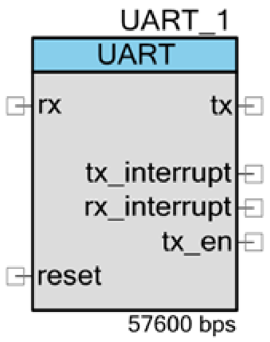
PSoC® Creator™ Component Datasheet

Universal Asynchronous Receiver Transmitter (UART)

2.50



Features

- 9-bit address mode with hardware address detection
- Baud rates from 110 to 921600 bps or arbitrary up to 4 Mbps
- RX and TX buffers = 4 to 65535
- Detection of Framing, Parity, and Overrun errors
- Full Duplex, Half Duplex, TX only, and RX only optimized hardware
- Two out of three voting per bit
- Break signal generation and detection
- 8x or 16x oversampling

General Description

The UART provides asynchronous communications commonly referred to as RS232 or RS485. The UART Component can be configured for Full Duplex, Half Duplex, RX only, or TX only versions. All versions provide the same basic functionality. They differ only in the amount of resources used.

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 and hardware FIFOs help to ensure that data will not be missed. This allows the CPU to spend more time on critical real time tasks rather than servicing the UART.

For most use cases, you can easily configure the UART by choosing the baud rate, parity, number of data bits, and number of start bits. The most common configuration for RS232 is often listed as “8N1,” which is shorthand for eight data bits, no parity, and one stop bit. This is the default configuration for the UART Component. Therefore, in most applications you only need to set the baud rate. A second common use for UARTs is in multidrop RS485 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.

UARTs have been around a long time, so there have been many physical-layer and protocol-layer variations over time. These include, but are not limited to, RS423, DMX512, MIDI, LIN bus, legacy terminal protocols, and IrDa. To support the commonly used UART variations, 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 choose 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

Use the UART any time a compatible asynchronous communications interface is required, especially RS232 and RS485 and other variations. You can also use the UART to create more advanced asynchronous based protocols such as DMX512, LIN, and IrDa, or customer or industry proprietary.

Do not use a UART in those cases where a specific Component has already been created to address the protocol. For example if a LIN or MIDI Component is provided, it has a specific implementation providing both hardware and protocol layer functionality. The UART is not needed in this case (subject to Component availability).

PSoC 4 Glitch Avoidance at System Reset

For PSoC 4 devices, use the following method to avoid low impulses on the TX output terminal when coming out of System Reset. This is important if you’re concerned with TX pin activity at either chip startup or when coming out of Hibernate mode:

1. Connect an external pull-up resistor for the Tx_1 output pin on your device.
2. From the PSoC Creator schematic, open the Tx_1 Pins Configure dialog.
   a. On the **Pins > General** tab, configure **Drive mode** to "High impedance digital."
   b. On the **Built-in** tab, change **CY_SUPPRESS_API_GEN** from true to false.
3. In your main code, change the drive mode of the TX_1 pin to "Strong drive" using the Tx_1_SetDriveMode(Tx_1_DM_STRONG) API, after calling the UART_1_Start() API.

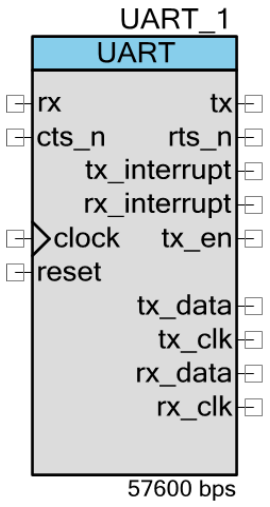
Additional Reading

Refer to the following documents for more information:

- AN54460 - PSoC® 3 and PSoC 5LP Interrupts
- AN90799 - PSoC® 4 Interrupts

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.



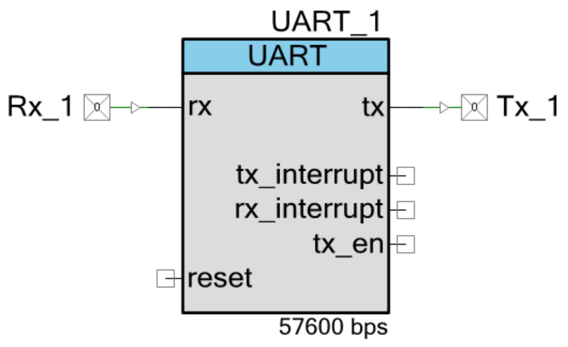
Input	May Be Hidden	Description
rx	Y	The rx input carries the input serial data from another device on the serial bus. This signal should be synchronized to the BUS_CLK by enabling **Input Synchronized** parameter in the related **Digital Input Pin** Component or by using **Sync** Component. This input 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 shows that another device is ready to receive data. It is an active-low input, (_n). This input is visible if the **Flow Control** parameter is set to **Hardware**.
clock	Y	The clock input defines the baud rate (bit-rate) of the serial communication. The baud-rate is one-eighth or one-sixteenth the input clock frequency depending on the **Oversampling Rate** parameter. This input is visible if the **Clock Selection** parameter is set to **External Clock**. If the internal clock is selected, you must define the desired baud rate during configuration and PSoC Creator solves the necessary clock frequency.
reset	N	The reset input resets the UART state machines (RX and TX) to the idle state. This throws out any data that was currently being transmitted or received. This input is a synchronous reset that requires at least one rising edge of the clock. The reset input may be left floating with no external connection. If nothing is connected to the reset line the Component will assign it a constant logic 0.

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).** Cypress recommends that you use an external pull-up resistor to protect the receiver from unexpected low impulses during active System Reset.

Output	May Be Hidden	Description
rts_n	Y	The rts output tells another device that your device is ready to receive data. This output is active-low (_n). The RTS signal goes high when internal FIFO and RX buffer, allocated by RX Buffer Size parameter (when it is greater than 4), are full. This output is visible if the **Flow Control** parameter is set to **Hardware**.
tx_interrupt	Y	The tx_interrupt output is the logical OR of the group of possible interrupt sources. This signal goes 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 goes 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_en	Y	The tx_en output is used primarily for RS485 communication to show that your device is transmitting on the bus. This output goes high before a transmit starts and low when transmit is complete. This shows a busy bus to the rest of the devices on the bus. This output is visible when the **Hardware TX Enable** parameter is selected.
tx_data	Y	The tx_data output is 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 selected.
tx_clk	Y	The tx_clk output provides the 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 selected.
rx_data	Y	The rx_data output is 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 selected.
rx_clk	Y	The rx_clk output provides the 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 selected.

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.



Component Parameters

Drag a UART Component onto your design and double-click it to open the **Configure** dialog.

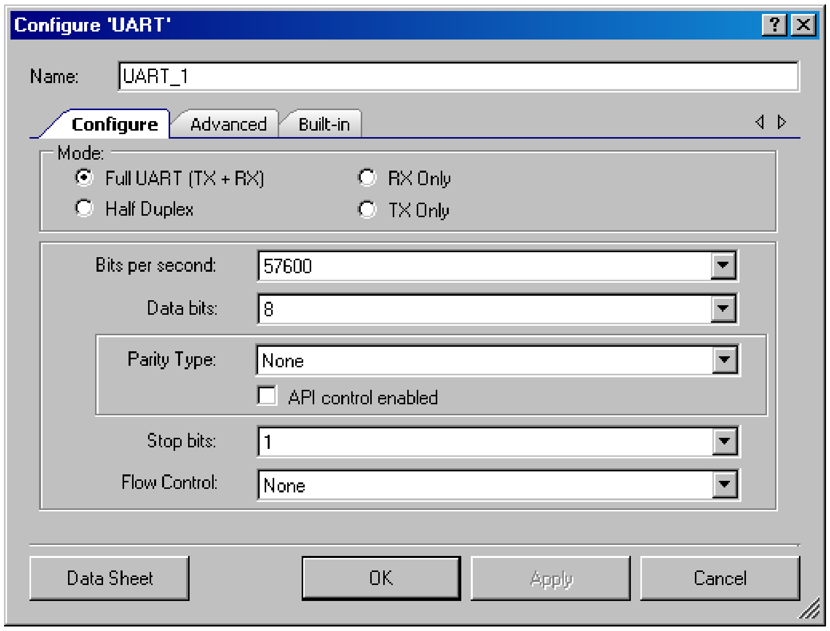
Hardware versus Software Options

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 you set these parameters before build time you are setting their initial value, which you can change 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, because the PC using the hyperterminal is quite often the other side of the bus.



Name: UART_1

Mode: [✓ Full UART (TX + RX)] Half Duplex RX Only TX Only

Bits per second: 57600

Data bits: 8

Parity Type: None

☐ API control enabled

Stop bits: 1

Flow Control: None

All of these options are hardware configuration options.

Mode

This parameter defines the functional Components you want 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), RS232 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), PSoC Creator generates the necessary clock to achieve this baud rate.

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**.

- Eight data bits is the default configuration, sending a byte per transfer.
- The 9-bit mode does not transmit 9 data bits; the ninth bit takes the place of the parity bit as an indicator of address using Mark/Space parity. Mark/Space parity should be selected if you are using 9 data bits mode.

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 uses more resources.

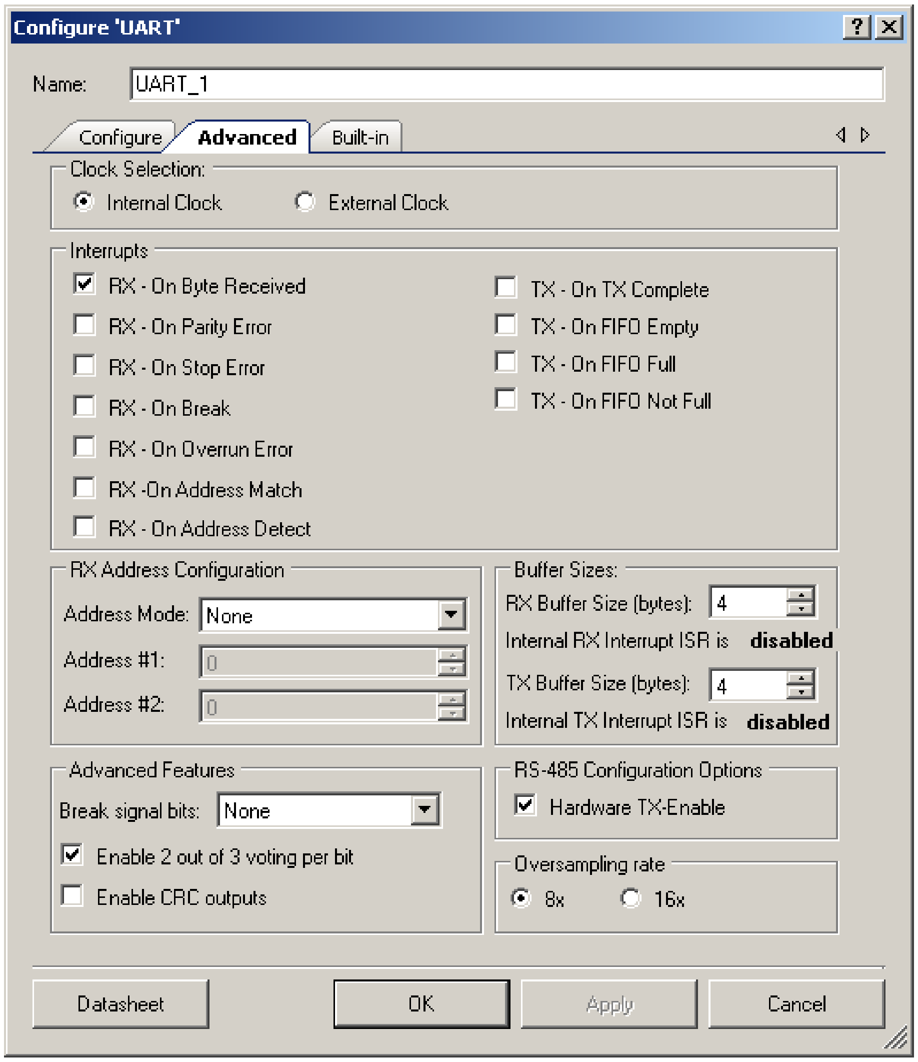
Stop bits

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



Configure 'UART'

Name: UART_1

Configure | **Advanced** | Built-in

Clock Selection:
[✓ Internal Clock] External Clock

Interrupts
[✓ RX - On Byte Received]
RX - On Parity Error
RX - On Stop Error
RX - On Break
RX - On Overrun Error
RX -On Address Match
RX - On Address Detect
TX - On TX Complete
TX - On FIFO Empty
TX - On FIFO Full
TX - On FIFO Not Full

RX Address Configuration
Address Mode: None
Address #1: 0
Address #2: 0

Buffer Sizes:
RX Buffer Size (bytes): 4
Internal RX Interrupt ISR is **disabled**
TX Buffer Size (bytes): 4
Internal TX Interrupt ISR is **disabled**

Advanced Features
Break signal bits: None
[✓ Enable 2 out of 3 voting per bit]
Enable CRC outputs

RS-485 Configuration Options
[✓ Hardware TX-Enable]

Oversampling rate
[✓ 8x] 16x

Datasheet | OK | Apply | Cancel

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 **Internal Clock**, the required clock frequency is calculated and configured by PSoC Creator. In the **External Clock** mode, the Component does not control the baud rate but can calculate the expected baud rate.

If this parameter is set to **Internal Clock**, 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 (UART_RX_STS_MRKSPC status) 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 or any other additional addresses.
- **Software Detect to Buffer** – Hardware indicates the detection of an address byte (UART_RX_STS_MRKSPC status). Software, embedded to RX ISR, reads the byte and determines if this address matches the device addresses defined as in the **Address #1** or **Address #2** parameters (uses UART_RX_STS_ADDR_MATCH status). It then copies all addressed data, along with the address byte, into the RX buffer defined by the **RX Buffer Size** parameter. **RX Buffer Size** should be set manually to greater than 4. Unaddressed data is read from FIFO, but not written to the buffer.
- **Hardware Byte By Byte** – Hardware detects addressed bytes and forces an interrupt (RX - On Byte Received) to move all data along with the address from the hardware FIFO into the data buffer defined by **RX Buffer Size**. Hardware does not save unaddressed bytes to the FIFO, and does not generate any interrupt for them.
- **Hardware Detect to Buffer** – Hardware detects addressed bytes and forces an interrupt (RX - On Byte Received) to move only the data (address byte is not included) from the hardware FIFO into the data buffer defined by **RX Buffer Size**. Hardware does not save unaddressed bytes to the FIFO, and does not generate any interrupt for them.
- **None** – No RX address detection is implemented.

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 **Address Mode** parameter. The hardware for **RX Address #2** does not implemented in **Half Duplex** mode. The parameters are available to firmware for the software address modes.

Advanced Features

- **Break signal bits** – Break signal bits parameter enables Break signal generation and detection and defines the number of logic 0s bits transmitted. Valid values include 11 to 14. This option saves resources when set to **None**.
- **Enable 2 out of 3 voting per bit** – The **Enable 2 out of 3 voting per bit** parameter enables or disables the error compensation algorithm. Disabling this option saves resources. For more information, see the API Memory Usage section of this datasheet.

- **Enable CRC outputs** – The **Enable CRC outputs** parameter enables or disables tx_data, tx_clk, rx_data, and rx_clk outputs. They are 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. Disabling this option saves resources.

Hardware TX Enable

This parameter enables or disables the use of the TX-Enable output of the TX UART. This signal is used in RS485 communications. The hardware provides the functionality of this output automatically, based on buffer conditions.

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 to 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 reconfigure these modes at any time; these parameters define an initial configuration.

- **RX - On Byte Received** (UART_RX_STS_FIFO_NOTEMPTY)
- **RX - On Parity Error** (UART_RX_STS_PAR_ERROR)
- **RX - On Stop Error** (UART_RX_STS_STOP_ERROR)
- **RX - On Break** (UART_RX_STS_BREAK)
- **RX - On Overrun Error** (UART_RX_STS_OVERRUN)
- **RX - On Address Match** (UART_RX_STS_ADDR_MATCH)
- **RX - On Address Detect** (UART_RX_STS_MRKSPC)
- **TX - On TX Complete** (UART_TX_STS_COMPLETE)
- **TX - On FIFO Empty** (UART_TX_STS_FIFO_EMPTY)
- **TX - On FIFO Full** (UART_TX_STS_FIFO_FULL)
- **TX - On FIFO Not Full** (UART_TX_STS_FIFO_NOT_FULL)

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, depending on the desired functionality.

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 equal to 4 bytes. Buffer sizes greater than 4 bytes require the use of interrupts to handle moving the data from the receive FIFO into this buffer. The UART_GetChar() or UART_ReadRXData() functions 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 equal to four bytes; otherwise, the RAM buffer is allocated. Buffer sizes greater than four bytes require the use of interrupts to handle moving 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 four 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 four 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.

When the (Rx/Tx) Buffer size is set to 4, the UART Component uses the Datapath's 4-byte deep Hardware FIFO for data movement. So, to handle any interrupt condition, use an external interrupt (connect an ISR Component to rx_interrupt or tx_interrupt).

When the (Rx/Tx) Buffer size is set to a value greater than 4, the UART Component uses the internal interrupts to move the data from Hardware FIFO to a software FIFO. In this case, use the internal ISR to add custom code to process the data in the "User Code" regions.

Note External interrupts are generated only with respect to the Hardware FIFO.

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 calculates the needed frequency and clock source and generates the resource needed for implementation. Otherwise, you must supply the clock and calculate the baud rate at one-eighth or one-sixteenth the input clock frequency.

The clock tolerance should be a maximum of ±2 percent. A warning is generated if the clock cannot be generated within this limit. In that case, you should change the Master Clock in the DWR or you should use an external crystal-based clock.

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.”

Functions

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

Function	Description
UART_GetChar()	Returns the next byte of received data
UART_GetByte()	Reads the UART RX buffer immediately and returns the received character and error condition
UART_GetRxBufferSize()	Returns the number of received bytes available in the RX buffer.
UART_ClearRxBuffer()	Clears the memory array of all received data
UART_SetRxAddressMode()	Sets the software-controlled Addressing mode used by the RX portion of the UART
UART_SetRxAddress1()	Sets the first of two hardware-detectable addresses
UART_SetRxAddress2()	Sets the second of two hardware-detectable addresses
UART_EnableTxInt()	Enables the internal interrupt irq
UART_DisableTxInt()	Disables the internal interrupt irq
UART_SetTxInterruptMode()	Configures the TX interrupt sources enabled
UART_WriteTxData()	Sends a byte without checking for buffer room or status
UART_ReadTxStatus()	Reads the status register for the TX portion of the UART
UART_PutChar()	Puts a byte of data into the transmit buffer to be sent when the bus is available
UART_PutString()	Places data from a string into the memory buffer for transmitting
UART_PutArray()	Places data from a memory array into the memory buffer for transmitting
UART_PutCRLF()	Writes a byte of data followed by a Carriage Return and Line Feed to the transmit buffer
UART_GetTxBufferSize()	Returns the number of bytes in the TX buffer which are waiting to be transmitted.
UART_ClearTxBuffer()	Clears all data from the TX buffer
UART_SendBreak()	Transmits a break signal on the bus
UART_SetTxAddressMode()	Configures the transmitter to signal the next bytes as address or data
UART_LoadRxConfig()	Loads the receiver configuration. Half Duplex UART is ready for receive byte
UART_LoadTxConfig()	Loads the transmitter configuration. Half Duplex UART is ready for transmit byte
UART_Sleep()	Stops the UART operation and saves the user configuration
UART_Wakeup()	Restores and enables the user configuration
UART_Init()	Initializes default configuration provided with customizer
UART_Enable()	Enables the UART block operation
UART_SaveConfig()	Save the current user configuration
UART_RestoreConfig()	Restores the user configuration

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.

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.

uint8 UART_ReadControlRegister(void)

Description: Returns the current value of the control register.

Return Value: uint8: Contents of the control register The following defines can be used to interpret the returned value.

See the Control registers description near the end of this datasheet for more information.

Value	Description
UART_CTRL_HD_SEND	Configures whether the half duplex UART (if enabled) is in RX mode (0), or in TX mode (1).
UART_CTRL_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_PARITY_TYPE_MASK	Two bits wide field configuring the parity for the next transfer if software configurable. The following defines, shifted left by UART_CTRL_PARITY_TYPE0_SHIFT, can be used to recognize 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	Three bits wide field configuring the expected hardware addressing operation for the UART receiver. The following defines, shifted left by UART_CTRL_RXADDR_MODE0_SHIFT, can be used to recognize 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

void UART_WriteControlRegister(uint8 control)

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

Note that to change control register it must be read first using UART_ReadControlRegister function, modified and then written.

Parameters: uint8 control: Control register value.

See the Control registers description near the end of this datasheet for more information.

Value	Description
UART_CTRL_HD_SEND	Configures whether the half duplex UART (if enabled) is in RX mode (0), or in TX mode (1).
UART_CTRL_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_PARITY_TYPE_MASK	Two bits wide field configuring the parity for the next transfer if software configurable. The following defines, shifted left by UART_CTRL_PARITY_TYPE0_SHIFT, can be used to recognize 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	Three bits wide field configuring the expected hardware addressing operation for the UART receiver. The following defines, shifted left by UART_CTRL_RXADDR_MODE0_SHIFT, can be used to recognize 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_BUFFERHardware Detect to Buffer address detection ▪ UART__B_UART__AM_NONE No address detection

void UART_EnableRxInt(void)

Description: Enables the internal receiver interrupt.

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.

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.

uint8 UART_ReadRxData(void)

Description: Returns the next byte of received data. This function returns data without checking the status. You must check the status separately.

Return Value: uint8: Received data from RX register

uint8 UART_ReadRxStatus(void)

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

Return Value: uint8: Current RX status register value

Value	Description
UART_RX_STS_FIFO_NOTEMPTY	If set, indicates the FIFO has data available.
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. The framing error is caused when the UART hardware sees the logic 0 where the stop bit should be (logic 1).
UART_RX_STS_BREAK	If set, indicates a break was detected.
UART_RX_STS_OVERRUN	If set, indicates the FIFO buffer was overrun.
UART_RX_STS_ADDR_MATCH	Indicates that the received byte matches one of the two addresses available for hardware address detection. It is not implemented if **Address Mode** is set to **None**. In **Half Duplex** mode, only **Address #1** is implemented for this detection.
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 not implemented if **Address Mode** is set to **None**.
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 datasheet.

uint8 UART_GetChar(void)

Description: Returns the last received byte of data. UART_GetChar() is designed for ASCII characters and returns a uint8 where 1 to 255 are values for valid characters and 0 indicates an error occurred or no data is present.

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.

uint16 UART_GetByte(void)

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

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

uint8/uint16 UART_GetRxBufferSize(void)

Description: Returns the number of received bytes available in the RX buffer.

- RX software buffer is disabled (**RX Buffer Size** parameter is equal to 4): returns 0 for empty RX FIFO or 1 for not empty RX FIFO.
- RX software buffer is enabled: returns the number of bytes available in the RX software buffer. Bytes available in the RX FIFO do not take to account.

Return Value: uint8/uint16: Number of bytes in the RX buffer. Return value type depends on **RX Buffer Size** parameter.

void UART_ClearRxBuffer(void)

Description: Clears the receiver memory buffer and hardware RX FIFO of all received data.

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

void UART_SetRxAddress1(uint8 address)

Description: Sets the first of two hardware-detectable receiver addresses.

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

void UART_SetRxAddress2(uint8 address)

Description: Sets the second of two hardware-detectable receiver addresses.

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

void UART_EnableTxInt(void)

Description: Enables the internal transmitter interrupt.

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.

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_FIFO_NOT_FULL	Interrupt when TX FIFO is not full

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. You must check status separately.

Parameters: uint8 txDataByte: data byte

uint8 UART_ReadTxStatus(void)

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

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_FIFO_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

void UART_PutString(const char8 string[])

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

Parameters: const char8 string[]: Pointer to the null terminated string array residing in RAM or ROM

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

void UART_PutArray(const uint8 string[], uint8/uint16 byteCount)

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

Parameters: const uint8 string[]: Address of the memory array residing in RAM or ROM

uint8/uint16 byteCount: Number of bytes to be transmitted. The type depends on **TX Buffer Size** parameter.

Side Effects: If there is not enough memory in the TX buffer for the entire array, this function blocks 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

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

uint8/uint16 UART_GetTxBufferSize(void)

Description: Returns the number of bytes in the TX buffer which are waiting to be transmitted.

- TX software buffer is disabled (**TX Buffer Size** parameter is equal to 4): returns 0 for empty TX FIFO, 1 for not full TX FIFO or 4 for full TX FIFO.
- TX software buffer is enabled: returns the number of bytes in the TX software buffer which are waiting to be transmitted. Bytes available in the TX FIFO do not take to account.

Return Value: uint8/uint16: Number of bytes in the TX buffer. Return value type depends on the **TX Buffer Size** parameter.

void UART_ClearTxBuffer(void)

Description: Clears all data from the TX buffer and hardware TX FIFO.

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

void UART_SendBreak(uint8 retMode)

Description: Transmits a break signal on the bus.

Note The break signal length is defined by the UART bit time; the maximum value is 14 bits. This can limit the break length for some UART variants. In these cases, the GPIO functionality can be used for logner break length generation.

Parameters: uint8 retMode: Send Break return mode. See the following table for options.

Options	Description
UART_SEND_BREAK	Initialize registers for break, send the Break signal and return immediately
UART_WAIT_FOR_COMPLETE_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_COMPLETE_REINIT. This option is recommended for most cases

Side Effects: The UART_SendBreak() function initializes registers to send a 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.

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

Side Effects: Valid only in half duplex mode. You must make sure 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.

Side Effects: Valid only in half duplex mode. You must make sure 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.

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.

Side Effects: This function clears the RX and TX software buffers and hardware 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_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.

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.

void UART_SaveConfig(void)

Description: This function saves the Component configuration and nonretention registers. It also saves 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.

Side Effects: All nonretention registers except FIFO are saved to RAM.

void UART_RestoreConfig(void)

Description: Restores the user configuration of nonretention registers.

Side Effects: All nonretention 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.

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. For correct operation of the Component, the UART must be 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 is used by interrupts, when the **RX Buffer Size** parameter is set to more than 4, to store received data. It is also used by UART_ReadRxData() and UART_GetChar() 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 is also used by the UART_ReadRxData() and UART_GetChar() functions to identify new data. Cleared to zero by the UART_ClearRxBuffer() function.
UART_rxBufferRead	This variable is used by the UART_ReadRxData() and UART_GetChar() functions as a cyclic index for UART_rxBuffer to read data. Cleared to zero by the UART_ClearRxBuffer() function.
UART_rxBufferLoopDetect	This variable is set to one in RX interrupt when the UART_rxBufferWrite index overtakes the UART_rxBufferRead index. This is a preoverload condition that affects UART_rxBufferOverflow when the next byte is received. It is set to zero when the UART_ReadRxData() or UART_GetChar() function is called. Cleared to zero by the UART_ClearRxBuffer() function.
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_rxBuffer to write new data. This condition is returned and cleared to zero by the UART_ReadRxStatus() function as an UART_RX_STS_SOFT_BUFF_OVER bit along with RX Status register bits. Cleared to zero by the UART_ClearRxBuffer() function.

Variable	Description
UART_txBuffer	This is a RAM-allocated TX buffer of user-defined length. This buffer is used by sending APIs when the **TX Buffer Size** parameter is set to more than 4, to store data for transmitting. It is also used by the TX interrupt to move data into the hardware FIFO.
UART_txBufferWrite	This variable is used by the UART_WriteTxData(), UART_PutChar(), UART_PutString(), UART_PutArray(), and UART_PutCRLF() functions 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() function.
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() function.

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 that you chose 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 that you chose 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_TX_BUFFER_SIZE	Defines the amount of memory to allocate for the TX memory array buffer. This does not include the four bytes included in the FIFO.
UART_RX_BUFFER_SIZE	Defines the amount of memory to allocate for the RX memory array buffer. This does not include the four 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_RX_HW_ADDRESS1	Defines the initial address selected in the configuration GUI. This address is loaded into the corresponding hardware register at startup of the UART.
UART_RX_HW_ADDRESS2	Defines the initial address selected in the configuration GUI. This address is loaded into the corresponding hardware register at startup of the UART.

Bootloader Support

The UART Component can be used as a communication Component for the Bootloader. Use the following configuration to support communication protocol from an external system to the Bootloader:

- **Mode**: Full UART (TX + RX)
- **Bits per second**: Must match Host (boot device) data rate.
- **Data bits**: 8
- **Parity Type, Stop bits, Flow Control**: Must match Host (boot device) configuration.
- **RX Buffer Size (bytes)**: 64
- **TX Buffer Size (bytes)**: 64

For more information about the Bootloader, refer to the Bootloader/Bootloadable Component Datasheet.

The UART Component provides a set of API functions for Bootloader use.

Function	Description
UART_CyBtldrCommStart()	Starts the UART Component and enables its interrupt.
UART_CyBtldrCommStop()	Disables the UART Component and disables its interrupt.
UART_CyBtldrCommReset()	Resets the receive and transmit communication buffers.
UART_CyBtldrCommRead()	Allows the caller to read data from the bootloader host. This function manages polling to allow a block of data to be completely received from the host device.
UART_CyBtldrCommWrite()	Allows the caller to write data to the boot loader host. This function uses a blocking write function for writing data using UART communication Component.

void UART_CyBtldrCommStart(void)

Description: Starts the UART communication Component.

Side Effects: This Component automatically enables global interrupt.

void UART_CyBtldrCommStop(void)

Description: This function disables the UART Component and disables its interrupt.

void UART_CyBtldrCommReset(void)

Description: Resets the receive and transmit communication Buffers.

cystatus UART_CyBtldrCommRead(uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description: This function allows the caller to read data from the bootloader host. The function manages polling to allow a block of data to be completely received from the bootloader host.

Parameters: uint8 pData[]: Pointer to storage for the block of data to be read from the bootloader host

uint16 size: Number of bytes to be read

uint16 *count: Pointer to the variable to write the number of bytes actually read

uint8 timeOut: Number of units in 10 ms to wait before returning because of a timeout

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information, see the "Return Codes" section of the *System Reference Guide*.

cystatus UART_CyBtldrCommWrite(const uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description: Allows the caller to write data to the boot loader host. This function uses a blocking write function for writing data using UART communication Component.

Parameters: const uint8 pData[]: Pointer to the block of data to be written to the bootloader host

uint16 size: Number of bytes to be written

uint16 *count: Pointer to the variable to write the number of bytes actually written

uint8 timeOut: Number of units in 10 ms to wait before returning because of a timeout

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information see the "Return Codes" section of the *System Reference Guide*.

Macro Callbacks

Macro callbacks allow users to execute code from the API files that are automatically generated by PSoC Creator. Refer to the PSoC Creator Help and *Component Author Guide* for more details.

In order to add code to the macro callback present in the Component's generated source files, perform the following:

- Define a macro to signal the presence of a callback (in *cyapicallbacks.h*). This will "uncomment" the function call from the Component's source code.
- Write the function declaration (in *cyapicallbacks.h*). This will make this function visible by all the project files.

- Write the function implementation (in any user file).

Callback Function [1]	Associated Macro	Description
UART_RXISR_EntryCallback	UART_RXISR_ENTRY_CALLBACK	Used at the beginning of the _RXISR() interrupt handler to perform additional application-specific actions.
UART_RXISR_ExitCallback	UART_RXISR_EXIT_CALLBACK	Used at the end of the _RXISR() interrupt handler to perform additional application-specific actions.
UART_TXISR_EntryCallback	UART_TXISR_ENTRY_CALLBACK	Used at the beginning of the _TXISR() interrupt handler to perform additional application-specific actions.
UART_TXISR_ExitCallback	UART_TXISR_EXIT_CALLBACK	Used at the end of the _TXISR() interrupt handler to perform additional application-specific actions.
UART_RXISR_ERROR_Callback	UART_RXISR_ERROR_CALLBACK	Used in the _RXISR() interrupt handler to perform additional application-specific actions.
UART_RXISR_ERROR_Callback	UART_RXISR_ERROR_CALLBACK	Used in the _RXISR() interrupt handler to perform additional application-specific actions.

Sample Firmware Source Code

PSoC Creator provides many code example projects that include schematics and example code in the Find Code Example 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 Code Example” topic in the PSoC Creator Help for more information.

1 The callback function name is formed by Component function name optionally appended by short explanation and “Callback” suffix.

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) /* Use any other address */
        {
            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;
            }
        }
    }
}
```

Printf() function Usage Model

The printf() function formats a series of strings and numeric values and builds a string to write to the output stream. It has different implementation in different compilers. Keil compiler use the putchar(), GCC use _write(), MDK and RVDS use fputc() while IAR use __write() function to send the data. Application should revise these functions and call the communication Component API to send data via selected interface.

Example:

```
#include <project.h>
#include <stdio.h>

#if (CY_PSOC3)
    /* For Keil compiler revise putchar() function with communication
       Component which has to send data */
    char putchar( char c)
    {
        UART_PutChar(c);
        return c;
    }
#else
    #if defined(__ARMCC_VERSION)

    /* For MDK/RVDS compiler revise fputc function */
    struct __FILE
    {
        int handle;
    };

    enum
    {
        STDIN_HANDLE,
        STDOUT_HANDLE,
        STDERR_HANDLE
    };

    FILE __stdin = {STDIN_HANDLE};
    FILE __stdout = {STDOUT_HANDLE};
    FILE __stderr = {STDERR_HANDLE};

    int fputc(int ch, FILE *file)
    {
        int ret = EOF;
        switch( file->handle )
        {
            case STDOUT_HANDLE:
                UART_PutChar(ch);
                ret = ch;
                break;

            case STDERR_HANDLE:
                ret = ch;
                break;
            default:
                file = file;
                break;
        }
        return(ret);
    }

#elif defined (__ICCARM__)      /* IAR */
/* For IAR compiler revise __write() function for printf functionality */
size_t __write(int handle, const unsigned char * buffer, size_t size)
{
    size_t nChars = 0;
    for (/* Empty */; size != 0; --size)
    {
        UART_PutChar(*buffer++);
        ++nChars;
    }
        return (nChars);
}

#else  /* (__GNUC__)  GCC */

    /* For GCC compiler revise _write() function */
    int _write(int file, char *ptr, int len)
    {
        int i;
        for (i = 0; i < len; i++)
        {
            UART_PutChar(*ptr++);
        }
        return(len);
    }

    #endif  /* (__ARMCC_VERSION) */

#endif /* CY_PSOC3 */

/* Add an explicit reference to the floating point printf library to allow
   the usage of floating point conversion specifier */
#if defined (__GNUC__)
    asm (".global _printf_float");
#endif

void main()
{
    uint32 i = 444444444;
    float f = 55.555f;

    CyGlobalIntEnable;  /* Enable interrupts */

    UART_Start();     /* Start communication Component */

    /* Use printf() function which will send formatted data through "UART" */
    printf("Test printf function. long:%ld,float:%f \n",i,f);
}
```

The log from terminal software:

```
Test printf function. long:444444444,float:55.555
```

Note The printf() function prepares the text stream in the buffer and executes it when it receives new-line character '\n'.

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the Component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator Components
- specific deviations – deviations that are applicable only for this Component

This section provides information on Component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The UART Component has the following specific deviations:

MISRA-C:2004 Rule	Rule Class (Required/ Advisory)	Rule Description	Description of Deviation(s)
10.5	R	If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	The return value is in not cast to uint16 after applying << operation in GetByte() function. There is no side effect in this particular case.

This Component has the following embedded Components: Interrupt, Clock. Refer to the corresponding Component datasheet for information on their MISRA compliance and specific deviations.

API Memory Usage

The Component memory usage varies significantly, depending on the compiler, device, number of APIs used and Component configuration. The following table provides the memory usage for all APIs available in the given Component configuration.

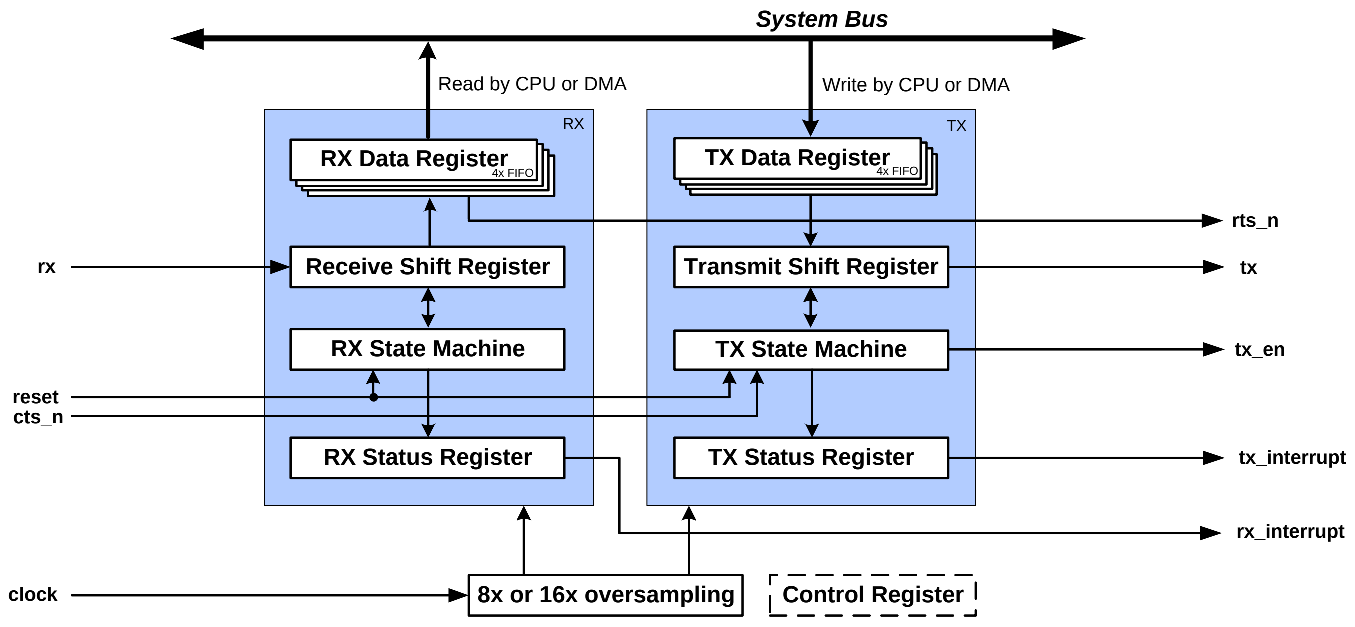
The measurements have been done with associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 4 (GCC)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Full UART	1633	23	N/A[2]	N/A	1860	23
Simple UART	605	3	822	3	814	3
Half Duplex	645	3	918	3	918	3
RX Only	288	2	432	2	444	2
TX Only	463	3	584	3	596	3

Functional Description

Block Diagram

The UART is implemented in the UDB blocks as shown in the following diagram.



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 of how to use the UART Component.

2. The maximum UART configuration doesn't fit in to PSoC 4 device because of maximum number or UDB macrocells exceeded (max=32, need=59).

Default Configuration

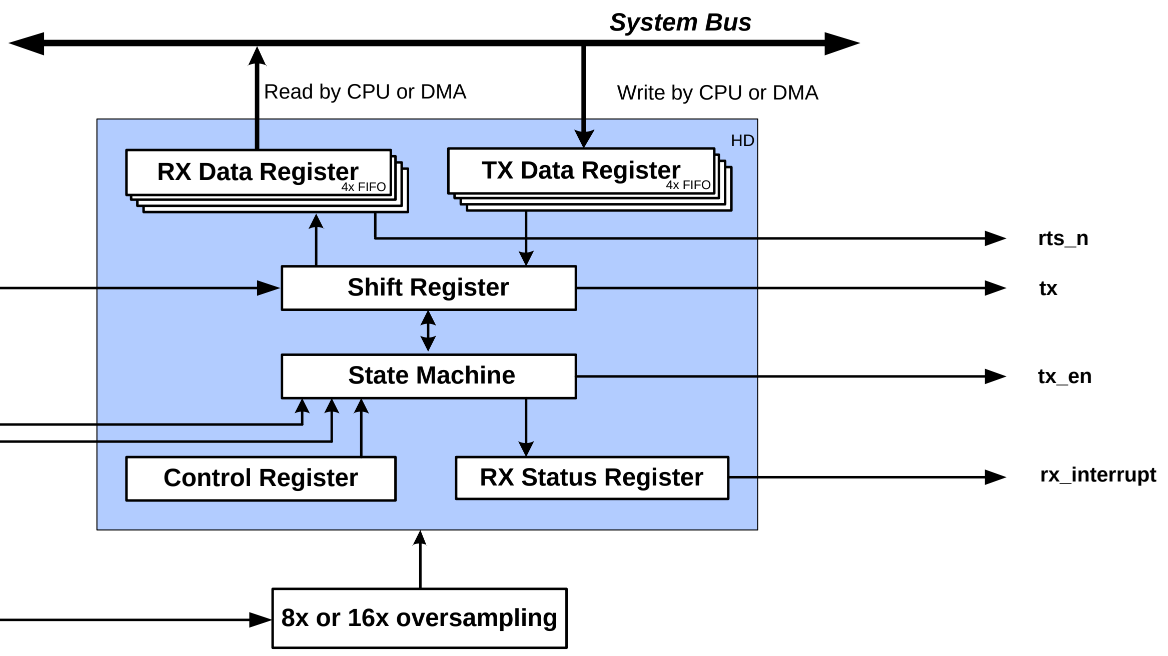
The default configuration for the UART is as an 8-bit UART with no flow control and no parity, running at a baud rate of 57.6 Kbps.

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

The UART implementation in Half Duplex mode is shown in the following block diagram.



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() function.

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

In **Half Duplex** mode, the Address2 parameter does not work for hardware address match status (UART_RX_STS_ADDR_MATCH), but it can still be used by software.

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 second**: 115200, **Data bits**: 8, **Parity Type**: None, **Rx Buffer Size**: 4, **Tx Buffer Size**: 4.

```
#include <project.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 until 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 CTS 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 RTS 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 RS485 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 automatically sends 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 <project.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

There are four different modes for the receiver:

1. Software Byte by Byte

Use this mode when you need custom code.

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

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 <project.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;
    }
}
```

2. Software Detect to Buffer

All necessary code is implemented in RX ISR in this mode.

- Configure UART to **Data bits**: 9, **Parity Type**: Mark/Space, **RX Buffer Size**: 20, **Address Mode**: Software Detect to Buffer, **Address#1**: 31.

Receive addressed packet example:

```
void main()
{
    uint8 rec_data = 0u;

    CyGlobalIntEnable;           /* Enable interrupts */
    UART_RX_Start();
    for(;;)
    {
        rec_data = UART_RX_GetChar();
        if(rec_data > 0u)
        {
            /* add analyze here */
        }
    }
}
```

3. Hardware Byte By Byte

The hardware filters unaddressed packets. The main code for this mode will look similar to the previous example.

- Configure UART to **Data bits**: 9, **Parity Type**: Mark/Space, **RX Buffer Size**: 20, **Address Mode**: Hardware Byte By Byte, **Address#1**: 31.

4. Hardware Detect to Buffer

This is the preferred mode for a project that doesn't require an address byte. The hardware filters the unaddressed packets within an address byte. The main code receives the addressed data only bytes.

- Configure UART to **Data bits**: 9, **Parity Type**: Mark/Space, **RX Buffer Size**: 20, **Address Mode**: Hardware Detect to Buffer, **Address#1**: 31.

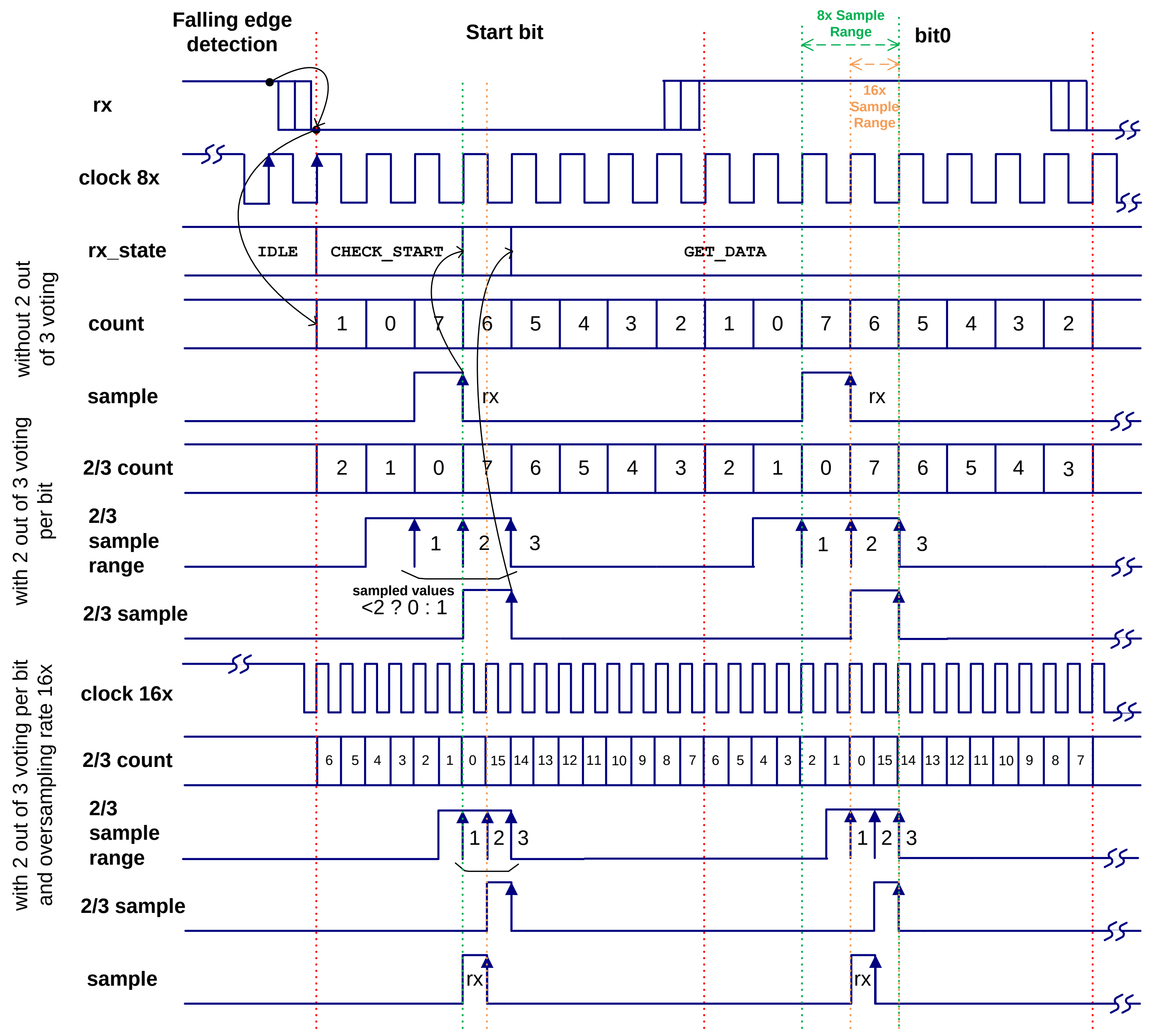
UART Stop Bits: One, Two

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 process the data before more data is sent. Sending two bit-widths of the stop signal, the transmitter 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 three 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 three times. If the RX line is verified to be low (for example, 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 that is enabled on counter cycles 3 to 5 (3 cycles). This counter counts the number of 1s seen on the RX input. If the counter value is 2 or greater, the output of this counter is 1; otherwise, the output is 0. This value is sampled into the datapath as the RX value on the fifth clock edge. If voting is not enabled, the RX input is simply sampled on the fourth clock edge after the detection of the start bit, and continues every eighth positive clock edge after that.

When an oversampling rate of 16x is enabled, the voting algorithm occurs on counter cycles 7 to 9 and the output of the counter is sampled by the datapath as the RX value on the ninth cycle. If voting is not enabled, the RX input is sampled on the eighth clock edge and continues on every sixteenth clock edge after that.

Registers

The API functions previously described provide support for the common run time 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 because these bit fields may be moved 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 not set to None.
UART_RX_STS_BREAK *	Indicates that a break signal was detected in the transfer.

Define	Description
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 the Receive FIFO is Not Empty.
UART_RX_STS_ADDR_MATCH *	Indicates that the received byte matches one of the two addresses available for hardware address detection. It is only implemented if the address mode is not set to None. In **Half Duplex** mode, only **Address #1** is implemented for this 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 because the status of that buffer is not indicated in hardware; it must be checked in firmware.
UART_TX_STS_FIFO_NOT_FULL[3]	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.

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 Resources section. When you read or write 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_CTRL_HD_SEND

Used to dynamically reconfigure 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.

[3] Not available in half-duplex mode.

UART_CTRL_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 reset by firmware.

UART_CTRL_PARITY_TYPE_MASK

The parity type control is a 2-bit-wide field that defines the parity operation for the next transfer. This bit field is two consecutive bits in the control register. 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() function 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 is three consecutive bits in the control register. 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 **Address Mode** configuration parameter and can be modified during run time using the UART_WriteControlRegister() function 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 functions 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 is transmitted on the bus. As soon as this register (FIFO) is empty, no more data is transmitted on the bus until it is added to the FIFO. DMA may be set up 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
UART_TXDATA_PTR	TX data register address

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 indicates that data has been received, at which time the data can be retrieved with either the CPU or DMA. DMA may be set up 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
UART_RXDATA_PTR	RX data register address

Constants

There are several constants defined for the status and control registers as well as some enumerated types. Most of these are described earlier 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 the CY_GET_REGX and CY_SET_REGX macros must be 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.

The control and status register bits must 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 that 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).

Resources

The UART Component is placed throughout the UDB array. The Component utilizes the following resources.

Configuration[4]	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
Full UART	3	58	3	3	–	2
Simple UART	3	21	3	1	–	0
Half Duplex	1	21	2	2	–	0
RX Only	1	12	2	1	–	0
TX Only	1	10	2	1	–	0

Note Disabling the TxBitClkGenDP parameter in the Expression View of the Advanced tab switches one Datapath Cell consumption to one Control Cell for the TX portion of the UART in Full UART and Simple UART configurations. These configurations will consume two Datapath Cells plus one Control Cell.

Note The UART Component also uses a different number of P-terms in different configurations. Please refer to build report for the exact number of used resources.

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted. Specifications are valid for 1.71 V to 5.5 V, except where noted.

DC Characteristics

Parameter	Description	Min	Typ[5]	Max	Units
$I_{DD(Full)}$	Component current consumption (Full UART)				
	Idle current[6]	–	520	–	µA/Mbps
	Operating current[7]	–	850	–	µA/Mbps
$I_{DD(Simple)}$	Component current consumption (Simple UART)				

4. Refer to Configuration Details section for selected parameters per each mode.
5. Device IO and clock distribution current not included. The values are at 25 °C.
6. Current consumed by Component while it is enabled but not transmitting/receiving data.
7. Current consumed by Component while it is enabled and transmitting/receiving data.

Parameter	Description	Min	Typ[5]	Max	Units
	Idle current[3]	–	130	–	µA/Mbps
	Operating current[4]	–	360	–	µA/Mbps
$I_{DD(HalfDuplex)}$	Component current consumption (Half Duplex)				
	Idle current[3]	–	100	–	µA/Mbps
	Operating current for receive operation[4]	–	140	–	µA/Mbps
	Operating current for transmit operation[4]	–	220	–	µA/Mbps
$I_{DD(RX)}$	Component current consumption (RX Only)				
	Idle current[3]	–	70	–	µA/Mbps
	Operating current[4]	–	100	–	µA/Mbps
$I_{DD(TX)}$	Component current consumption (TX Only)				
	Idle current[3]	–	50	–	µA/Mbps
	Operating current[4]	–	200	–	µA/Mbps

AC Characteristics

Parameter	Description		Min	Typ	Max[8]	Units
f_{CLOCK}	Component clock frequency [9]					
		Full UART	–	–	30	MHz
		Simple UART	–	–	46	MHz
		Half Duplex UART	–	–	51	MHz
		RX Only	–	–	62	MHz
		TX Only	–	–	57	MHz
t_{CLOCK}	Clock period		$1/f_{CLOCK}$	–	–	ns
f_b	Bit rate		–	–	f_{CLOCK}/ Oversampling	Mbps
T_{CLOCK}[10]	Clock tolerance					
		8x Oversampling	–	3.9	–	%

8. The values provide a maximum safe operating frequency of the Component. The Component may run at higher clock frequencies, at which point you will need to validate the timing requirements with STA results.

9. The maximum Component clock frequency depends on the selected mode and additional features.

10. Clock tolerance is showed for the UART configuration: 8 data bits, no parity, 1 stop bit, 2 out of 3 voting enabled. The value for other configuration can be calculated as described later in this datasheet.

Parameter	Description		Min	Typ	Max[8]	Units
		16x Oversampling	–	4.6	–	%
$\%_{ERR}$	Error		–	STA[11]	–	%
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		–	–	STA[12]	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}

Configuration Details

Full UART options:

Mode:	Full UART
Parity:	Even
API control enabled:	Enable
Flow Control:	Hardware (pins)
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 (Output pins)
Hardware TX:	Enable (Output pin)
Oversampling rate:	16x
Reset:	Input pin

11. $\%_{ERR}$ is present on the system when PSoC Creator cannot generate the exact frequency clock. The value must be calculated as described later in this datasheet.

12. t_{RTS_RX} value depends on the Static Timing Analysis results and must be calculated as described later in this datasheet.

Simple UART options:

Mode:	Full UART
Parity:	None
API control enabled:	Disable
Flow Control:	None
Address Mode:	None
RX Buffer Size (bytes)	4
TX Buffer Size (bytes)	4
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
TxBitClkGenDP	False (To switch go to Expression View of Advanced tab).

All other options same as the Simple UART

Figure 1. TX Mode Timing Diagram

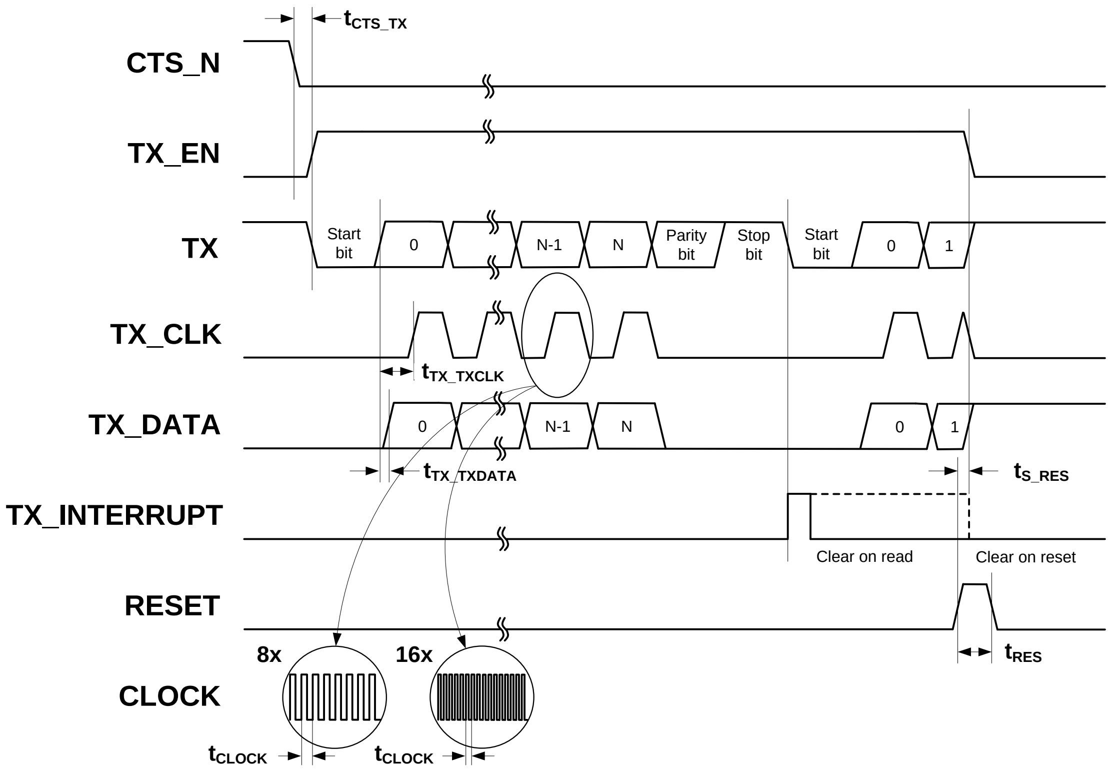
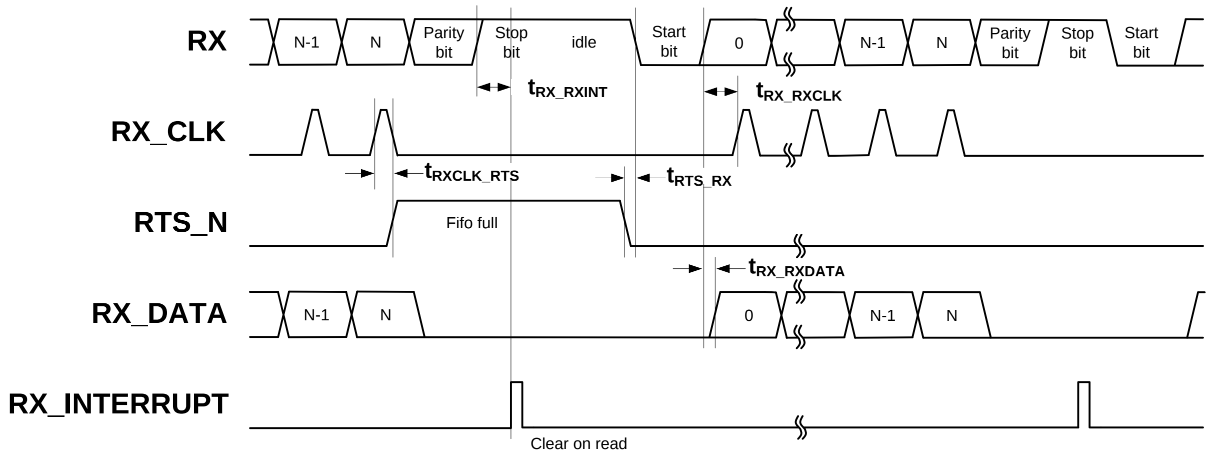


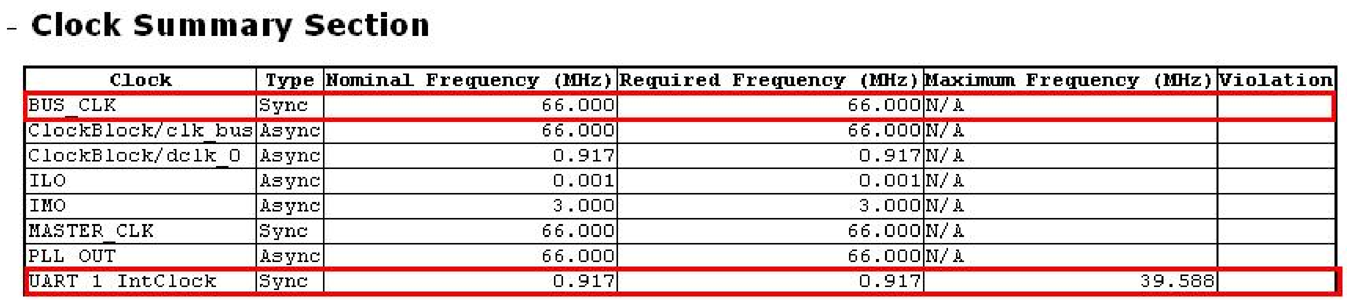
Figure 2. RX Mode Timing Diagram



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 with the STA results using 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. The following graphic shows an example of the internal clock limitations from the *_timing.html*.



- **Clock Summary Section**

Clock	Type	Nominal Frequency (MHz)	Required Frequency (MHz)	Maximum Frequency (MHz)	Violation
BUS_CLK	Sync	66.000	66.000	N/A	
ClockBlock/clk_bus	Async	66.000	66.000	N/A	
ClockBlock/dclk_0	Async	0.917	0.917	N/A	
ILO	Async	0.001	0.001	N/A	
IMO	Async	3.000	3.000	N/A	
MASTER_CLK	Sync	66.000	66.000	N/A	
PLL_OUT	Async	66.000	66.000	N/A	
UART_1_IntClock	Sync	0.917	0.917	39.588	

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 fourth 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 three clock cycles, and the midpoint of each subsequent bit to appear every eight clock cycles. If the UART clock has 0-percent error, the sampling happens exactly at the midpoint of the Stop bit. But, because the UART clock will not have zero error, the sampling happens earlier or later than the midpoint on every bit. This error keeps accumulating and results in the maximum error on the Stop bit. If you sample a bit one-half 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 usually equals 25 percent of the bit time for the normal signal quality. This value depends on RS-232 cable length, cable quality, and transceiver parameters. These factors are not taken into account in this analysis.

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 and this makes the clock tolerance asymmetrical. So, our error budget reduces to +3 and -4 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:

$$+3 \div 76 \times 100\% = +3.9\%, \ -4 \div 76 \times 100\% = -5.2\%.$$

The clock tolerance for 16x oversampling is:

$$+7 \div (9.5 \times 16) \times 100\% = +4.6\%$$

$$-8 \div (9.5 \times 16) \times 100\% = -5.2\%$$

The 2 out of 3 voting per bit feature makes the sampling wider on ±1 clock. For the faster clock this doesn’t affect on clock tolerance budget because voting algorithm will compensate one missed bit. For the slower clock case this feature affects the following byte receive procedure. This effect could be eliminated by using 2 stop bits feature, otherwise it makes clock tolerance requirements symmetrical and they become:

8x oversampling, voting enabled: ±3.9%

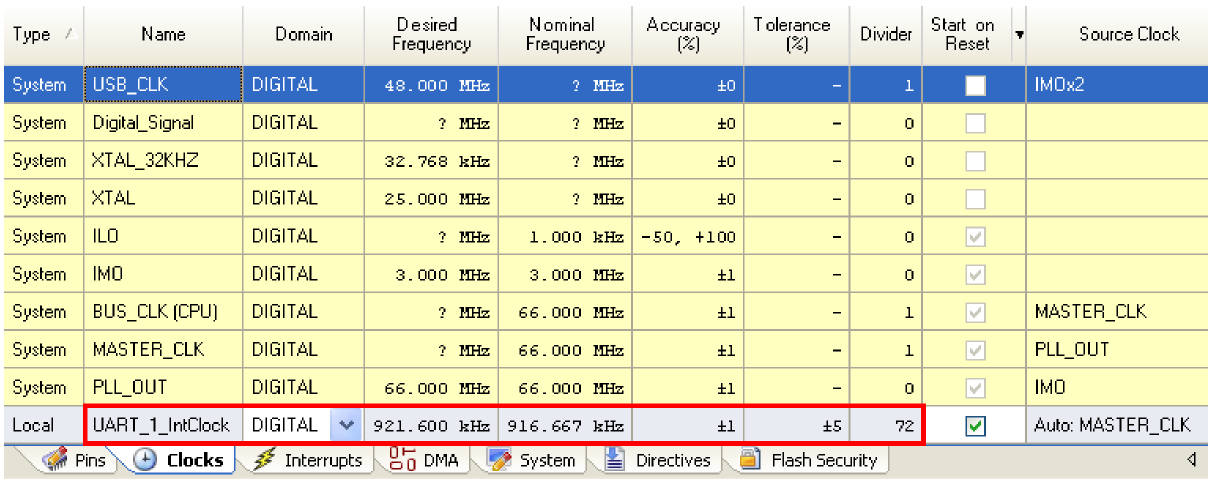
16x oversampling, voting enabled: ±4.6%

This total tolerance must 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 100-ppm crystal oscillator, the device on the other side can use almost the entire tolerance budget.

%ERR This error is present on the system when PSoC Creator cannot generate the exact frequency clock required by the UART because of the PLL clock frequency and divider value. You can see the difference in the design wide resources (DWR) as the desired and nominal frequency for the CharComp_clock. The error is calculated using the following equation:

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



Type	Name	Domain	Desired Frequency	Nominal Frequency	Accuracy (%)	Tolerance (%)	Divider	Start on Reset	Source Clock
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	25.000 MHz	? MHz	±0	-	0	☐	
System	ILO	DIGITAL	? MHz	1.000 kHz	-50, +100	-	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	UART_1_IntClock	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) ÷ 921.6 × 100 = ~0.5%

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

Clock accuracy depends on the selected IMO clock. It is equal to ±1% for the 3-MHz IMO. The total error is: 0.5 + 1 = 1.5% and it is less than the minimum clock tolerance for 8x oversampling.

Other IMO clock settings have larger accuracy error and are not recommended for use 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 the 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.

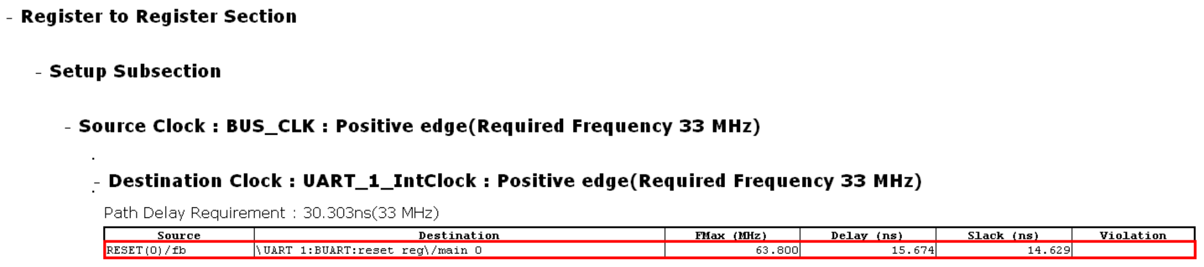
$$t_{TX\_TXCLK} = t_{CLOCK} * \left(\frac{Oversampling}{2} + 1 \right)$$

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.

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

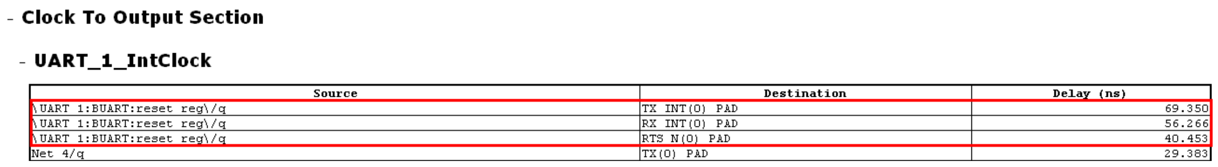
t_{S_RES} RESET activation time is the pin to internal register routing path delay time plus clock to output delay time. This is provided in the STA results as shown below:



- **Register to Register Section**
  - **Setup Subsection**
    - **Source Clock : BUS_CLK : Positive edge(Required Frequency 33 MHz)**
      - **Destination Clock : UART_1_IntClock : Positive edge(Required Frequency 33 MHz)**

Path Delay Requirement : 30.303ns(33 MHz)

Source	Destination	FMax (MHz)	Delay (ns)	Slack (ns)	Violation
RESET(0)/fb	\UART 1:BUART:reset reg\/main 0	63.800	15.674	14.629	



- **Clock To Output Section**
  - **UART_1_IntClock**

Source	Destination	Delay (ns)
\UART 1:BUART:reset reg\/q	TX INT(0) PAD	69.350
\UART 1:BUART:reset reg\/q	RX INT(0) PAD	56.266
\UART 1:BUART:reset reg\/q	RTS N(0) PAD	40.453
Net 4/q	TX(0) PAD	29.383

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.

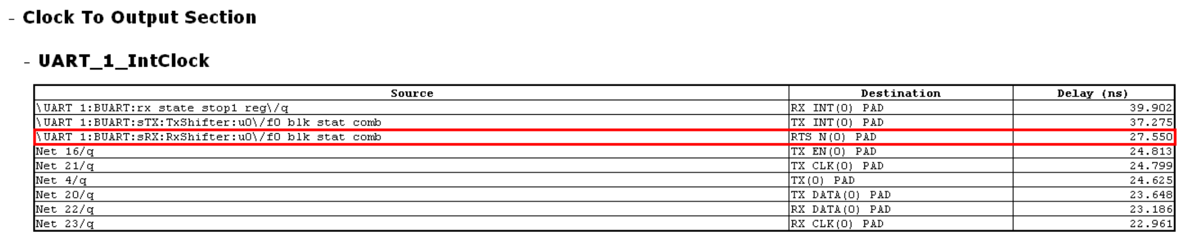
t_{RXCLK_RTS} Delay from the last RX_CLK raise to RTS_N active. This happens when the 4-byte FIFO is full. The RTS_N signal is automatically set by hardware as soon as input FIFO is full. The FIFO is loaded with one Component clock cycle delay from the last RX_CLK rising edge.

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 section as shown below.


- **Clock To Output Section**
  - **UART_1_IntClock**

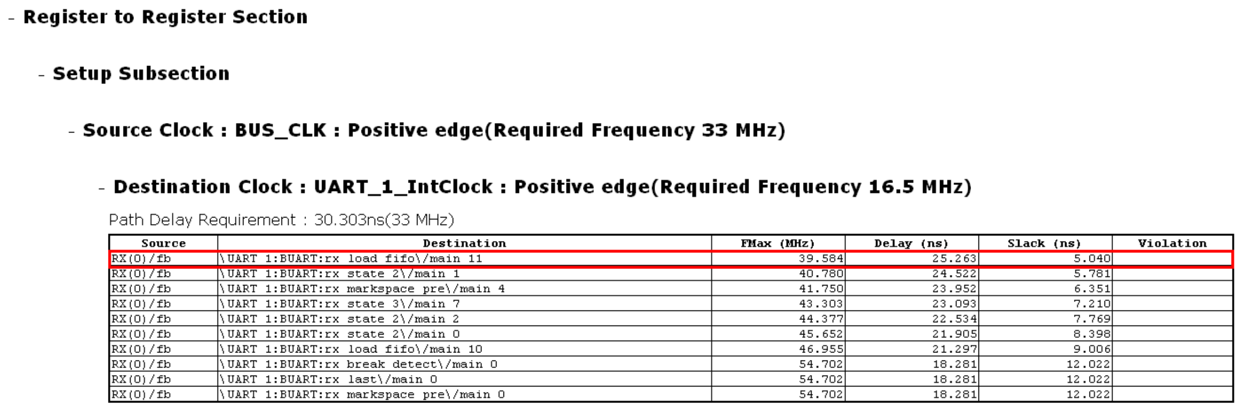
Source	Destination	Delay (ns)
\UART 1:BUART:rx state stop1 reg\/q	RX INT(0) PAD	39.902
\UART 1:BUART:sTX:TxShifter:u0\/f0 blk stat comb	TX INT(0) PAD	37.275
\UART 1:BUART:sRX:RxShifter:u0\/f0 blk stat comb	RTS N(0) PAD	27.550
Net 16/q	TX EN(0) PAD	24.813
Net 21/q	TX CLK(0) PAD	24.799
Net 4/q	TX(0) PAD	24.625
Net 20/q	TX DATA(0) PAD	23.648
Net 22/q	RX DATA(0) PAD	23.186
Net 23/q	RX CLK(0) PAD	22.961

RTS_{PD_PCB} is the PCB path delay from the 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 path delay time of the RX signal. This is provided in the STA results Register to Register section as shown below.


- **Register to Register Section**
  - **Setup Subsection**
    - **Source Clock : BUS_CLK : Positive edge(Required Frequency 33 MHz)**
      - **Destination Clock : UART_1_IntClock : Positive edge(Required Frequency 16.5 MHz)**

Path Delay Requirement : 30.303ns(33 MHz)

Source	Destination	FMax (MHz)	Delay (ns)	Slack (ns)	Violation
RX(0)/fb	\UART 1:BUART:rx load fifo\/main 11	39.584	25.263	5.040	
RX(0)/fb	\UART 1:BUART:rx state 2\/main 1	40.780	24.522	5.781	
RX(0)/fb	\UART 1:BUART:rx markspace pre\/main 4	41.750	23.952	6.351	
RX(0)/fb	\UART 1:BUART:rx state 3\/main 7	43.303	23.093	7.210	
RX(0)/fb	\UART 1:BUART:rx state 2\/main 2	44.377	22.534	7.769	
RX(0)/fb	\UART 1:BUART:rx state 2\/main 0	45.652	21.905	8.398	
RX(0)/fb	\UART 1:BUART:rx load fifo\/main 10	46.955	21.297	9.006	
RX(0)/fb	\UART 1:BUART:rx break detect\/main 0	54.702	18.281	12.022	
RX(0)/fb	\UART 1:BUART:rx last\/main 0	54.702	18.281	12.022	
RX(0)/fb	\UART 1:BUART:rx markspace pre\/main 0	54.702	18.281	12.022	

Component Errata

This section lists known problems with the Component.

Cypress ID	Component Version	Problem	Workaround
243067	All	When the RX Buffer Size is greater than 4, the UART_GetChar can read data directly from the RX FIFO and clear error condition statuses of the hardware. If such a case occurs, the error conditions are not detected in the RX interrupt handler.	Call UART_GetRxBufferSize to ensure that the RX buffer is not empty. Then call UART_GetChar to read data. This flow prevents missing the error detection, because the RX interrupt handler always services received data (puts data in the RX buffer and tracks errors).

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
2.50.d	Datasheet update.	Added errata item 243067. Added limitation for break length generation in the UART_SendBreak function. Updated description of UART_ReadControlRegister and UART_WriteControlRegister functions.
2.50.c	Datasheet update.	Fixed broken link. Added a note that the Component uses a different number of P-terms in different configurations.
2.50.b	Datasheet update.	Added Macro Callbacks section. Added PSoC 4 Glitch Avoidance at System Reset section.
2.50.a	Datasheet edit.	Added Additional Reading section with links to related Application Notes.
2.50	Fixed data transmission issue in Half duplex mode.	The wrong data were sent by UART v2.40 when multiple characters sent at once.
	Refactored internal RX interrupt ISR code to read RX status register in one place.	Previously, the RX clear-on-read error status could be lost because of read of RX status register in multiple places.
	Updated the datasheet.	Updated the numbers in the API Memory and Resources sections. Updated MISRA Compliance section with Component specific deviations.

Version	Description of Changes	Reason for Changes / Impact
2.40	Added software TX interrupt triggering to UART_PutChar() API.	The transmission from TX software buffer stuck in the dead loop when UART_PutChar() was interrupted for time greater than transmission of full TX FIFO.
	Fixed the duration of the Stop bit transmission.	The Stop bit duration had the one clock cycle overhead for one by one data transmission.
	Changed the definition from: UART_TXBUFFERSIZE UART_RXBUFFERSIZE UART_RXHWADDRESS2 UART_RXHWADDRESS1 to UART_TX_BUFFER_SIZE UART_RX_BUFFER_SIZE UART_RX_HW_ADDRESS2 UART_RX_HW_ADDRESS1	Add underscores between separate words to improve readability of definitions. The Component supports both definitions, but the old definitions are going to become obsolete.
2.30.d	Edited the datasheet.	Updated TX and RX data register addresses. Updated the Resources section. Updated the printf section. Update the Internal RX Interrupt section. Rearranged a few sections to comply with the template.
2.30.c	Edited datasheet to add Component Errata section. Added section to explain printf function usage model for the UART.	Document that the Component was changed, but there is no impact to designs. Datasheet was lacking this printf explanation.
2.30.b	Edited datasheet to remove PSoC 5 reference.	PSoC 5 replaced by PSoC 5LP.
2.30.a	Updated datasheet with memory usage for PSoC 4.	
2.30	Fixed HalfDuplex mode.	The receiver didn't work correctly in HalfDuplex mode when 16x oversampling rate selected.
	Added MISRA Compliance section.	The Component does not have any specific deviations.
	Integrated specific APIs to support the bootloader: CyBtldrCommStart, CyBtldrCommStop, CyBtldrCommReset, CyBtldrCommWrite, CyBtldrCommRead.	UART could be used as a communication Component for the Bootloader with this feature.

Version	Description of Changes	Reason for Changes / Impact
2.20	Changed sampling time with and without Voting option. Now RX line is sampled on fourth clock edge , and on 3th to 5th with majority voting enabled. Refer to section “2 out of 3 Voting” for more details.	This change gives symmetrical clock tolerance.
	Added PSoC 5LP support.	
2.10	Changed the parameter type for UART_PutString() API from uint8* to char*.	The common usage of this API is with an embedded string as a parameter: UART_PutString(“Hello World”). The “char“ type should be used for this usage without compiler warnings.
	UART_ClearRxBuffer()/ UART_ClearTxBuffer() APIs clears hardware FIFO too.	Hardware FIFO needs to be cleared to guarantee that no more data is pending for reception/transmission.
	Fixed typo with the parameter for UART_SendBreak() API. UART_WAIT_FOR_COMLETE_REINT changed to UART_WAIT_FOR_COMPLETE_REINT.	Typo fix.
	Added all UART APIs with CYREENTRANT keyword when they included in .cyre file.	Not all APIs are truly reentrant. Comments in the Component API source files indicate which functions are candidates. This change is required to eliminate compiler warnings for functions that are not reentrant used in a safe way: protected from concurrent calls by flags or Critical Sections.
	Updated Address Mode functionality.	These modes are upgraded for automatically skip unaddressed packets.
	Fixed Hardware Flow control mode when internal RX buffer is used. The code in the RX ISR stops to read data from the FIFO when internal buffer overruns. As a result, the RTS signal holds the transmitter UART.	The data was read from the hardware FIFO and moved to the s/w buffer regardless of whether the s/w buffer has overrun.
	Updated internal clock Component with cy_clock_v1_60.	Clock v1_60 is the latest Component version.
	Limited RX and TX Buffer Size minimum value to 4.	UART always uses 4 bytes FIFO as a buffer.
	Minor datasheet edits and updates	
2.0.a	Minor datasheet edits and updates	
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.

Version	Description of Changes	Reason for Changes / Impact
	Reset input registered.	Registering improves maximum baud rate when Reset input is used.
	Added characterization data to datasheet	
	Minor datasheet edits and updates	
1.50	Added Sleep/Wakeup and Init/Enable APIs.	To support low-power modes, as well as to provide common interfaces to separate control of initialization and enabling of most Components.
	Break signal has length selection (11 to 14 bits) and added parameter to SendBreak function.	Break signal length for UART is not specified, therefore 11 to 14 bits selection is provided.
	Added 16x oversampling mode.	16x oversample mode reduces jitter effect on error at higher speeds.
	Software option removed from **Parity Type** selection, **API control enabled** check box has been added instead.	This allowed a way to select a default value when needed parity API control. If updating from version 1.20 of the UART Component with this option selected, it is recommended to select the "None" parity option in version 1.50.