__ODD_REVB	Odd parity
UART__B_UART__MARK_SPACE_REVB	Mark/Space parity

UART_CTRL_RXADDR_MODE_MASK	3 bit wide field configuring the expected hardware addressing operation for the UART receiver. The following defines can be used to set the address mode:
UART__B_UART__AM_SW_BYTE_BYTE	Software Byte-by-Byte address detection
UART__B_UART__AM_SW_DETECT_TO_BUFFER	Software Detect to Buffer address detection
UART__B_UART__AM_HW_BYTE_BY_BYTE	Hardware Byte-by-Byte address detection
UART__B_UART__AM_HW_DETECT_TO_BUFFER	Hardware Detect to Buffer address detection
UART__B_UART__AM_NONE	No address detection

Return Value: void

Side Effects:

void UART_EnableRxInt(void)

Description: Enables the internal receiver interrupt

Parameters: void

Return Value: void

Side Effects: Only available if the RX internal interrupt implementation is selected in the UART

void UART_DisableRxInt(void)

Description: Disables the internal receiver interrupt

Parameters: void

Return Value: void

Side Effects: Only available if the RX internal interrupt implementation is selected in the UART

void UART_SetRxInterruptMode(uint8 intSrc)

Description: Configures the RX interrupt sources enabled

Parameters: (uint8) intSrc: Bit-Field containing the RX interrupts to enable. Based on the bit-field arrangement of the status register. This value must be a combination of status register bit-masks shown below:

Value	Description
UART_RX_STS_FIFO_NOTEMPTY	Interrupt on byte received.
UART_RX_STS_PAR_ERROR	Interrupt on parity error.
UART_RX_STS_STOP_ERROR	Interrupt on stop error.
UART_RX_STS_BREAK	Interrupt on break.
UART_RX_STS_OVERRUN	Interrupt on overrun error.
UART_RX_STS_ADDR_MATCH	Interrupt on address match.
UART_RX_STS_MRKSPC	Interrupt on address detect.

Return Value: void

Side Effects:

uint8 UART_ReadRxData(void)

Description: Returns the next byte of received data. This function returns data without checking the status, it is up to the user to separately check the status.

Parameters: void

Return Value: (uint8) Received data from RX register

Side Effects:

uint8 UART_ReadRxStatus(void)

Description: Returns the current state of the receiver status register and the software buffer overflow status.

Parameters: void

Return Value: (uint8) Current RX status register value

Value	Description
UART_RX_STS_MRKSPC	Indicates whether a mark or a space was received in the parity bit (if Mark/Space parity selected).
UART_RX_STS_BREAK	If set, indicates a break was detected.
UART_RX_STS_PAR_ERROR	If set, indicates a parity error was detected.
UART_RX_STS_STOP_ERROR	If set, indicates a framing error was detected.
UART_RX_STS_OVERRUN	If set, indicates the FIFO buffer was overrun.
UART_RX_STS_FIFO_NOTEMPTY	If set Indicates the FIFO has data available.
UART_RX_STS_ADDR_MATCH	If set Indicates the received address matches the configured Rx address.
UART_RX_STS_SOFT_BUFF_OVER	If set, indicates the Rx buffer was overrun.

Side Effects: All status register bits are clear on read except UART_RX_STS_FIFO_NOTEMPTY. UART_RX_STS_FIFO_NOTEMPTY clears immediately after RX data register read. See the Registers section later in this data sheet.

uint8 UART_GetChar(void)

Description: Returns the last received byte of data. GetChar is designed for ASCII characters and returns a unit8 where 1 to 255 are values for valid characters and 0 indicates an error occurred or no data is present.

Parameters: void

Return Value: (uint8) Character read from UART RX buffer. ASCII character values from 1 to 255 are valid. A returned zero signifies an error condition or no data available.

Side Effects:

uint16 UART_GetByte(void)

Description: Reads UART RX buffer immediately, returns received character and error condition.

Parameters: void

Return Value: (uint16) MSB contains status and LSB contains UART RX data. If the MSB is nonzero, an error has occurred.

Side Effects:

uint8/uint16 UART_GetRxBufferSize(void)

Description: Returns the number of bytes left in the RX buffer.

Parameters: void

Return Value: (uint8/uint16) Integer count of the number of bytes left in the RX buffer. Type depends on RX Buffer Size parameter.

Side Effects:

void UART_ClearRxBuffer(void)

Description: Clears the receiver memory buffer of all received data.

Parameters: void

Return Value: void

Side Effects:

void UART_SetRxAddressMode(uint8 addressMode)

Description: Sets the software controlled Addressing mode used by the RX portion of the UART

Parameters: (uint8) addressMode: Enumerated value indicating the mode of RX addressing to implement.

Value	Description
UART__B_UART__AM_SW_BYTE_BYTE	Software Byte-by-Byte address detection
UART__B_UART__AM_SW_DETECT_TO_BUFFER	Software Detect to Buffer address detection
UART__B_UART__AM_HW_BYTE_BY_BYTE	Hardware Byte-by-Byte address detection
UART__B_UART__AM_HW_DETECT_TO_BUFFER	Hardware Detect to Buffer address detection
UART__B_UART__AM_NONE	No address detection

Return Value: void

Side Effects:

void UART_SetRxAddress1(uint8 address)

Description: Sets the first of two hardware detectable receiver addresses

Parameters: (uint8) address: Address #1 for hardware address detection

Return Value: void

Side Effects:

void UART_SetRxAddress2(uint8 address)

Description: Sets the second of two hardware detectable receiver addresses

Parameters: (uint8) address: Address #2 for hardware address detection

Return Value: void

Side Effects:

void UART_EnableTxInt(void)

Description: Enables the internal transmitter interrupt

Parameters: void

Return Value: void

Side Effects: Only available if the TX internal interrupt implementation is selected in the UART configuration

void UART_DisableTxInt(void)

Description: Disables the internal transmitter interrupt

Parameters: void

Return Value: void

Side Effects: Only available if the TX internal interrupt implementation is selected in the UART configuration

void UART_SetTxInterruptMode(uint8 intSrc)

Description: Configures the TX interrupt sources to be enabled (but does not enable the interrupt)

Parameters: (uint8) intSrc: Bit-Field containing the TX interrupt sources to enable.

Value	Description
UART_TX_STS_COMPLETE	Interrupt on TX byte complete.
UART_TX_STS_FIFO_EMPTY	Interrupt when TX FIFO is empty.
UART_TX_STS_FIFO_FULL	Interrupt when TX FIFO is full.
UART_TX_STS_NOT_FULL	Interrupt when TX FIFO is not full.

Return Value: void

Side Effects:

void UART_WriteTxData(uint8 txDataByte)

Description: Places a byte of data into the transmit buffer to be sent when the bus is available without checking the TX status register. It is up to the user to separately check status.

Parameters: (uint8) txDataByte: data byte

Return Value: void

Side Effects:

uint8 UART_ReadTxStatus(void)

Description: Reads the status register for the TX portion of the UART

Parameters: void

Return Value: (uint8) Contents of the TX Status register

Value	Description
UART_TX_STS_COMPLETE	If set, indicates byte was transmitted successfully.
UART_TX_STS_FIFO_EMPTY	If set, indicates the TX FIFO is empty.
UART_TX_STS_FIFO_FULL	If set, indicates the TX FIFO is full.
UART_TX_STS_NOT_FULL	If set, indicates the FIFO is not full.

Side Effects: This function reads the TX status register, which is cleared on read.

void UART_PutChar(uint8 txDataByte)

Description: Puts a byte of data into the transmit buffer to be sent when the bus is available. This is a blocking API that waits until the TX buffer has room to hold the data.

Parameters: (uint8) txDataByte: Byte containing the data to transmit.

Return Value: void

Side Effects:

void UART_PutString(uint8* string)

Description: Sends a NULL terminated string to the TX buffer for transmission.

Parameters: (uint8*) string: Pointer to the null terminated string array residing in RAM or ROM

Return Value: void

Side Effects: If there is not enough memory in the TX buffer for the entire string, this function will block until the last character of the string is loaded into the TX buffer.

void UART_PutArray(uint8* string, uint8/uint16 byteCount)

Description: Places N bytes of data from a memory array into the TX buffer for transmission.

Parameters: (uint8*) string: Address of the memory array residing in RAM or ROM
(uint8/uint16) byteCount: Number of bytes to be transmitted. Type depends on TX Buffer Size parameter.

Return Value: void

Side Effects: If there is not enough memory in the TX buffer for the entire array, this function will block until the last byte of the array is loaded into the TX buffer.

void UART_PutCRLF(uint8 txDataByte)

Description: Writes a byte of data followed by a carriage return (0x0D) and line feed (0x0A) to the transmit buffer.

Parameters: (uint8) txDataByte: Data byte to transmit before the carriage return and line feed

Return Value: void

Side Effects: If there is not enough memory in the TX buffer for all three bytes, this function will block until the last of the three bytes are loaded into the TX buffer.

uint8/uint16 UART_GetTxBufferSize(void)

Description: Determines the amount of available space left in the TX buffer and return the count in bytes.

Parameters: void

Return Value: (uint8/uint16) Buffer size in bytes. Type depends on TX Buffer Size parameter.

Side Effects:

void UART_ClearTxBuffer(void)

Description: Clears all data from the TX buffer

Parameters: void

Return Value: void

Side Effects: Data waiting in the transmit buffer will not be sent; a byte that is currently transmitting will finish transmitting.

void UART_SendBreak(uint8 retMode)

Description: Transmit a break signal on the bus

Parameters: (uint8) retMode: Send Break return mode. See table below for options.

Options	Description
UART_SEND_BREAK	Initialize registers for Break, sends the Break signal and return immediately.
UART_WAIT_FOR_COMLETE_REINIT	Wait until Break transmission is complete, reinitialize registers to normal transmission mode then return.
UART_REINIT	Reinitialize registers to normal transmission mode then return.
UART_SEND_WAIT_REINIT	Performs both options: UART_SEND_BREAK and UART_WAIT_FOR_COMLETE_REINIT. It is recommended to use this option for most cases.

Return Value: void

Side Effects: The SendBreak function initializes registers to send break signal. Break signal length depends on the Break signal bits configuration. The register configuration should be reinitialized before normal 8-bit communication can continue.

void UART_SetTxAddressMode(uint8 addressMode)

Description: Configures the transmitter to signal the next bytes is address or data.

Parameters: (uint8) addressMode:

Options	Description
UART_SET_SPACE	Configure the transmitter to send the next byte as a data.
UART_SET_MARK	Configure the transmitter to send the next byte as an address.

Return Value: void

Side Effects: This function sets and clears UART_CTRL_MARK bit in Control register.

void UART_LoadRxConfig(void)

Description: Loads the receiver configuration in half duplex mode. After calling this function, the UART is ready to receive data.

Parameters: void

Return Value: void

Side Effects: Valid only in half duplex mode.It is the user's responsibility to ensure that the previous transaction is complete and it is safe to unload the transmitter configuration.

void UART_LoadTxConfig(void)

Description: Loads the transmitter configuration in half duplex mode. After calling this function, the UART is ready to transmit data.

Parameters: void

Return Value: void

Side Effects: Valid only in half duplex mode.It is the user's responsibility to ensure that the previous transaction is complete and it is safe to unload the receiver configuration.

void UART_Sleep(void)

Description: This is the preferred API to prepare the component for sleep. The UART_Sleep() API saves the current component state. Then it calls the UART_Stop() function and calls UART_SaveConfig() to save the hardware configuration.

Call the UART_Sleep() function before calling the CyPmSleep() or the CyPmHibernate() function. Refer to the PSoC Creator *System Reference Guide* for more information about power management functions.

Parameters: void

Return Value: void

Side Effects:

void UART_Wakeup(void)

Description: This is the preferred API to restore the component to the state when UART_Sleep() was called. The UART_Wakeup() function calls the UART_RestoreConfig() function to restore the configuration. If the component was enabled before the UART_Sleep() function was called, the UART_Wakeup() function will also re-enable the component.

Parameters: void

Return Value: void

Side Effects: This function clears the RX and TX software buffers, but it will not clear data from the FIFOs and will not reset any hardware state machines. Calling the UART_Wakeup() function without first calling the UART_Sleep() or UART_SaveConfig() function may produce unexpected behavior.

void UART_SaveConfig(void)

Description: This function saves the component configuration. This will save non-retention registers. This function will also save the current component parameter values, as defined in the Configure dialog or as modified by appropriate APIs. This function is called by the UART_Sleep() function.

Parameters: None

Return Value: None

Side Effects: All non-retention registers except FIFO saved to RAM.

void UART_RestoreConfig(void)

Description: Restores the user configuration of non-retention registers.

Parameters: None

Return Value: None

Side Effects: All non-retention registers except FIFO loaded from RAM. This function should be called only after UART_SaveConfig() is called otherwise incorrect data will be loaded into the registers.

void UART_Init(void)

Description: Initializes or restores the component according to the customizer Configure dialog settings. It is not necessary to call UART_Init() because the UART_Start() API calls this function and is the preferred method to begin component operation.

Parameters: None

Return Value: None

Side Effects: All registers will be set to values according to the customizer Configure dialog.

void UART_Enable(void)

Description: Activates the hardware and begins component operation. It is not necessary to call UART_Enable() because the UART_Start() API calls this function, which is the preferred method to begin component operation.

Parameters: None

Return Value: None

Side Effects: None

Defines

The following defines are provided only for reference. The define values are determined by the component customizer settings.

Define	Description
UART_INIT_RX_INTERRUPTS_MASK	Defines the initial configuration of the interrupt sources chosen by the user in the configuration GUI. This is a mask of the bits in the status register that have been enabled at configuration as sources for the RX interrupt.
UART_INIT_TX_INTERRUPTS_MASK	Defines the initial configuration of the interrupt sources chosen by the user in the configuration GUI. This is a mask of the bits in the status register that have been enabled at configuration as sources for the TX interrupt.
UART_TXBUFFERSIZE	Defines the amount of memory to allocate for the TX memory array buffer. This does not include the 4 bytes included in the FIFO.
UART_RXBUFFERSIZE	Defines the amount of memory to allocate for the RX memory array buffer. This does not include the 4 bytes included in the FIFO.
UART_NUMBER_OF_DATA_BITS	Defines the number of bits per data transfer which is used to calculate the Bit Clock Generator and Bit-Counter configuration registers.
UART_BIT_CENTER	Based on the number of data bits this value is used to calculate the center point for the RX Bit-Clock Generator which is loaded into the configuration register at startup of the UART.
UART_RXHWADDRESS1	Defines the initial address selected in the configuration GUI. This address is loaded into the corresponding hardware register at startup of the UART.
UART_RXHWADDRESS2	Defines the initial address selected in the configuration GUI. This address is loaded into the corresponding hardware register at startup of the UART.

Sample Firmware Source Code

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the "Find Example Project" topic in the PSoC Creator Help for more information.

Functional Description

The UART component provides synchronous communication commonly referred to as RS232 or RS485. The UART can be configured for Full Duplex, Half Duplex, RX only or TX only operation. The following sections give an overview in how to use the UART component.

Default Configuration

The default configuration for the UART is as an 8-bit UART with no Flow control and None Parity, running at a baud-rate of 57.6 Kbps

UART Mode: Full UART (RX+TX)

This mode implements a full duplex UART consisting of an asynchronous Receiver and Transmitter. A single clock is needed in this mode to define the baud-rate for both the receiver and transmitter.

UART Mode: Half Duplex

This mode implements a full UART, but uses half as many resources as the full UART configuration. In this configuration, the UART can be configured to switch between RX mode and TX mode, but cannot perform RX and TX operations simultaneously. The RX or TX configuration can be loaded by calling the UART_LoadRxConfig() or UART_LoadTxConfig() APIs.

In this mode, the **TX – On FIFO Not Full** status is not available, but the **TX – On FIFO Full** status can be used instead**.** As TX interrupts are not available in this mode, the TX buffer size is limited to 4 bytes.

Half Duplex mode example:

- This example assumes the component has been placed in a design with the name "UART_1."
- Configure UART to Mode: Half Duplex, Bits per seconds: 115200, Data bits: 8, Parity Type: None, Rx Buffer Size:1, Tx Buffer Size:1.

```
#include <device.h>

void main()
{

    uint8 recByte;
    uint8 tmpStat;

    CYGlobalIntEnable;                    /* Enable interrupts */

    UART_1_Start();                    /* Start UART */
    UART_1_LoadTxConfig();                /* Configure UART for transmitting */
    UART_1_PutString("Half Duplex Test");   /* Send message */
    /* make sure that data has been transmitted */
```

```
    CyDelay(30);    /* Appropriate delay could be used */
                    /* Alternatively, check TX_STS_COMPLETE status bit */
    UART_1_LoadRxConfig(); /* Configure UART for receiving */
    while(1)
    {
        recByte = UART_1_GetChar();     /* Check for receive byte */
        if(recByte > 0)                 /* If byte received */
        {
            UART_1_LoadTxConfig();      /* Configure UART for transmitting */
            UART_1_PutChar(recByte);    /* Send received byte back */
            do                          /* wait till transmission complete */
            {   /* Read Status register */
                tmpStat = UART_1_ReadTxStatus();
                /* Check the TX_STS_COMPLETE status bit */
            }while(~tmpStat & UART_1_TX_STS_COMPLETE);
            UART_1_LoadRxConfig();      /* Configure UART for receiving */
        }
    }
}
```

UART Mode: RX Only

This mode implements only the receiver portion of the UART. A single clock is needed in this mode to define the baud-rate for the receiver.

UART Mode: TX Only

This mode implements only the transmitter portion of the UART. A single clock is needed in this mode to define the baud-rate for the transmitter.

UART Flow Control: None, Hardware

Flow control on the UART provides separate RX and TX status indication lines to the existing bus. When hardware flow control is enabled, a ‘Request to Send’ (RTS) line and a ‘Clear to Send’ (CTS) line are available between this UART and another UART. The RTS line is an input to the UART that is set by the other UART in the system when it is OK to send data on the bus. The CTS line is an output of the UART informing the other UART on the bus that it is ready to receive data. The RTS line of one UART is connected to the CTS line of the other UART and vice versa. These lines are only valid before a transmission is started. If the signal is set or cleared after a transfer is started the change will only affect the next transfer.

UART Parity: None

In this mode, there is no parity bit. The data flow is "Start, Data, Stop."

UART Parity: Odd

Odd parity begins with the parity bit equal to 1. Each time a 1 is encountered in the data stream, the parity bit is toggled. At the end of the data transmission the state of the parity bit is transmitted. Odd parity ensures that there is always a transition on the UART bus. If all data is zero then the parity bit sent will equal 1. The data flow is "Start, Data, Parity, Stop". Odd parity is the most common parity type used.

UART Parity: Even

Even parity begins with the parity bit equal to 0. Each time a 1 is encountered in the data stream, the parity bit is toggled. At the end of the data transmission the state of the parity bit is transmitted. The data flow is "Start, Data, Parity, Stop."

UART Parity: Mark/Space, Data bits: 9

Mark/Space parity is most typically used to define whether the data sent was an address or standard data. A mark (1) in the parity bit indicates data was sent and a space (0) in the parity bit indicates an address was sent. The mark or space is sent in the parity bit position in the data transmission. The data flow is "Start, Data, Parity, Stop" similar to the other parity modes but this bit is set by software before the transfer rather than being calculated based on the data bit values. This parity is available for RS-485 and similar protocols.

TX Usage model

Firmware should use the UART_SetTxAddressMode API with the UART_SET_MARK parameter to configure the transmitter for the first address byte in the packet. This API sets the UART_CTRL_MARK bit in the control register. After setting the MARK parity, the first byte sent is an address and the remaining bytes are sent as data with SPACE parity. The transmitter will automatically send data bytes after the first address byte. Before sending another packet, the UART_CTRL_MARK bit in control register should be cleared for at least for one clock. This can be done by calling the UART_SetTxAddressMode API with the UART_SET_SPACE parameter. This is shown in the code example below.

Send addressed packet example:

- This example assumes the component has been placed in a design with the name "UART_TX."
- Configure UART to Data bits: 9, Parity Type: Mark/Space.

```
#include <device.h>

void main()
{
    UART_TX_Start();
    /*Set UART_CTRL_MARK bit in Control register*/
    UART_TX_SetTxAddressMode(UART_TX_SET_MARK);
    /*Send data packet with the address in first byte*/
    /*The address byte is character '1', which is equal to 0x31 in hex format*/
    UART_TX_PutString("1UART TEST\r");

    /*Clear UART_CTRL_MARK bit in Control register*/
```

```
        UART_TX_SetTxAddressMode(UART_TX_SET_SPACE);
    }
```

RX Usage model

The UART_RX_STS_MRKSPC bit in the status register indicates that receiver got the address or data byte.

Receive addressed packet example:

- This example assumes the component has been placed in a design with the name "UART_RX."
- Configure UART to Data bits: 9, Parity Type: Mark/Space, Interrupts: RX - On Byte Received, Address Mode: Software Byte by Byte, Address#1: 31.
- Connect external ISR to rx_interrupt pin with the name “isr_rx”.

```
#include <device.h>

#define STR_LEN_MAX     60u
char rx_buffer[STR_LEN_MAX];
uint8 packet_receivedRX = 0u;

void main()
{
    CYGlobalIntEnable;          /* Enable interrupts */
    isr_rx_Start();
    UART_RX_Start();

    If(packet_receivedRX == 1u)
    {
        /* add analyze here */
        packet_receivedRX = 0u;
    }
}
```

Source Code Example for ISR routine

```
uint8 rec_status = 0u;
uint8 rec_data = 0;
static uint8 pointerRX = 0u;
static uint8 address_detected = 0u;

rec_status = UART_RX_RXSTATUS_REG;
if(rec_status & UART_RX_RX_STS_FIFO_NOTEMPTY)
{
    rec_data = UART_RX_RXDATA_REG;
    if(rec_status & UART_RX_RX_STS_MRKSPC)
    {
        if (rec_data == UART_RX_RXHWADDRESS1)
        {
            address_detected = 1;
        }
```

```
            else
            {
                address_detected = 0;
            }
        }
        else
        {
            if(address_detected)
            {
                if(pointerRX >= STR_LEN_MAX)
                {
                    pointerRX = 0u;
                }
                /* Detect end of packet */
                if(rec_data == '\r')
                {   /* write null terminated string */
                    rx_buffer[pointerRX++] = 0u;
                    pointerRX = 0u;
                    paket_receivedRX = 1u;
                }
                else
                {
                    rx_buffer[pointerRX++] = rec_data;
                }
            }
        }
    }
```

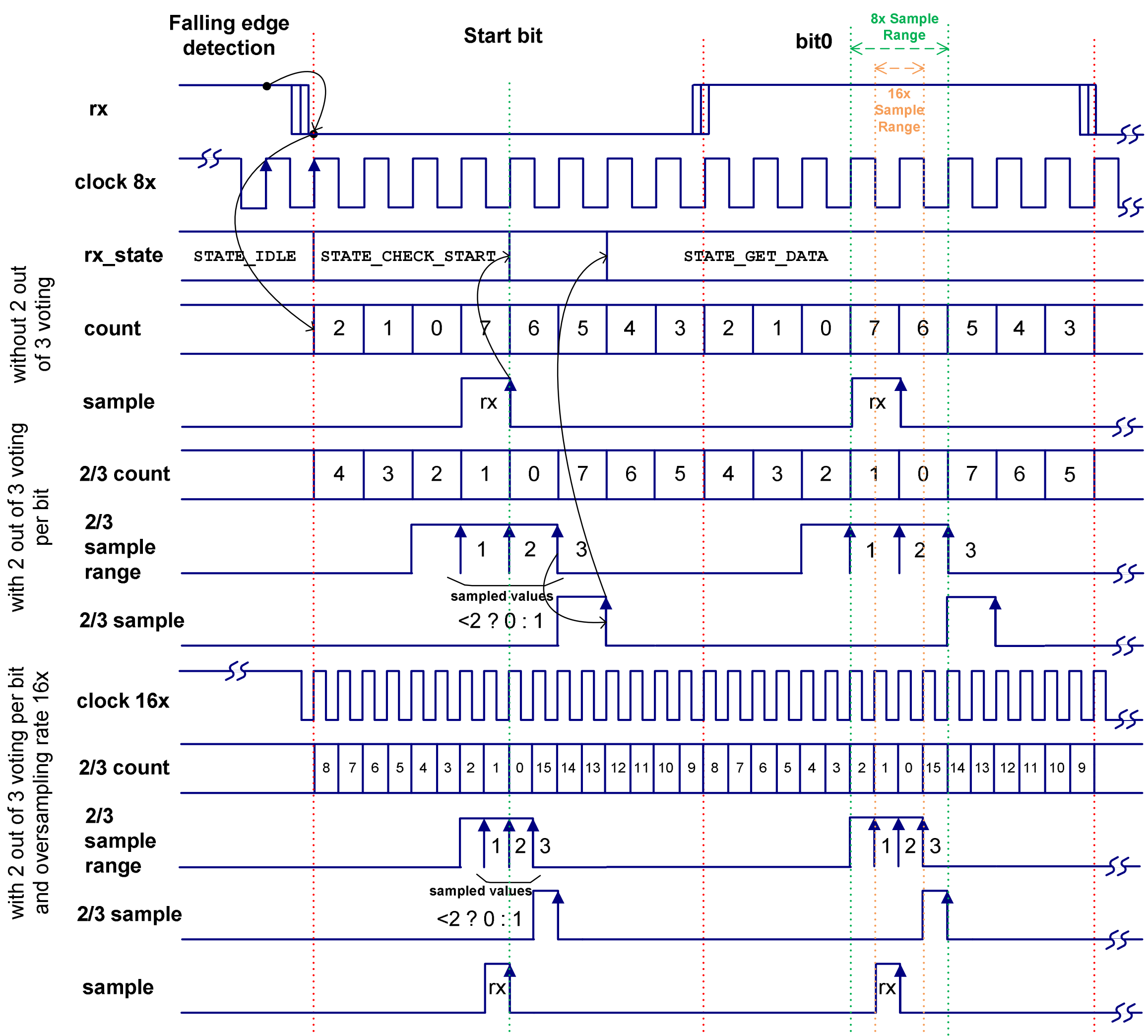
UART Stop Bits: One, Two

The number of stop bits is available as a synchronization mechanism. In slower systems it is sometimes necessary for the stop command to occupy two bit times in order to allow the receiving side to be able to process the data before more data is sent. Sending two bit-widths of the stop signal, the transmitter is allows the receiver extra time to interpret the data byte and parity. The second stop bit is not checked for a framing error by the receiver. The data flow is the same "Start, Data, [Parity], Stop," the stop bit time can be configured to either one or two bit-widths.

2 out of 3 Voting

The 2 out of 3 voting feature enables an error compensation algorithm. This algorithm essentially oversamples the middle of each bit 3 times and performs a majority vote to decide whether the bit is a 0 or a 1. If 2 out of 3 voting is not enabled, the middle of each bit is only sampled once.

When enabled, this parameter requires additional hardware resources to implement a 3-bit counter based on the RX input for three oversampling clock cycles. The following diagram shows the implementation of 8-bit and 16-bit oversampling, with and without 2 out of 3 voting.



Falling edge detection is implemented to recognize the start bit. After this detection, the counter starts down counting from the half bit length to 0, and the receiver switches to CHECK_START state. When the counter reaches 0, the RX line is sampled 3 times. If the RX line is verified to be low (e.g. at least 2 out of 3 bits were 0), the receiver goes to the GET_DATA state. Otherwise, the receiver will return to the IDLE state. The start bit detection sequence is the same for 8x or 16x oversampling rates.

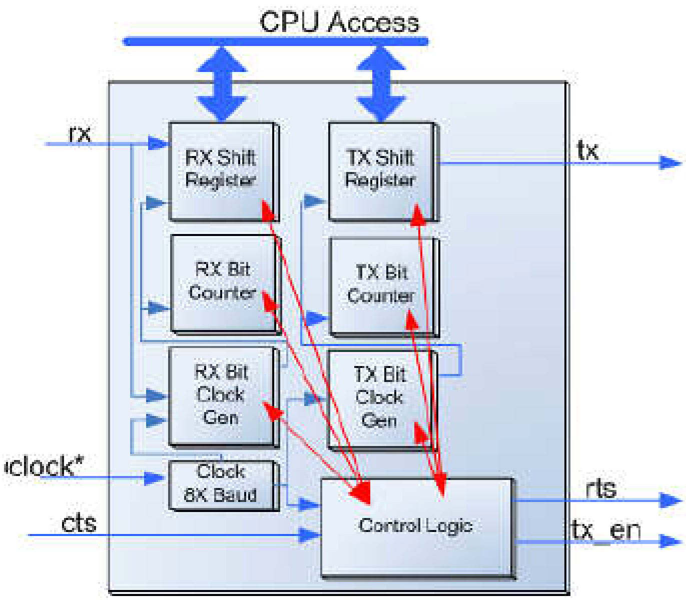
Once the receiver has entered the GET_DATA state, the RX input is fed into a counter which is enabled on counter cycles 4-6 (3 cycles). This counter will count the number of 1s seen on the RX input. If the counter value is 2 or greater, the output of this counter will be a 1 otherwise the output will be 0. This value will be sampled into the datapath as the RX value on the 7th clock edge. If voting is not enabled, the RX input is simply sampled on the 5th clock edge after the detection of the start bit, and continues every 8th positive clock edge after that.

When an oversampling rate of 16x is enabled, the voting algorithm will occur on counter cycles 8-10 and the output of the counter will be sampled by the datapath as the RX value on the 11th cycle. If voting is not enabled, the RX input is sampled on the 9th clock edge and will continue on every 16th clock edge after that.

Block Diagram and Configuration

The UART is implemented in the UDB blocks and is described in the following block diagram.

Figure 1 UDB Implementation



Registers

The APIs previously described provide support for the common runtime functions required for most applications. The following sections provide brief descriptions of the UART registers for the advanced user.

RX and TX Status

The status registers (RX and TX have independent status registers) are read-only registers that contain the various status bits defined for the UART. The value of these registers can be accessed using the UART_ReadRxStatus() and UART_ReadTxStatus() function calls.

The interrupt output signals (tx_interrupt and rx_interrupt) are generated by ORing the masked bit-fields within each register. The masks can be set using the UART_SetRxInterruptMode() and UART_SetTxInterruptMode() function calls. Upon receiving an interrupt the interrupt source can be retrieved by reading the respective status register with the UART_GetRxInterruptSource() and UART_GetTxInterruptSource() function calls. The status registers are clear on read so the interrupt source is held until one of the UART_ReadRxStatus() or UART_ReadTxStatus()

functions is called. All operations on the status register must use the following defines for the bit-fields as these bit-fields may be moved around within the status register at build time.

There are several bit-fields masks defined for the status registers. Any of these bit-fields may be included as an interrupt source. The #defines are available in the generated header file (.h).

The status data is registered at the input clock edge of the UART. Several of these bits are sticky and are cleared on a read of the status register. They are assigned as clear on read for use as an interrupt output for the UART. All other bits are configured as transparent and represent the data directly from the inputs of the status register; they are not sticky and therefore are not clear on read.

All bits configured as sticky are indicated with an asterisk (*) in the following defines:

RX Status Register

Define	Description
UART_RX_STS_MRKSPC *	Status of the mark/space parity bit. This bit indicates whether a mark or space was seen in the parity bit location of the transfer. It is only implemented if the address mode is set to use hardware addressing.
UART_RX_STS_BREAK *	Indicates that a break signal was detected in the transfer.
UART_RX_STS_PAR_ERROR *	Indicates that a parity error was detected in the transfer.
UART_RX_STS_STOP_ERROR *	This bit indicates framing error. The framing error is caused when the UART hardware sees the logic 0 where the stop bit should be (logic 1).
UART_RX_STS_OVERRUN *	Indicates that the receive FIFO buffer has been overrun.
UART_RX_STS_FIFO_NOTEMPTY	Indicates whether or not the Receive FIFO is Not empty.
UART_RX_STS_ADDR_MATCH *	Indicates that the address byte received matches one of the two addresses available for hardware address detection.

TX Status Register

Define	Description
UART_TX_STS_FIFO_FULL	Indicates that the transmit FIFO is full. This should not be confused with the transmit buffer implemented in memory as the status of that buffer is not indicated in hardware, it must be checked in firmware.
UART_TX_STS_FIFO_NOT_FULL**	Indicates that the transmit FIFO is not full.
UART_TX_STS_FIFO_EMPTY	Indicates that the transmit FIFO is empty.
UART_TX_STS_COMPLETE *	Indicates that the last byte has been transmitted from FIFO.

** - Not available in Half Duplex mode

Control

The Control register allows you to control the general operation of the UART. This register is written with the UART_WriteControlRegister() function and read with the UART_ReadControlRegister() function. The control register is not used if simple UART options are selected in the customizer, for more details see the Recourses paragraph. When reading or writing the control register you must use the bit-field definitions as defined in the header (.h) file. The #defines for the control register are as follows:

UART_HD_SEND

Used for dynamically reconfiguring between RX and TX operation in half duplex mode. This bit is set by the UART_LoadTxConfig() function and cleared by the UART_LoadRxConfig() function.

UART_HD_SEND_BREAK

When set, will send a break signal on the bus. This bit is written by the UART_SendBreak() function.

UART_CTRL_MARK

Used to control the Mark/Space parity operation of the transmit byte. When set, this bit indicates that the next byte transmitted on the bus will include a 1 (Mark) in the parity bit location. All subsequent bytes will contain a 0 (Space) in the parity bit location until this bit is cleared and re-set by firmware.

UART_CTRL_PARITYTYPE_MASK

The parity type control is a 2 bit wide field used to define the parity operation for the next transfer. This bit-field will be 2 consecutive bits in the control register and all operations on this bit-field must use the #defines associated with the parity types available. These are:

Value	Description
UART__B_UART__NONE_REVB	No parity
UART__B_UART__EVEN_REVB	Even parity
UART__B_UART__ODD_REVB	Odd parity
UART__B_UART__MARK_SPACE_REVB	Mark/Space parity

This bit-field is configured at initialization with the parity type defined in the "Parity Type" configuration parameter and may be modified during run-time using the UART_WriteControlRegister() API call.

UART_CTRL_RXADDR_MODE_MASK

The RX address mode control is a 3 bit field used to define the expected hardware addressing operation for the UART receiver. This bit field will be 3 consecutive bits in the control register and all operations on this bit-field must use the #defines associated with the compare modes available. These are:

Value	Description
UART__B_UART__AM_SW_BYTE_BYTE	Software Byte-by-Byte address detection
UART__B_UART__AM_SW_DETECT_TO_BUFFER	Software Detect to Buffer address detection
UART__B_UART__AM_HW_BYTE_BY_BYTE	Hardware Byte-by-Byte address detection
UART__B_UART__AM_HW_DETECT_TO_BUFFER	Hardware Detect to Buffer address detection
UART__B_UART__AM_NONE	No address detection

This bit field is configured at initialization with the "AddressMode" configuration parameter and may be modified during run time using the UART_WriteControlRegister() API call.

TX Data (8-bits)

The TX data register contains the data to be transmitted. This is implemented as a FIFO . There is a software state machine to control data from the transmit memory buffer to handle larger portions of data to be sent. All APIs dealing with the transmission of data must go through this register in order to place the data onto the bus. If there is data in this register and flow control indicates that data can be sent, then the data will be transmitted on the bus. As soon as this register (FIFO) is empty, no more data will be transmitted on the bus until it is added to the FIFO. DMA may be setup to fill this FIFO when empty using the TX data register address defined in the header file.

Value	Description
UART_TXDATA_REG	TX data register

RX Data

The RX data register contains the received data, implemented as a FIFO. There is a software state machine to control data movement from this receive FIFO into the memory buffer. Typically the RX interrupt will indicate that data has been received, at which time the data can be retrieved with either the CPU or DMA. DMA may be setup to retrieve data from this register whenever the FIFO is not empty using the RX data register address defined in the header file.

Value	Description
UART_RXDATA_REG	RX data register

Constants

There are several constants defined for the status and control registers as well as some enumerated types. Most of these are described above for the status and control registers. However, there are more constants needed in the header file. Each of the register definitions requires either a pointer into the register data or a register address. Due to multiple endianness` of the compilers it is required that the CY_GET_REGX and CY_SET_REGX macros are used to access registers greater than 8 bits in length. These macros require the use of the defines ending in _PTR for each of the registers.

It is also required that the control and status register bits be allowed to be placed and routed by the fitter engine during build-time. Constants are created to define the placement of the bits. For each of the status and control register bits there is an associated _SHIFT value which defines the bit's offset within the register. These are used in the header file to define the final bit mask as a _MASK definition (the _MASK extension is only added to bit-fields greater than a single bit, all single bit values drop the _MASK extension).

References

Not applicable

DC and AC Electrical Characteristics

The following values are indicative of expected performance and based on initial characterization data.

Timing Characteristics "Maximum with Nominal Routing"

Data collection is currently in progress. This table will be updated in a future release.

Parameter	Description		Min	Typ	Max	Units
f_{CLOCK}	Component clock frequency [1]					
		Full UART			16	MHz
		Simple UART			24	MHz
		Half Duplex UART			18	MHz
		RX Only			26	MHz
		TX Only			38	MHz
t_{CLOCK}	Clock period		1 / f_{CLOCK}			ns
f_b	Bit rate				f_{CLOCK} / Oversampling	Mbps
T_{CLOCK}	Clock tolerance					
		8x Oversampling		2.6		%
		16x Oversampling		3.2		%
$\%_{ERR}$	Error			TBD		%
t_{RES}	Reset pulse width		t_{CLOCK}+5			ns
t_{CTS_TX}	CTS_N Inactive to TX_EN Active and Start bit on TX		1		2	t_{CLOCK}
t_{TX_TXDATA}	Delay from TX to TX_DATA			1		t_{CLOCK}
t_{TX_TXCLK}	Delay from TX change to TX_CLK Active					
		8x Oversampling		5		t_{CLOCK}
		16x Oversampling		9		t_{CLOCK}
t_{S_RES}	Reset setup time		5			ns
t_{RTS_RX}	RTS_N Inactive to RX data			TBD		ns
t_{RX_RXCLK} t_{RX_RXINT}	Delay from RX to RX_CLK					
		8x Oversampling	4		5	t_{CLOCK}
		16x Oversampling	8		9	t_{CLOCK}
t_{RXCLK_RTS}	Delay from last RX_CLK raise to RTS_N Active			1		t_{CLOCK}
t_{RX_RXDATA}	Delay from RX to RX_DATA		0		1	t_{CLOCK}

[1] The maximum component clock frequency depends on the selected mode and additional features.

Timing Characteristics "Maximum with All Routing[1]"

Data collection is currently in progress. This table will be updated in a future release.

Parameter	Description	Min	Typ	Max	Units
f_{CLOCK}	Component clock frequency[2]				
	Full UART			8	MHz
	Simple UART			12	MHz
	Half Duplex UART			9	MHz
	RX Only			13	MHz
	TX Only			19	MHz
t_{CLOCK}	Clock period	$1 / f_{CLOCK}$			ns
f_b	Bit rate			f_{CLOCK} / Oversampling	Mbps
T_{CLOCK}	Clock tolerance				
	8x Oversampling		2.6		%
	16x Oversampling		3.2		%
$\%_{ERR}$	Error		TBD		%
t_{RES}	Reset pulse width	t_{CLOCK}+5			ns
t_{CTS_TX}	CTS_N Inactive to TX_EN Active and Start bit on TX	1		2	t_{CLOCK}
t_{TX_TXDATA}	Delay from TX to TX_DATA		1		t_{CLOCK}
t_{TX_TXCLK}	Delay from TX change to TX_CLK Active				
	8x Oversampling		5		t_{CLOCK}
	16x Oversampling		9		t_{CLOCK}
t_{S_RES}	Reset setup time	5			ns
t_{RTS_RX}	RTS_N Inactive to RX data		TBD		ns
t_{RX_RXCLK} t_{RX_RXINT}	Delay from RX to RX_CLK				
	8x Oversampling	4		5	t_{CLOCK}
	16x Oversampling	8		9	t_{CLOCK}
t_{RXCLK_RTS}	Delay from last RX_CLK raise to RTS_N Active		1		t_{CLOCK}
t_{RX_RXDATA}	Delay from RX to RX_DATA	0		1	t_{CLOCK}

1 Maximum for "All Routing" is calculated by <nominal>/2 rounded to the nearest integer. This value provides a basis for the user to not have to worry about meeting timing if they are running at or below this component frequency.

2 The maximum component clock frequency depends on the selected mode and additional features.

Full UART options:

Mode:	Full UART
Parity:	Even
API control enabled:	Enable
Flow Control:	Hardware
Address Mode:	Software Byte by Byte
RX Buffer Size (bytes)	5
TX Buffer Size (bytes)	5
Break signal bits:	13
2 out of 3 voting:	Enable
CRC outputs:	Enable
Hardware TX:	Enable
Oversampling rate:	16x
Reset:	Input pin

Simple UART options:

Mode:	Full UART
Parity:	None
API control enabled:	Disable
Flow Control:	None
Address Mode:	None
RX Buffer Size (bytes)	1
TX Buffer Size (bytes)	1
Break signal bits:	None
2 out of 3 voting:	Disable
CRC outputs:	Disable
Hardware TX:	Disable
Oversampling rate:	8x
Reset:	None

Half Duplex UART options:

Mode: Half Duplex
All other options same as the Simple UART

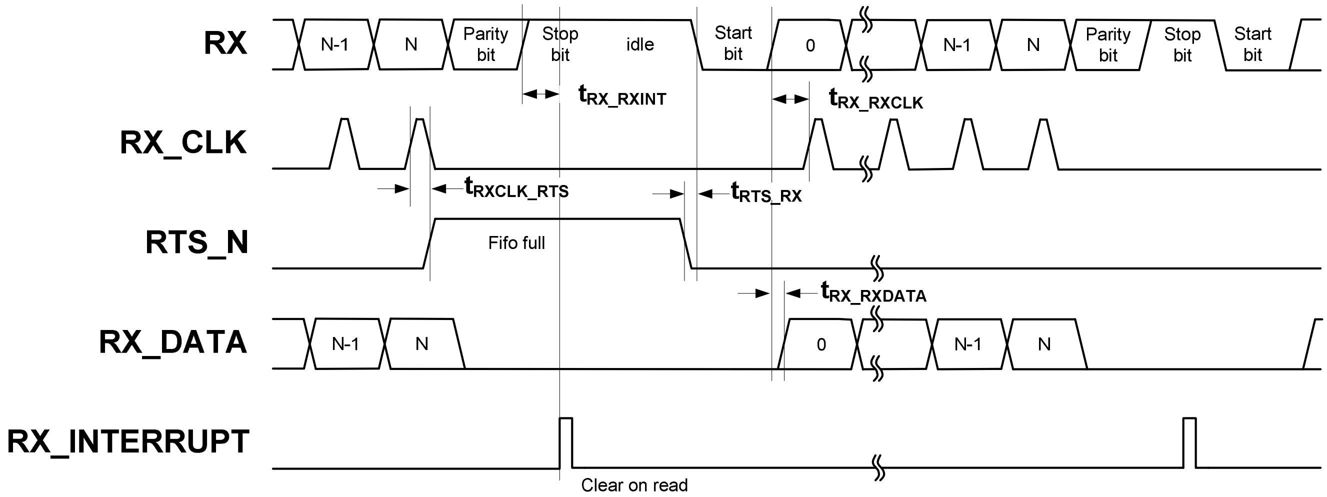
RX Only options:

Mode: RX Only
All other options same as the Simple UART

TX Only options:

Mode: TX Only
All other options same as the Simple UART

Figure 2. RX Mode Timing Diagram



How to Use STA Results for Characteristics Data

Nominal route maximums are gathered through multiple test passes with Static Timing Analysis (STA). You can calculate the maximums for your designs using the STA results with the following mechanisms:

f_{CLOCK} Maximum Component Clock Frequency appears in Timing results in the clock summary as the IntClock (if internal clock is selected) or the named external clock. See below for an example of the internal clock limitations from the *_timing.html*.

-Clock Summary

Clock	Actual Freq	Max Freq	Violation
BUS_CLK	24.000 MHz	58.046 MHz	
UART_1_IntClock	0.462 MHz	42.758 MHz	

t_{CLOCK} Calculate clock period from the following equation:

$$t_{CLOCK} = \frac{1}{f_{CLOCK}}$$

f_b Bit rate is equal to clock frequency (**f_{CLOCK}**) divided by the oversampling rate. Use oversampling rate 8x for maximum baud rate calculations, as shown in the equation below:

$$f_b = \frac{f_{CLOCK}}{Oversampling}$$

T_{CLOCK} Calculate clock tolerance using the following method:

Assume that UART is configured as 8x oversampling, 2 out of 3 voting disabled, 8 data bits, parity none, and one Stop bit. The Receiver samples the RX line at the fifth clock of every bit. A new frame is recognized by the falling edge at the beginning of the active-low Start bit. The receive UART resets its counters on this falling edge, and expects the mid Start bit to occur after 4 clock cycles, and the mid-point of each subsequent bit to appear every 8 clock cycles. If the UART clock has 0% error, then the sampling happens exactly at the mid-point of the Stop bit. But, because the UART clock will not have zero error, the sampling happens earlier or later than the mid-point on every bit. This error keeps accumulating and results in the maximum error on the Stop bit. If you sample a bit ½ bit period (8/2 = ±4 clocks) too early or too late, you will sample at the bit transition and have incorrect data. The bit transition time equals 25% of the bit time for the normal signal quality. So, the allowed error at the middle of the Stop bit will equal ±3 periods of the UART clock.

Another error to include in this budget is the synchronization error when the falling edge of the Start bit is detected. The UART starts on the next rising edge of its 8x clock after Start bit detection. Because the 8x clock and the received data stream are asynchronous, the falling edge of the Start bit could occur just after an 8x clock rising edge. This means that the UART has a ±1 clock error built in at the synchronization point. So, our error budget reduces to ±2 periods.

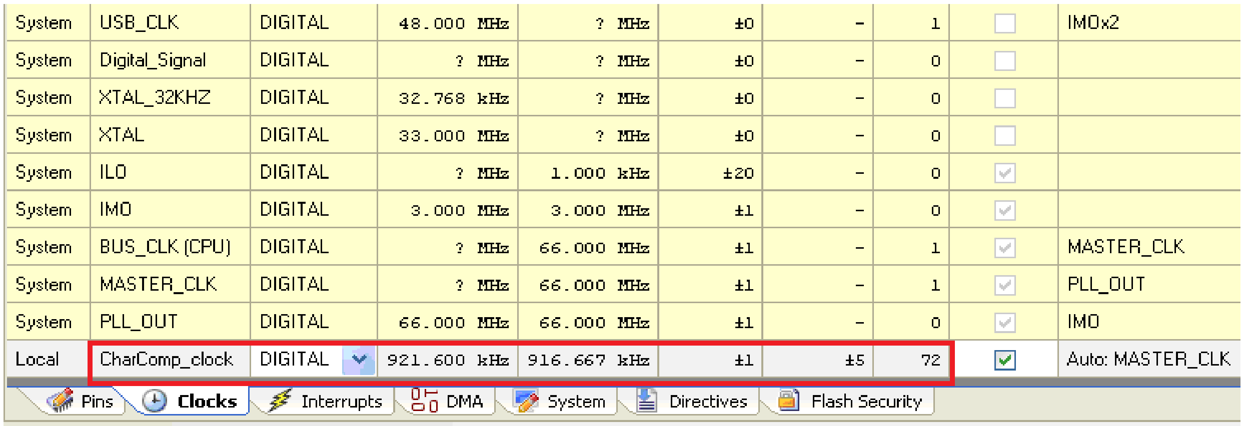
The total clock periods from the falling edge of the Start bit to the middle of the Stop bit is equal to 9.5*8=76. The total clock tolerance is ±2 / 76 * 100% = ±2.6%.

The clock tolerance for 16x oversampling is: (16/2 * (1-0.25) – 1) / (9.5*16) * 100% = ±3.2 %

This total tolerance has to be split between the receiver and transmitter in any proportion. For example, if the device on one side of the UART bus (microcontroller or PC) runs on a standard 100ppm crystal oscillator, the device on the other side can use almost all of the tolerance budget.

$\%_{ERR}$ This error is present on the system when Creator cannot generate the exact frequency clock required by the UART due to the PLL clock frequency and divider value. The difference can be seen in the design wide resources (DWR) as the desired and nominal frequency for the CharComp_clock. The error is calculated using the following equation:

$$\%_{ERR} = \frac{f_{des} - f_{nom}}{f_{des}} * 100\%$$



System	USB_CLK	DIGITAL	48.000 MHz	? MHz	±0	-	1	☐	IMOx2
System	Digital_Signal	DIGITAL	? MHz	? MHz	±0	-	0	☐	
System	XTAL_32KHZ	DIGITAL	32.768 kHz	? MHz	±0	-	0	☐	
System	XTAL	DIGITAL	33.000 MHz	? MHz	±0	-	0	☐	
System	ILO	DIGITAL	? MHz	1.000 kHz	±20	-	0	☑	
System	IMO	DIGITAL	3.000 MHz	3.000 MHz	±1	-	0	☑	
System	BUS_CLK (CPU)	DIGITAL	? MHz	66.000 MHz	±1	-	1	☑	MASTER_CLK
System	MASTER_CLK	DIGITAL	? MHz	66.000 MHz	±1	-	1	☑	PLL_OUT
System	PLL_OUT	DIGITAL	66.000 MHz	66.000 MHz	±1	-	0	☑	IMO
Local	CharComp_clock	DIGITAL	921.600 kHz	916.667 kHz	±1	±5	72	☑	Auto: MASTER_CLK

For example, for a UART configured for 115200 bits per second and 8x oversampling, the system needs a 921.6 kHz clock. When the PLL is configured for 66 MHz, the DWR uses a divide by 72 and generates 66000 / 72 = 916,667 kHz clock. For this example the error is:

(921,6 - 916,667) / 912,6 * 100 = ~0.5 %

The summation of this error plus the clock accuracy error should not exceed the clock tolerance (**T_{CLOCK}**), or you will start to see error in the data.

Clock accuracy is dependent upon the selected IMO clock. It is equal to ±1% for the 3 MHz IMO. The total error will be: 0.5 + 1 = 1.5% and it is less than the minimum clock tolerance for 8x oversampling (2.6%)

Other IMO clock settings have larger accuracy error and are not recommended for using with UART.

t_{CTS_TX} This parameter is characterized based on the UART implementation analysis. The state machine synchronously, to the **f_{CLOCK}** clock, checks the falling edge CTS_N signal and sets TX_EN with up to one clock delay. The TX_EN signal has additional synchronization on the output to remove possible glitches – this adds one clock delay. The Shift register starts pushing TX data out at same time as the TX_EN signal goes high.

t_{TX_TXCLK} The delay time from TX output to TX_CLK, based on the UART implementation analysis, is equal to half a bit length and is delayed one clock to be at the middle of the TX_DATA signal.

$$t_{TX\_TXCLK} = t_{CLOCK} * \left(\frac{Oversampling}{2} + 1 \right)$$

t_{TX_TXDATA} This parameter is characterized based on the UART implementation analysis. The TX signal is additionally synchronized to the **f_{CLOCK}** on the TX_DATA output, therefore one clock delay is present between these signals.

t_{RES} This parameter is characterized based on the UART implementation analysis and on the results of STA. The reset input is synchronous, requiring at least one rising edge of the component clock. Setup time should be added to guarantee not missing the reset signal.

$$t_{RES} = t_{CLOCK} + t_{S\_RES}$$

t_{S_RES} RESET setup time is the pin to internal logic routing path delay of the master component. This is provided in the STA results input setup times as shown below:

-**Setup times to clock BUS_CLK**

Start	Register	Clock	Delay (ns)
RESET(0):iocell.pad_in	RESET(0):iocell.ind	BUS_CLK	18.350
CTS_N(0):iocell.pad_in	CTS_N(0):iocell.ind	BUS_CLK	16.500
RX(0):iocell.pad_in	RX(0):iocell.ind	BUS_CLK	13.630

t_{RX_RXCLK} The delay time from RX to RX_CLK, based on the UART implementation analysis, is equal to half a bit length and is delayed up to one clock to be in the middle of the RX_DATA signal.

$$t_{RX\_RXCLK} = t_{CLOCK} * \left(\frac{Oversampling}{2} + 1 \right)$$

t_{RX_RXINT} The RX_INTERRUPT signal is generated when the Stop bit is received at RX_CLK

t_{RX_RXDATA} The RX signal is additionally synchronized to the **f_{CLOCK}** on the RX_DATA output, therefore up to one clock delay is present between these signals.

t_{RXCLK_RTS} Delay from the last RX_CLK raise to RTS_N active. This happens when the 4 byte FIFO is full. The RTS_N signal is automatically set by hardware as soon as input FIFO is full. The FIFO is loaded with one component clock cycle delay from the last RX_CLK rising edge.

t_{RTS_RX} The delay time between RTS_N Inactive to RX data is equal to:

$t_{RTS\_RX} = t_{PD\_RTS} + RTS_{PD\_PCB} + t_{CTS\_TX(transmitter)} + RX_{PD\_PCB} + t_{S\_RX}]$

Where:

t_{PD_RTS} is the path delay of RTS_N to the pin. This is provided in the STA results clock to output times as shown below.

-**Clock to output times from clock UART_1_IntClock**

Start	Register	End	Delay (ns)
UART_1_IntClock	\UART_1:BUART:sRX:RxShifter:u0\:datapathcell._f0_blk_stat_comb	RTS_N(0):iocell.pad_out	37.996
UART_1_IntClock	Net_47:macrocell.mc_q	TX(0):iocell.pad_out	32.928

RTS_{PD_PCB} is the PCB Path delay from RTS_N pin of the receiver component to the CTS_N pin of the transmitter device.

$t_{CTS\_TX(transmitter)}$ must come from the Transmitter Datasheet.

RX_{PD_PCB} is the PCB path delay from the TX pin of the transmitter device to the RX pin of the receiver component.

t_{S_RX} is the setup time of RX signal. This is provided in the STA results input setup times as shown below.

-Setup times to clock BUS_CLK

Start	Register	Clock	Delay (ns)
RESET(0):iocell.pad_in	RESET(0):iocell.ind	BUS_CLK	18.350
CTS_N(0):iocell.pad_in	CTS_N(0):iocell.ind	BUS_CLK	16.500
RX(0):iocell.pad_in	RX(0):iocell.ind	BUS_CLK	13.630

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
2.0	tx_en output registered	Any combinatorial output can glitch, depend on placement and delay between signals. To remove glitching the outputs should be registered.
	Reset input registered.	Registering improves maximum baud rate when Reset input is used.
	Added characterization data to datasheet	
	Minor datasheet edits and updates	