8-Bit UART Datasheet UART V 5.3

Copyright © 2001-2015 Cypress Semiconductor Corporation. All Rights Reserved.

Resources		PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
		Digital	Analog CT	Analog SC	Flash	RAM	
CY8C29/27/24/22/21xxx, CY8C23x33, CY7C64215, CYWUSB6953, CY8CLED02/04/08/16, CY8CLED0xD, CY8CLED0xG, CY8CTST110, CY8CTMG110, CY8CTST120, CY8CTMG120, CY8CTMA120, CY8CTMA140, CY8C21x45, CY8C22x45, CY8CTMA30xx, CY8C28x45, CY8CPLC20, CY8CLED16P01, CY8C28x43, CY8C28x52, CY7C603XX							
CY8C29/27xxx, CY8CLED08/16, CY8CPLC20, CY8C28x43, CY8C28x45, CY8C28x52	Low Level API	2	–	–	311	0	2
	High Level API	2	–	–	507	3 + Buffer size (default - 16)	2
CY8C24/22/21xxx, CY8C23x33, CY7C64215, CYWUSB6953, CY8CLED02/04, CY8CLED0xD, CY8CLED0xG, CY8CTST110, CY8CTMG110, CY8CTST120, CY8CTMG120, CY8CTMA120, CY8CTMA140, CY8C21x45, CY8C22x45, CY8CTMA30xx, CY8CPLC20, CY8CLED16P01	Low Level API	2	–	–	339	0	2
	High Level API	2	–	–	608	3 + Buffer size (default - 16)	2

For one or more fully configured, functional example projects that use this user module go to www.cypress.com/psocexampleprojects.

Introduction

Universal Asynchronous Receiver/Transmitter (UART) is a standard serial communication protocol for exchanging data between two devices. In this communication protocol, data is transferred sequentially, one bit at a time. This implementation uses a frame consisting of 8 data bits, one start bit, one optional parity bit, and one or more stop bits. A parity bit is a bit, with a value of 0 or 1, that is added to a block of data for error detection purposes. This bit is optional; it may or may not be added to the data payload. It can also be set either to odd or even. These bits are often used in data transmission to ensure that data is not corrupted during the transfer process. If the data transmission protocol is set to an odd parity, each data packet must have an odd parity. If it is set to even, each packet must have an even parity. If a packet is received with the wrong parity, an error will be produced and the data will need to be retransmitted. The parity bit for each data packet is computed before the data is transmitted.

The UART data frame is shown in Figure 1.

Figure 1. UART Data Frame

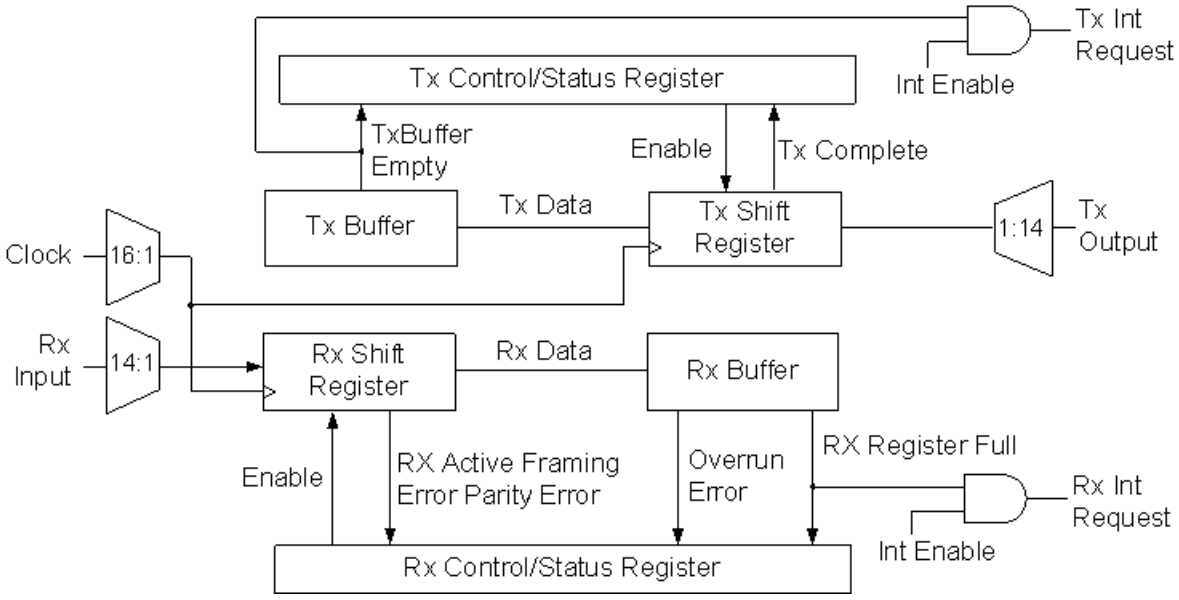
Because the UART protocol is asynchronous, it does not need a clock signal. In UART communication, speed is defined by the baud rate. The baud rate is equal to the number of bits transmitted per second including the start and stop bits. For reliable data transmission and reception without any loss of bits, both the transmitter and receiver should have same baud rate. Mismatch in baud rates typically results in framing error. Common baud rates are 4800, 9600, 19200, 38400, 57600, and 115200 but other rates may also be used.

Features and Overview

- Asynchronous receiver and transmitter
- Data-format compliant with RS-232 serial-data format
- Burst rates up to 6 Mbits/second
- Data framing consists of start, optional parity, and stop bits
- Optional interrupt on receive register full and transmit buffer empty
- Parity, overrun, and framing error detection
- High level transmit and receive functions

The UART User Module is an 8-bit Universal Asynchronous Receiver Transmitter that supports duplex RS-232-compliant, data format serial communications over two wires. Received and transmitted data format includes a start bit, optional parity, and a stop bit. Programmable clocking and selectable interrupt or polling style operation is supported. Application Programming Interface (API) firmware routines are given to initialize, configure, and operate the UART. An additional high level API is also given to support background command receiving and string printing.

Figure 2. UART Block Diagram



Functional Description

The UART User Module implements a serial transmitter and receiver. The UART maps onto two PSoC Digital Communication blocks designated TX and RX, in the Device Editor of PSoC Designer™. The TX PSoC block gives transmitter functionality and the RX PSoC block gives receiver functionality.

RX and TX operate independently. Each has their own Control and Status register, programmable interrupts, I/O, Buffer register, and Shift register. They share the same enable, clock, and data format.

Setting the Enable bit in the RX Control and TX Control registers enables the UART for operation. Enabling and disabling is performed using the API provided functions.

The UART User Module clock is shared by both the RX and TX components. The clock frequency selected must be eight times the frequency of the required data bit rate. Each received or transmitted data bit requires eight input clock cycles. The clock is configured using the PSoC Designer Device Editor.

The data received and transmitted is a bit stream that consists of a start bit, eight data bits, an optional parity bit, and a stop bit. The parity may be set to none, even, or odd, and is set using the PSoC Designer Device Editor or using the UART API. Both RX and TX are set to the same parity configuration.

TX - UART Transmitter

The transmitter uses the TX Buffer, TX Shift, and TX Control registers of a Digital Communications type PSoC block.

The TX Control register is initialized and configured using the UART User Module firmware API routines. When the Enable bit in the TX Control register is set, an internal divide-by-eight bit clock is generated.

A data byte to transmit is written by an API routine into the TX Buffer register, clearing the Tx Buffer Empty status bit in the TX Control register. This status bit can be used to detect and prevent transmit overrun errors.

The rising edge of the next bit clock transfers the data to the Shift register and sets the Tx Buffer Empty bit in the TX Control register. If the interrupt enable mask is enabled, an interrupt is triggered. This interrupt enables queuing the next byte to transmit. So when the current data byte is completely transmitted, the new byte is transmitted on the next available transmit clock.

The start bit is transmitted at the same time that the data byte is transferred from the TX Buffer register to the TX Shift register. Successive bit clocks shift a serial bit stream to the output. The stream is composed of each bit of the data byte, least significant bit first, an optional parity bit, and a final stop bit. When the stop bit is completely transmitted, the TX Control register's Tx Complete status bit is set. This bit remains valid until read. If a new data byte has been written to the TX Buffer register, the data byte is transferred to the TX Shift register and transmission of the data begins on the next rising edge of the bit clock.

RX - UART Receiver

The receiver uses the RX Buffer, RX Shift, and RX Control registers of a Digital Communications type PSoC block.

The RX Control register is initialized and configured using the UART User Module firmware API routines. Initialization of the RX consists of setting the UART parity, optionally enabling the interrupt on the Rx Register Full condition, and then enabling the UART.

When a start bit is detected on the RX input, a divide-by-eight bit clock is started and synchronized to sample the data in the center of the received bits. On the rising edge of the next eight-bit clock, the input data is sampled and shifted into the RX Shift register. If parity is enabled, the next bit clock samples the parity bit. The sampling of the stop bit, on the next clock, results in the received data byte transfer to the RX Buffer register and the triggering of one or more of the following events:

- Rx Register Full bit in the RX Control register is set, and if the interrupt for the RX is enabled, then the associated interrupt is triggered.
- If the stop bit is not detected at the expected bit position in the data stream, then the Framing Error bit in the RX Control register is set.
- If the Buffer register has not been read before the stop bit of the currently received data, then the Over-run Error bit in the RX Control register is set.

- If a parity error was detected, then the Parity Error bit is set in the RX Control register.

For polling detection of a completely received data byte, the Rx Register Full bit in the RX Control register must be monitored. Data must be read out of the RX Buffer register before the next byte is completely received, to prevent the overrun error condition.

Communication System Accuracy

The maximum deviation allowed in the clock source is ±4% for reliable UART communication. The IMO of the PSoC 1 has a maximum tolerance of 2.5% and hence can be used. But the 6MHz SLIMO clock cannot be used since it has a tolerance of ±4.2%, which is not acceptable for a reliable UART communication.

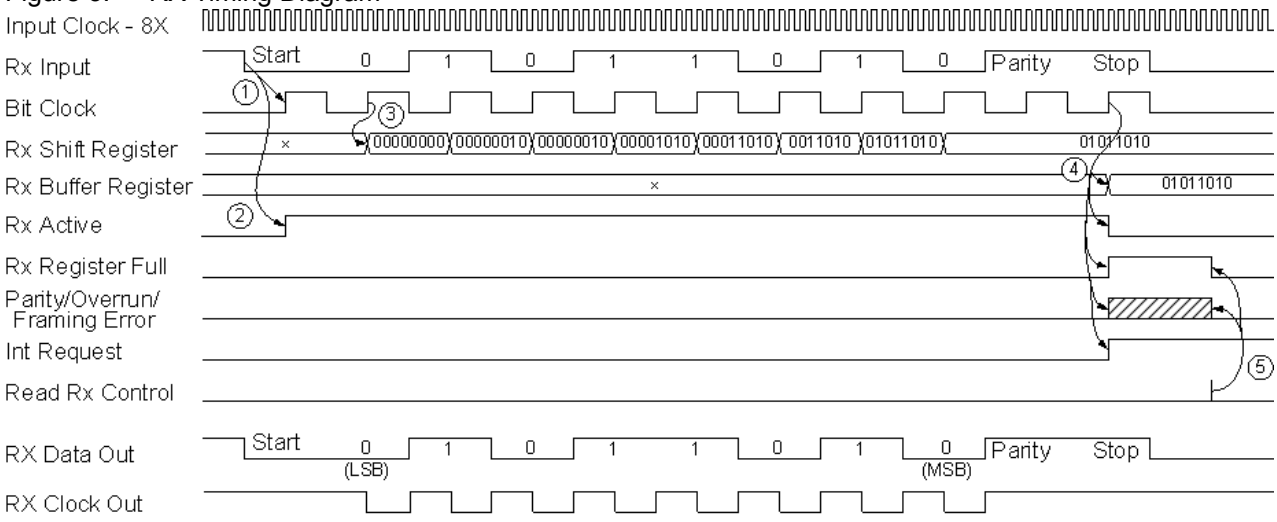
Some of the PSoC 1 devices such as CY24x94 and CY21x34 have IMO with a tolerance of >=4%. When connected to the USB bus, the IMO gets synced to the USB clock and hence will become as accurate as the USB bus clock. When not connected to USB bus, it runs at ±4% tolerance. In these devices, when you place a UART User Module, the Design Rule Checker gives a warning. For example, when you place the UART UM in CY8C24x94, the warning generated is “UART should not be used in the CY8C24x94 devices without connection to the USB bus”.

The system error, the sum of the error at both ends of the communication link should be less than 4% for the UART communication to work properly. See the device datasheets for more information about the accuracy of SysClk.

Timing

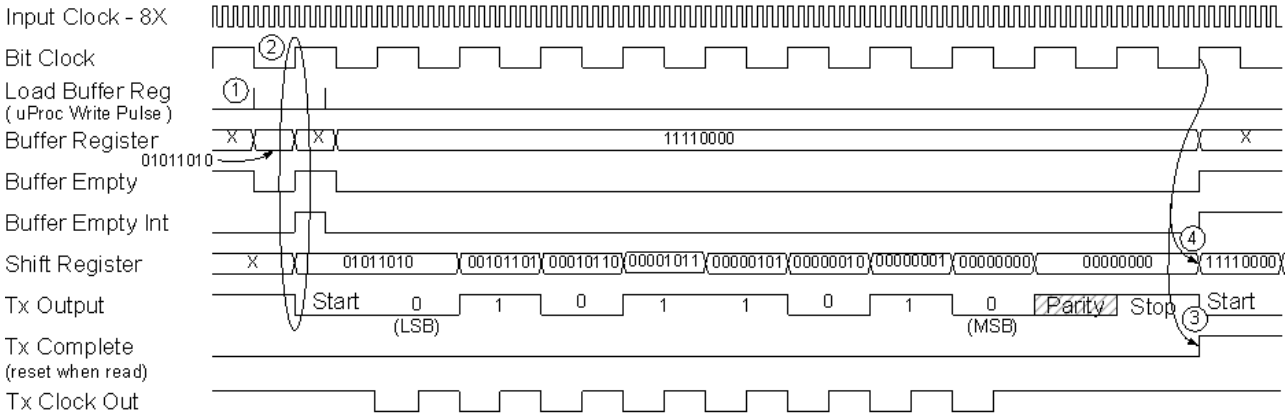
The following RX Timing diagram illustrates the operation of the RX component of the UART User Module:

Figure 3. RX Timing Diagram



The following TX Timing diagram illustrates the operation of the TX component of the UART User Module:

Figure 4. TX Timing Diagram



DC and AC Electrical Characteristics

Table 1. UART DC and AC Electrical Characteristics

Parameter	Conditions and Notes	Typical	Limit	Units
F_{max}	Maximum character transmission frequency	--	6	Mbits

Placement

The UART User Module can be placed only in the Digital Communications blocks. It requires two blocks. Note that the same clock source is used for both receiver and transmitter components.

Parameters and Resources

Clock

UART is clocked by one of 16 possible sources. The Global I/O buses can be used to connect the clock input to an external pin or a clock function generated by a different PSoC block. When using an external digital clock for the block, the row input synchronization must be turned off for accuracy and sleep operation. The 48 MHz clock, the CPU_32 kHz clock, one of the divided clocks, 24V1 or 24V2, or another PSoC block output can be specified as the clock input.

The clock rate must be set to eight times the desired bit rate. One data bit is received or transmitted every eight clock cycles. The tolerance of clock must be ± 2% for it to work properly.

Some examples here show how you can set the different baud rate.

If the desired baud rate is 9600, the clock to the UART should be 8 x 9600 = 76.8 KHz. To get a frequency of 76.8 KHz from 24 MHz, the required divider is 312.5. Unfortunately, we cannot have a fraction in the divider and have to round off to 312 or 313. Following are some of the options to generate this divider:

– SysClk = **24 MHz**, VC1 divider = **8**, VC3 source = **VC1**, and VC3 divider = **39**

– SysClk = **24 MHz**, VC1 divider = **2**, VC2 divider = **4**, VC3 source = **VC2**, and VC3 divider = **39**

– SysClk = **24 MHz**, VC1 divider = **2**, VC3 source = **VC1**, and VC3 divider = **156**

In all the above cases, the Clock parameter is set to VC3.SysClk = 24 MHz, VC1 divider = 4, VC3 Source = VC1 and VC3 divider = 39. SysClk = 24 MHz, VC1 divider = 2, VC2 divider = 2, VC3 Source

= VC2 and VC3 divider = 39.SysClk = 24 MHz, VC3 Source = SysClk/1 and VC3 divider = 156.In all the above cases the Clock parameter is set to VC3.

– SysClk = **24 MHz**, VC1 divider = **2**, VC2 divider = **2**, VC3 source = **VC2**, and VC3 divider = **39**

– SysClk = **24 MHz**, VC1 divider = **2**, VC2 divider = **2**, VC3 source = **VC2**, and VC3 divider = **39**

– SysClk = **24 MHz**, VC3 source = **SysClk/1**, and VC3 divider = **156**

In all the above cases the Clock parameter is set to VC3.

RX Input

The input of the receiver can be connected to a low, high, neighboring PSoC block, the analog comparator output bus, or one of the global buses. Using a global bus, the input can be connected to one of the external pins.

Invert RX Input

This parameter enables you to invert the RX input signal. This option may be used for certain RS232 transceivers that produce an inverted output.

TX Output

The output of the transmitter can be routed to the Global Output Bus. The Global Output Bus can then be connected to an external pin or to another PSoC block for further processing.

TX Interrupt Mode

This option determines when an interrupt is generated for the TX block. The “TxRegEmpty" option causes an interrupt to be generated as soon as the data is transferred from the Data register to the Shift register. Choosing the second option, “TxComplete," delays the interrupt until the last bit is shifted out the Shift register. This second option is useful when it is important to know that the character has been completely sent. The first option, “TxRegEmpty," is best used to maximize the output of the transmitter. It allows a byte to be loaded while the previous byte is being sent. In the interrupt service routine, the TX_CONTROL_REG should be read to enable subsequent interrupts.

ClockSync

In PSoC devices, digital blocks may provide clock sources in addition to the system clocks. Digital clock sources may even be chained in a ripple fashion. This introduces skew with respect to the system clocks. These skews are more critical in the CY8C29/27/24/22/21xxx and CY8CLED04/08/16 PSoC device families because of various data-path optimizations, particularly those applied to the system buses. This parameter may be used to control clock skew and ensure proper operation when reading and writing PSoC block register values. Appropriate values for this parameter should be determined from the following table:

ClockSync Value	Use
Sync to SysClk	Use this setting for any 24 MHz (SysClk) derived clock source that is divided by two or more. Examples include VC1, VC2, VC3 (when VC3 is driven by SysClk), 32 kHz, and digital PSoC blocks with SysClk-based sources. Externally generated clock sources should also use this value to ensure that proper synchronization occurs.
Sync to SysClk*2	Use this setting for any 48 MHz (SysClk*2) based clock unless the resulting frequency is 48 MHz (in other words, when the product of all divisors is 1).
Use SysClk Direct	Use this setting when a 24 MHz (SysClk/1) clock is required. This does not actually perform synchronization but provides low-skew access to the system clock itself. If selected, this option overrides the setting of the Clock parameter. It should always be used instead of VC1, VC2, VC3, or Digital Blocks where the net result of all dividers in combination produces a 24 MHz output.
Unsynchronized	Use this setting when the 48 MHz (SysClk*2) input is selected. Use when unsynchronized inputs are required. In general, this use is advisable only when interrupt generation is the only application of the Counter.

RX Output

This parameter enables you to route the Input signal to one of the row buses. This signal along with the Data Clock Out can be used to facilitate data verification functions such as Cyclical Redundancy Checks using the CRC16 User Module.

RX Clock Out

This parameter enables you to route the bit clock from the RX block to one of the row buses. The bit clock is the Clock input divided by eight. The rising edge of the Data Clock Out signal corresponds to the time when the data is stable and should be sampled. The signal along with the RX Output can be used to facilitate data verification functions such as Cyclical Redundancy Checks using the CRC16 User Module.

TX Clock Out

This parameter enables you to route the bit clock from the TX block to one of the row buses. The bit clock is the Clock input divided by eight. The rising edge of the Data Clock Out signal corresponds to the time when the data is stable and should be sampled. The signal along with the TX Output can be used to facilitate data verification functions such as Cyclical Redundancy Checks.

RxCmdBuffer

This parameter enables the receive command buffer and firmware used for command processing. The UART RX interrupt must be enabled for the command buffer to operate. The command buffer works in the UART RX interrupt service routine to collect the characters in a buffer until the command terminator character is received. Once the terminator character is received, a flag is set to signal that the command buffer is ready to be read.

RxBufferSize

This parameter determines the number of RAM locations that are reserved for the receive buffer. The largest command that can be received is one less than the buffer size selected, because the string must be null terminated. This parameter is only valid when the RxCmdBuffer is enabled and the UART RX interrupt is enabled.

CommandTerminator

This parameter selects the character that signals the end of a command. When received, a flag is set signaling that a complete command has been received. After this flag is set, additional characters are not accepted until the cmdReset() function is called.

Param_Delimiter

This parameter selects the character used to delimit the command and parameters in the command receiver buffer. For example, if the Param_Delimiter is set to a space character (32), each substring separated by a space is a parameter. Given the string ‘cmd foo bar c‘, the parameters are ‘cmd‘, ‘foo‘, ‘bar‘, and ‘c‘. Each call of szGetParam() returns a pointer to the next substring in the order from left to right as a null terminated string.

IgnoreCharsBelow

This parameter enables the receive buffer to ignore characters below a set value. The characters are received, but are not added to the receive buffer. This parameter is only valid when the RxCmdBuffer is enabled and the UART RX interrupt is active.

Enable_BackSpace

This parameter enables you to use a backspace or delete character to delete the last character in the UART receive buffer. The three possible settings are “Disable", “Backspace", and “Delete". This parameter is only valid when the RxCmdBuffer is enabled and the UART RX interrupt is active.

Interrupt Generation Control

The following parameters, InterruptAPI and IntDispatchMode, are only accessible by setting the **Enable interrupt generation control** check box in PSoC Designer. This is available under **Project > Settings > Chip Editor**.

InterruptAPI

The InterruptAPI parameter allows conditional generation of a user module’s interrupt handler and interrupt vector table entry. Select “Enable" to generate the interrupt handler and interrupt vector table entry. Select “Disable" to bypass the generation of the interrupt handler and interrupt vector table entry. If the Receive Command Buffer is used, then the InterruptAPI parameter must be set to “Enable". It is recommended that you carefully consider before choosing Interrupt API generation, particularly with projects that have multiple overlays where a single block resource is used by the different overlays. By selecting Interrupt API generation only when it is necessary, the need to generate an interrupt dispatch code can be eliminated, thereby reducing overhead.

IntDispatchMode

The IntDispatchMode parameter is used to specify how an interrupt request is handled for interrupts shared by multiple user modules existing in the same block but in different overlays. When “ActiveStatus" is selected, the firmware tests which overlay is active before servicing the shared interrupt request. This test occurs every time the shared interrupt is requested. This adds latency and also produces a nondeterministic procedure of servicing shared interrupt requests, but does not require any RAM. If “OffsetPreCalc" is selected, the firmware calculates the source of a shared interrupt request only when an overlay is initially loaded. This calculation decreases the interrupt latency and produces a deterministic procedure for servicing shared interrupt requests, but at the expense of a byte of RAM.

Application Programming Interface

The Application Programming Interface (API) routines are given as part of the user module to enable you to deal with the module at a higher level. This section specifies the interface to each function together with related constants provided by the “include" files.

Note

In this, as in all user module APIs, the values of the A and X register may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This “registers are volatile" policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must ensure their code observes the policy, too. Though some user module API function may leave A and X unchanged, there is no guarantee they may do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

The following table lists the low level UART supplied API functions:

Table 2. Low Level UART API

Function	Description
void UART_Start(BYTE bParity)	Enable user module and set parity.
void UART_Stop(void)	Disable user module.
void UART_EnableInt(void)	Enable both RX and TX interrupts.
void UART_DisableInt(void)	Disable both RX and TX interrupts.
void UART_SetTxIntMode(BYTE bTxIntMode)	Set the source of the Tx interrupt.
void UART_SendData(BYTE bTxData)	Send byte without checking TX status.
BYTE UART_bReadTxStatus(void)	Return status of TX Status register.
BYTE UART_bReadRxData(void)	Return data in RX Data register without checking status of character is valid.
BYTE UART_bReadRxStatus(void)	Check status of RX Status register.

UART_Start

Description:

Sets the parity and enables the UART receiver and transmitter. When enabled, data can be received and transmitted.

C Prototype:

```
void  UART_Start(BYTE bParitySetting)
```

Assembly:

```
mov   A, UART_PARITY_NONE
lcall  UART_Start
```

Parameters:

bParitySetting: One byte that specifies the transmit parity. Symbolic names are given in C and assembly, and their associated values are listed in the following table:

TX Parity	Value
UART_PARITY_NONE	0x00
UART_PARITY_EVEN	0x02
UART_PARITY_ODD	0x06

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_Stop

Description:

Disables the UART module.

C Prototype:

```
void  UART_Stop(void)
```

Assembly:

```
lcall  UART_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_EnableInt

Description:

Selects and enables both UART interrupt(s), by setting the appropriate interrupt enable bits.

C Prototype:

```
void  UART_EnableInt(void)
```

Assembly:

```
lcall  UART_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

If an interrupt is pending and this API is called, the interrupt is triggered immediately. This call must be made before calling Start(). The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_DisableInt

Description:

Disables all UART interrupts.

C Prototype:

```
void  UART_DisableInt(void)
```

Assembly:

```
lcall  UART_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_SetTxIntMode

Description:

Sets the source of the Tx interrupt.

C Prototype:

```
void  UART_SetTxIntMode(BYTE bTxIntMode)
```

Assembly:

```
lcall  UART_SetTxIntMode
```

Parameters:

bTxIntMode: One byte that specifies the interrupt mode of the Tx block. Symbolic names are given in C and assembly, and their associated values are listed in the following table:

TX Interrupt Mode	Value
UART_INT_MODE_TX_REG_EMPTY	0x00
UART_INT_MODE_TX_COMPLETE	0x01

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_SendData

Description:

Initiates data transmission by loading the TX Buffer register with specified data to transmit. The TX Complete status bit in the TX Control register must be monitored to ensure transmission was initiated.

C Prototype:

```
void  UART_SendData(BYTE bTxData)
```

Assembly:

```
mov   A, bTxData
lcall  UART_SendData
```

Parameters:

bTxData: Data to be loaded into the TX Buffer register and passed in the Accumulator.

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_bReadTxStatus

Description:

Returns the contents of the TX Control register.

C Prototype:

```
BYTE  UART_bReadTxStatus(void)
```

Assembly:

```
lcall  UART_bReadTxStatus
and   A, UART_TX_COMPLETE
jnz   TxIsComplete
```

Parameters:

None

Return Value:

Returns TX status byte. Use defined masks to test for specific status conditions. Note that masks can be OR'ed together to check for multiple conditions.

RX Status Masks	Value
UART_TX_COMPLETE	0x20
UART_TX_BUFFER_EMPTY	0x10

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_bReadRxData

Description:

Reads received data byte from the RX Buffer register.

C Prototype:

```
BYTE  UART_bReadRxData(void)
```

Assembly:

```
lcall  UART_bReadRxData
mov   [bRxData],A
```

Parameters:

None

Return Value:

Data byte received and passed in the Accumulator.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_bReadRxStatus

Description:

Reads and returns the RX Control register.

C Prototype:

```
BYTE  UART_bReadRxStatus(void)
```

Assembly:

```
lcall  UART_bReadRxStatus
push   A
and    A, UART_RX_COMPLETE
pop    A
jz     RxNotCompleted
cmp    A, UART_RX_ERROR              ; determine if RX was without error
jz     RxCompleteWithoutError        ; receive was completed
```

Parameters:

None

Return Value:

Returns RX status byte. Use the following defined masks to test for specific status conditions. Note that masks can be OR'ed together to check for combined conditions.

RX Status Masks	Value	Description (when bit is set)
UART_RX_REG_FULL	0x08	Data register contains unread data
UART_RX_PARITY_ERROR	0x80	Parity error since last time status read
UART_RX_OVERRUN_ERROR	0x40	Data has been over written to buffer register
UART_RX_FRAMING_ERROR	0x20	Stop bit is low
UART_RX_ERROR	0xE0	All Error bits OR'ed together

Side Effects:

A read of this register clears all of the status bits except for UART_RX_REG_FULL, which is only cleared when the data register is read directly or with the API functions. Ensure that all applicable status conditions are checked before discarding the return value. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

High Level API

High level APIs add additional firmware on top of the basic functions, to give command and string level functionality. The Device Editor allows you to set the size of the receive command buffer, command terminator, parameter delimiter, and below what value the receiver should ignore characters. Not all of the high level API functions require enabling this buffer. Transmit functions accept pointers to strings, so that an entire string may be printed with a single function call instead of additional lines of C or ASM code. The transmit functions do not require the UART transmit interrupt to be enabled.

To use the high level receiver functions that require the receive buffer, go to the Device Editor window, select the UART and select the "Enable" option for the "RxCmdBuffer" parameter. Next, select a "RxBufferSize" that is large enough to hold your largest command plus one. Select a command terminator character "CommandTerminator." This is most often set to be either a carriage return (13), or a line feed (10). If your commands contain two or more parameters, select a parameter delimiter "Param_Delimiter." Common command delimiters are usually a space (32) or a comma (44) character. Control characters below a selected value may also be ignored. Most control characters are in the range between 0 and 31. Set "IgnoreCharsBelow" to 32 to ignore these characters. Set this parameter to '1' if all characters are valid. The character selected for the command terminator (CommandTerminator) is not affected by the "IgnoreCharsBelow" option. The following flow chart shows the proper sequence for basic operation of the command buffer functions.

The command buffer works in the UART RX interrupt service routine to collect the characters in a buffer until the command terminator character is received. At that time, a flag is set to signal that the command buffer is ready to be read. The receive buffer may be accessed directly by reading the array INSTANCE_NAME_aRxBuffer or by using either szGetParam() or szGetRestOfParams() functions. If more characters are received than the buffer_size -1, the subsequent characters are ignored.

The command buffer collects characters until the buffer is full or a command terminator is detected. Any characters received after either of these two conditions is ignored, until the CmdReset command is executed. After the CmdReset command is executed, the RX ISR firmware begins to collect characters once again.

Figure 5. Command Buffer Flow

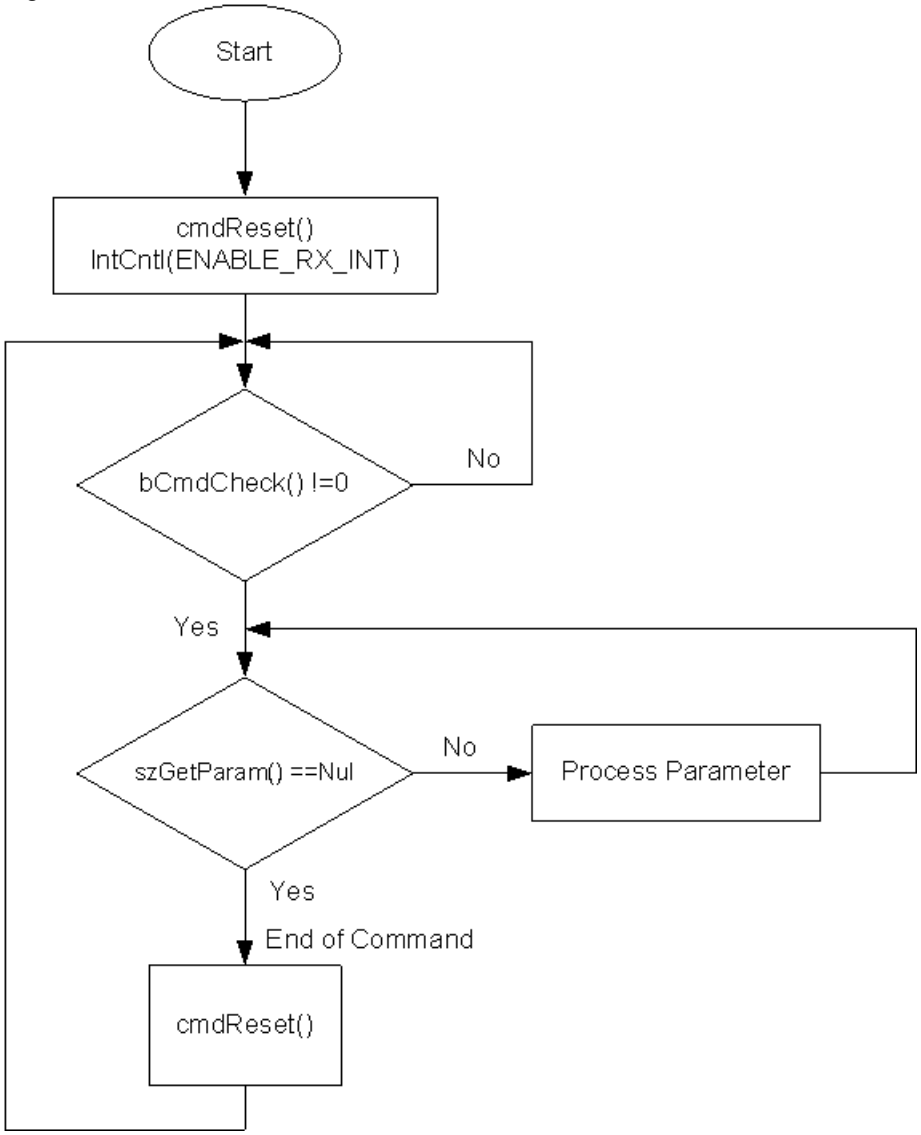


Table 3. High Level UART API

Function	Description
void UART_IntCntl(BYTE bMask)	Selectively enable/disable RX and TX interrupts.
void UART_PutString(char * szStr)	Send NULL terminated string out TX port.
void UART_CPutString(const char * azStr)	Send NULL terminated constant (ROM) string out TX port.
void UART_PutChar(char bData)	Send character to TX port when TX register is empty. Function does not return until TX Data register can be written to without a data overrun error.
char UART_cGetChar(void)	Return character from RX Data register when valid data is available. Function does not return until character is received.
void UART_Write(char * aStr, BYTE bCnt)	Send bCnt bytes from aStr array to TX port.

Function	Description
void UART_CWrite(const char * aStr, int iCnt)	Send iCnt bytes from constant aStr array to TX port.
char UART_cReadChar(void)	Read RX Data register immediately. If valid data is not available, return 0, else ASCII char between 1 and 255 is returned.
int UART_ iReadChar(void)	Read Rx Data register immediately. If data is not available or an error condition exists, return an error status in the MSB. The received char is returned in the LSB.
void UART_PutSHexByte(BYTE bValue)	Send a two character hex representation of bValue to the TX port.
void UART_PutSHexInt(int iValue)	Send a four character hex representation of iValue to the TX port.
void UART_PutCRLF(void)	Send a carriage return (0x0D) and a line feed (0x0A) to the TX port.
void UART_CmdReset(void)	Reset Rx command buffer. [1]
BYTE UART_bCmdCheck(void)	Returns a non-zero value if a valid command terminator has been received.[1]
BYTE UART_bCmdLength(void)	Returns the current command length.[1]
char * UART_szGetParam(void)	Return pointer to next parameter in RX buffer.[1]
char * UART_szGetRestOfParams(void)	Return pointer to remaining parameter string. [1]
BYTE UART_bErrCheck(void)	Return command buffer error status.[1]

Note

1. This function requires you to enable the RxCmdBuffer parameter and set up the Rx receive buffer in the Device Editor.

UART_IntCntl

Description:

Enables or disables the RX and TX interrupts independently.

C Prototype:

```
void  UART_IntCntl(BYTE bMask)
```

Assembly:

```
mov   A,(UART_ENABLE_RX_INT|UART_ENABLE_TX_INT)
lcall  UART_IntCntl
```

Parameters:

BYTE bMask: Mask to enable or disable TX and RX blocks.

Mask	Description
UART_ENABLE_RX_INT	Enable Receiver
UART_ENABLE_TX_INT	Enable Transmitter
UART_DISABLE_RX_INT	Disable Receiver
UART_DISABLE_TX_INT	Disable Transmitter

Return Value:

None

Side Effects:

If there is a pending interrupt on the TX or RX when interrupts are enabled, the interrupt occurs immediately. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_PutString

Description:

Sends a null terminated (RAM) string to the UART TX.

C Prototype:

```
void UART_PutString(char * szRamString)
```

Assembler:

```
mov  A,>szRamString     ; Load MSB part of pointer to RAM based null
                        ; terminated string.
mov  X,<szRamString     ; Load LSB part of pointer to RAM based null
                        ; terminated string.
lcall UART_PutString     ; Call function to send string out UART TX port
```

Parameters:

char * aRamString: Pointer to the string to be sent to the UART TX. MSB is passed in the Accumulator and LSB is passed in X register.

Return Value:

None

Side Effects:

Program flow stays in this function until the last character is loaded into the UART transmit buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the IDX_PP page pointer register is modified.

UART_CPutString

Description:

Sends constant (ROM), null terminated string out the UART TX port.

C Prototype:

```
void UART_CPutString(const char * szROMString)
```

Assembler:

```
mov  A,>szRomString   ; Load MSB part of pointer to ROM based null
                      ; terminated string.
mov  X,<szRomString   ; Load LSB part of pointer to ROM based null
                      ; terminated string.
lcall UART_CPutString  ; Call function to send constant string out
                      ; UART TX
```

Parameters:

const char * szROMString: Pointer to string to be sent to the UART. MSB of string pointer is passed in the Accumulator and LSB of the pointer is passed in the X register.

Return Value:

None

Side Effects:

Program flow stays in this function, until the last character is loaded into the UART transmit buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_PutChar

Description:

Writes a single character to the UART TX port when the port buffer is empty.

C Prototype:

```
void UART_PutChar(CHAR cData)
```

Assembler:

```
mov  A,0x33           ; Load ASCII character "3" in A
lcall UART_PutChar     ; Call function to send single character to
                      ; UART TX port.
```

Parameters:

CHAR cData: The character to be sent to the UART TX port. Data is passed in the Accumulator.

Return Value:

None

Side Effects:

Program flow stays in this function until the data can be written to the UART TX buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When

necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_cGetChar

Description:

Waits for a valid character in UART RX and returns its value.

C Prototype:

```
CHAR UART_cGetChar(void)
```

Assembler:

```
lcall UART_cGetChar         ; Call function to print single character to
                           ; serial port.
mov  [CharBuffer,]A        ; Store retrieved character in buffer
```

Parameters:

None

Return Value:

Char bData: The character read from UART RX is returned in the accumulator.

Side Effects:

Program flow stays in this function until a character is received or a character was previously received but not read. The UART RX interrupt should be disabled when this function is used. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_Write

Description:

Sends ‘n‘ characters (RAM) to UART TX port.

C Prototype:

```
void UART_Write(char * szRamString, BYTE bCount)
```

Assembler:

```
mov   A,20              ; Load string/array count
push  A
mov   A,>szRamString     ; Load MSB part of pointer to RAM string
push  A
mov   A,<szRamString     ; Load LSB part of pointer to RAM string
push  A
mov   X,SP              ; Set X register to point to variables
dec   X
lcall  UART_Write        ; Make call to function
add   SP,253            ; Reset stack pointer to original position
```

Parameters:

CHAR * szRamString: Pointer to the string to be sent to the UART TX.

BYTE bCount: Number of characters to be sent to the UART TX.

Return Value:

None

Side Effects:

Program flow stays in this function until the last character is loaded into the UART TX buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the IDX_PP page pointer register is modified.

UART_CWrite

Description:

Sends n characters (ROM) to the UART TX port.

C Prototype:

```
void UART_CWrite(const char * szRomString, INT iCount)
```

Assembler:

```
mov   A,0               ; Load MSB of string/array count
push  A
mov   A,20              ; Load LSB of string/array count
push  A
mov   A,>szRomString     ; Load MSB part of pointer to ROM string
push  A
mov   A,<szRomString     ; Load LSB part of pointer to ROM string
push  A
mov   X,SP              ; Set X register to point to variables
dec   X
lcall  UART_CWrite       ; Make call to function
add   SP,252            ; Reset stack pointer to original position
```

Parameters:

const char * szRomString: Pointer to the string to be sent to the UART TX port.

int iCount: Number of characters to be sent to the UART TX port.

Return Value:

None

Side Effects:

Program flow stays in this function until the last character is loaded into the UART TX buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_cReadChar

Description:

Reads UART RX port immediately if data is not available or an error condition exists, or zero is returned. Otherwise, the character is read and returned.

C Prototype:

```
CHAR UART_cReadChar(void)
```

Assembler:

```
lcall UART_cReadChar          ; Call function to read a character
cmp  A,0x00                  ; Check for error
jz   ProcessError            ; If error, Process the error condition
mov  [CharBuffer],A          ; Store retrieved character in buffer
```

Parameters:

None

Return Value:

CHAR bData: Character read from UART RX port. ASCII characters from 1 to 255 are valid. A returned zero signifies an error condition or no data available.

Side Effects:

Function only accepts characters from 1 to 255 as valid. A 0x00 (null) character is detected as an error condition. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_iReadChar

Description:

Reads UART RX port immediately. Returns received character and error condition.

C Prototype:

```
INT UART_iReadChar(void)
```

Assembler:

```
lcall UART_iReadChar          ; Call function to read a character
cmp  X,0x00                  ; Check for error
jnz  ProcessError            ; If error, Process the error condition
mov  [CharBuffer]A           ; Store retrieved character in buffer
```

Parameters:

None

Return Value:

unsigned int iData: MSB contains status and LSB contains UART RX data. If the MSB is non-zero, an error has occurred. This table lists the possible returned error codes in the MSB:

Error Flags	Value	Description
UART_RX_PARITY_ERROR	0x80	Parity Error
UART_RX_OVERRUN_ERROR	0x40	Buffer Overrun Error
UART_RX_FRAMING_ERROR	0x20	Character Framing Error
UART_RX_NO_DATA	0x10	RX Buffer Full

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_PutSHexByte

Description:

Sends two byte ASCII Hex representation of data to the UART TX port.

C Prototype:

```
void UART_PutSHexByte(BYTE bData)
```

Assembler:

```
mov  A,0x33               ; Load data to be sent to UART TX
lcall UART_PutSHexByte     ; Call function to output hex
                          ; representation of data.  The output
                          ; for this value would be "33".
```

Parameters:

BYTE bData: Byte to be converted to ASCII string.

Return Value:

None

Side Effects:

Program flow stays in this function, until the last character is loaded into the UART transmit buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_PutSHexInt

Description:

Sends four byte ASCII Hex representation of data to the UART TX port.

C Prototype:

```
void UART_PutSHexInt(INT iData)
```

Assembler:

```
mov A,0x34           ; Load LSB in A
mov  X,0x12           ; Load MSB in X

lcall UART_PutSHexInt  ; Call function to output hex
                      ; representation of data.  The output
                      ; for this value would be "1234".
```

Parameters:

int iData: Integer to be converted to ASCII string.

Return Value:

None

Side Effects:

Program flow stays in this function, until the last character is loaded into the UART transmit buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_PutCRLF

Description:

Function to send out carriage return (0x0D) and line feed (0x0A) out of the UART TX port.

C Prototype:

```
void UART_PutCRLF(void)
```

Assembler:

```
lcall UART_PutCRLF          ; Send a carriage return and line feed out TX
```

Parameters:

None

Return Value:

None

Side Effects:

Program execution stays in this function, until all characters have been written to the UART TX buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_CmdReset

Description:

Resets command buffer and flags. This allows the characters for the next command to be accepted.

C Prototype:

```
void UART_CmdReset(void)
```

Assembler:

```
lcall UART_CmdReset         ; Call function to reset command buffer.
```

Parameters:

None

Return Value:

None

Side Effects:

Resets UART character count and clears the receive buffer. Any characters remaining in the buffer are lost. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified.

UART_bCmdCheck

Description:

Checks if command terminator has been received.

C Prototype:

```
BYTE UART_bCmdCheck(void)
```

Assembler:

```
lcall UART_bCmdCheck      ; Call function to get command complete status.
cmp  A,0x00              ; Check if command complete
jnz  ProcessCmd          ; Process command buffer
```

Parameters:

None

Return Value:

A nonzero value is returned if a command terminator has been received.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified.

UART_bCmdLength

Description:

Returns length of characters in the command buffer. This command returns the current command length, whether a command terminator has been received or not.

C Prototype:

```
BYTE UART_bCmdLength(void)
```

Assembler:

```
lcall UART_bCmdLength      ; Call function to get current command
                          ; Command length is returned in Accumulator
```

Parameters:

None

Return Value:

Length of current string in receive buffer.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified.

UART_szGetParam

Description:

Returns the next parameter from the receive buffer, which is delimited by the Param_Delimiter set in the Device Editor. After all parameters have been returned, any subsequent calls return a null pointer (zero).

C Prototype:

```
char * UART_szGetParam(void)
```

Assembler:

```
lcall UART_szGetParam       ; Call function to return pointer to the
                           ; next parameter.
                           ; Pointer is returned in A and X.
```

Parameters:

None

Return Value:

char * strPtr: Pointer to parameter string.

Side Effects:

The receive buffer is modified each time szGetParam is called. Nulls are placed after each parameter. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, the CUR_PP and IDX_PP page pointer registers are modified.

UART_szGetRestOfParams

Description:

Returns a pointer to the remainder of the receive buffer string that is not returned with szGetParam. If this function is called before szGetParam, a pointer to the entire receive buffer is returned.

C Prototype:

```
char * UART_szGetRestOfParams(void)
```

Assembler:

```
lcall UART_szGetRestOfParams    ; Call function to return pointer to the
                               ; remainder of the receive buffer.
                               ; Pointer is returned in A and X.
```

Parameters:

None

Return Value:

char * strPtr: Pointer to remainder of receive string.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified.

UART_bErrCheck

Description:

Checks command error since the last time bErrCheck was called.

C Prototype:

```
BYTE UART_bErrCheck(void)
```

Assembler:

```
lcall UART_bErrCheck            ; Call function to return error status
cmp  A,0x00                     ; Check for error
jnz  ProcessError               ; If error, Process the error condition
```

Parameters:

None

Return Value:

BYTE bErr: MSB contains the status of any error condition that may have occurred since the last time this function was called.

Error Flags	Value	Description
UART_RX_PARITY_ERROR	0x80	Parity Error
UART_RX_OVERRUN_ERROR	0x40	Buffer overrun Error
UART_RX_FRAMING_ERROR	0x20	Character framing Error
UART_RX_BUF_OVERRUN	0x10	Software RX buffer overrun

Side Effects:

Error status is cleared. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified.

Sample Firmware Source Code

Here is an assembly language code sample for the UART User module:

```
;***************************************************************
;  LoopBack:
;
;  This code sample illustrates how to setup a UART loopback.
;  Data received is echoed back to the receiver.  If an error
;  is received, then a NAK is returned.
;
;  This sample is written to be performed in the non-interrupt
;  processing.  This code could easily be written to be
;  interrupt driven.
;  In this example a Counter8 user module is used to generate
;  this clock.  The following are the parameter settings for the
;  Counter8 User Module used for this example.
;  Counter8  (placed in DBA03)
;  Clock         48MHz
;  Enable        High
;  Output        None
;  Period        155
;  CompareValue  78
;  CompareType   Less Than Or Equal
;  InterruptType Terminal Count
;
;  Using the settings above the UART baud rate can be calculated.
;  baud rate = 48MHz / ((Period+1) * (over Sample rate))
;  = 48MHz / ((155+1) * 8)
;  = 38.4 kBaud
;
;  The settings for the UART are as follows:
;
;  UART  (Placement RX => DAC06, TX => DCA07)
;  Clock              DBA03 Broadcast
;  RX Input           Global_IN_5
;  TX Output          Global_OUT_7
;  TX InterruptMode   TXComplete
;  RxCmdBuffer        Enable
;  RxBufferSize       16 Bytes
;  CommandTerminator  13      (CR)
;  Param_Delimiter    32      (space)
;  IgnoreCharsBelow   32      (Ignore all chars below space, except ;  CR)
;
;
;  First the UART and Counter8 user modules are initialized.  The
;  period and compare values for the Counter8 User Module are set to
;  generate the 8X clock.  If the period and compare values for
;  the Counter8 User Module are set in the parameter window, it is
;  not required to set them in the user program.
;
;***************************************************************
include "m8c.inc"       ; part specific constants and macros
include "PSoCAPI.inc"   ; PSoC API definitions for all User Modules

export  _main
```

```
NAK_RESPONSE:   equ   00
_main:

mov   A,UART_PARITY_NONE  ; No parity
    call  UART_Start
call  Counter8_Start
.WaitForData:               ; wait for data to be received
call  UART_bReadRxStatus
and   A, UART_RX_COMPLETE
jz    .WaitForData

and   A, UART_RX_ERROR ; data received - see if data is valid
jz    .GetData

mov   A, NAK_RESPONSE  ; error detected setup to send a NAK
jmp   .TxData
.GetData:

call  UART_bReadRxData  ; read the data from the receiver
.TxData:

call  UART_SendData     ; transmit the response data
jmp   .WaitForData      ; go wait for next byte
```

C code example:

```
//-------------------------------------------------------------------
// Description:
//    This example program shows how easy it is to use the PSoC
//    serial port with this user module.
//
//    The UART requires a clock that is 8 times the desired Baud Rate.
//    In this example a Counter8 user module is used to generate
//    this clock.  The following are the parameter settings for the
//    Counter8 User Module used for this example.
//
//    Counter8  (placed in DBA03)
//      Clock         48MHz
//      Enable        High
//      Output        None
//      Period        155
//      CompareValue  78
//      CompareType   Less Than Or Equal
//      InterruptType Terminal Count
//
//    Using the settings above the UART baud rate can be calculated.
//    baud rate = 48MHz / ((Period+1) * (over Sample rate))
//              = 48MHz / ((155+1) * 8)
//              = 38.4 kBaud
//
//    The settings for the UART are as follows:
//
//    UART  (Placement RX => DAC06, TX => DCA07)
//      Clock              DBA03 Broadcast
```

```
//      RX Input           Global_IN_5
//      TX Output          Global_OUT_7
//      TX InterruptMode   TXComplete
//      RxCmdBuffer        Enable
//      RxBufferSize       16 Bytes
//      CommandTerminator  13      (CR)
//      Param_Delimiter    32      (space)
//      IgnoreCharsBelow   32      (Ignore all chars below space, except CR)
//
//
//    First the UART and Counter8 user modules are initialized.  The
//    period and compare values for the Counter8 User Module are set to
//    generate the 8X clock.  If the period and compare values for
//    the Counter8 User Module are set in the parameter window, it is
//    not required to set them in the user program, but are set here
//    to make it clear how the UART clock is generated. Next a
//    sample string is printed to welcome you to the program.  The
//    program then sits and waits for characters until a command
//    terminator is received (CR).  Once the command terminator is received
//    it parses the parameters and outputs them to the UART TX port.
//    The command buffer is then reset and waits for the next command.
//
//  Example:
//  Input:
//             "foobar aa bbb cc"(CR)
//
//  Output:
//
//    Welcome to PSoC UARTlus test program. V1.1
//    Found valid command
//    Command =>foobar<
//    Parameters:
//     <aa>
//      <bbb>
//      <c>
//
//-------------------------------------------------------------------
#include <m8c.h>
#include "PSoCAPI.h"

void main(void)
{

    char * strPtr;                      // Parameter pointer

UART_CmdReset();                      // Initialize receiver/cmd
                                        // buffer
    UART_IntCntl(UART_ENABLE_RX_INT);   // Enable RX interrupts

    Counter8_WritePeriod(155);          // Set up baud rate generator
    Counter8_WriteCompareValue(77);
    Counter8_Start();                   // Turn on baud rate generator

UART_Start(UART_PARITY_NONE);         // Enable UART
```

```
M8C_EnableGInt ;                        // Turn on interrupts

UART_CPutString("\r\nWelcome to PSoC UART test program. V1.1 \r\n");

while(1) {
       if(UART_bCmdCheck()) {                    // Wait for command
if(strPtr = UART_szGetParam()) {       // More than delimiter"
             UART_CPutString("Found valid command\r\nCommand =>");
   UART_PutString(strPtr);             // Print out command
   UART_CPutString("<\r\nParamaters:\r\n");
   while(strPtr = UART_szGetParam()) { // loop on each parameter
      UART_CPutString("   <");
      UART_PutString(strPtr);          // Print each parameter
      UART_CPutString(">\r\n");
   }
}
UART_CmdReset();                           // Reset command buffer
}
   }
}
```

Configuration Registers

This section describes the PSoC block and other system registers used or modified by the UART User Module. Only the parameterized symbols are explained.

Table 4. Block RX, Data Shift Register: DR0

Bit	7	6	5	4	3	2	1	0
Value	RX Shift Register							

RX Shift Register: When a start bit is detected on the input, the RX state machine hardware generates a divide-by-8 bit clock that shifts data into this register.

Table 5. Block RX, Data Register: DR1

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	0

This register is not used.

Table 6. Block RX, Data Buffer Register: DR2

Bit	7	6	5	4	3	2	1	0
Value	RX Buffer Register							

RX Buffer Register: Data is transferred from the RX Shift register after the stop bit has been sampled.

Table 7. Block RX, Control Register: CR0

Bit	7	6	5	4	3	2	1	0
Value	Parity Error	Overrun Error	Framing Error	Rx Active	Rx Reg Full	Parity Type	Parity Enable	Rx Enable

Parity Error is a flag that indicates parity computation result of received data byte. Overrun Error is a flag that indicates that the RX Buffer register data is overwritten. Framing Error is a flag that indicates the stop bit was properly received. Rx Active is a flag that indicates whether or not a data byte is actively being

received. Rx Reg Full is a flag that indicates a data byte has been completely received, the data byte has been transferred to the Rx Buffer register, and the error conditions are valid. Parity Type is a type of parity to compute. This bit is a “don’t care if Parity Enable bit is not set." Parity Enable enables or disables the computation of the received parity bit. Parity is selected by setting the Parity Type bit. Rx Enable enables or disables the RX8 receiver.

A read of the Rx Control register puts the status data onto the data bus and clears all the status bits. When parity or overflow errors are detected, the firmware should not need to perform any actions. If a framing error is detected, the framer immediately begins to look for the next data byte. If there is a chance in the system that the RX input could be stuck at logic 0 for an extended period of time, the receiver should be stopped and the line should be polled for a return to logic 1 before the re-enabling the receiver. If the RX input is stuck at logic 0, this avoids repeated detection of 'false' start bits while the line is logic 0 and resulting framing errors when a logic 0 is also detected where the start bit should be.

Table 8. Block RX, Register: Function

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	1	0	1

This register defines the personality of this Digital Communications Type ‘A’ Block to be a receiver.

Table 9. Block RX, Register: Input

Bit	7	6	5	4	3	2	1	0
Value	RX Input				Clock Source			

RX Input selects the RX input source. Clock Source selects the clock to drive the receiver timing.

Table 10. Block RX, Register: Output

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	0

Table 11. Block TX, Data Shift Register: DR0

Bit	7	6	5	4	3	2	1	0
Value	TX Shift Register							

TX Shift Register is controlled by the TX state machine hardware, to shift the contents and transmit the least significant bit. Data is loaded into this register from the DR1 data register by the TX state machine hardware.

Table 12. Block TX, Data Buffer Register: DR1

Bit	7	6	5	4	3	2	1	0
Value	TX Buffer Register							

TX Buffer Register is used to transmit data that is written to this register by the user module APIs. The data loaded into this register is transferred to the TX Shift register by the TX state machine.

Table 13. Block TX, Data Register: DR2

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	0

This register is not used.

Table 14. Block TX, Control/Status Register: CR0

Bit	7	6	5	4	3	2	1	0
Value	Reserved	Reserved	TX Complete	TX Reg Empty	Reserved	Parity Type	Parity Enable	TX Enable

Tx Complete is a flag that indicates if a data byte is in the process of being transmitted. This bit is reset when the register is read. Tx Reg Empty is a flag that indicates when the Buffer register is empty. Parity Type is the type of parity to compute. This bit is a “don’t care if Parity Enable bit is not set." Parity Enable enables or disables the computation and transmission of a parity bit with data byte. Parity is selected by setting the Parity Type bit. Tx Enable enables or disables the TX transmitter.

Table 15. Block TX, Register: Function

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	1	1	0	1

This register defines the personality of this Digital Communications Type ‘A’ Block to be a transmitter.

Table 16. Block TX, Register: Input

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	Clock Source			

Clock Source is the selected clock to drive the transmitter timing.

Table 17. Block TX, Register: Output

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	TX Output Enable and Select		

TX Output Enable and Select specifies the output of the TX8.

Version History

Version	Originator	Description
5.3	DHA	Added compatibility for Large Memory Model chips.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.

Copyright © 2001-2015 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.