IR Optical Transmitter Datasheet IrDATX V 2.3

Copyright © 2002-2015 Cypress Semiconductor Corporation. All Rights Reserved.

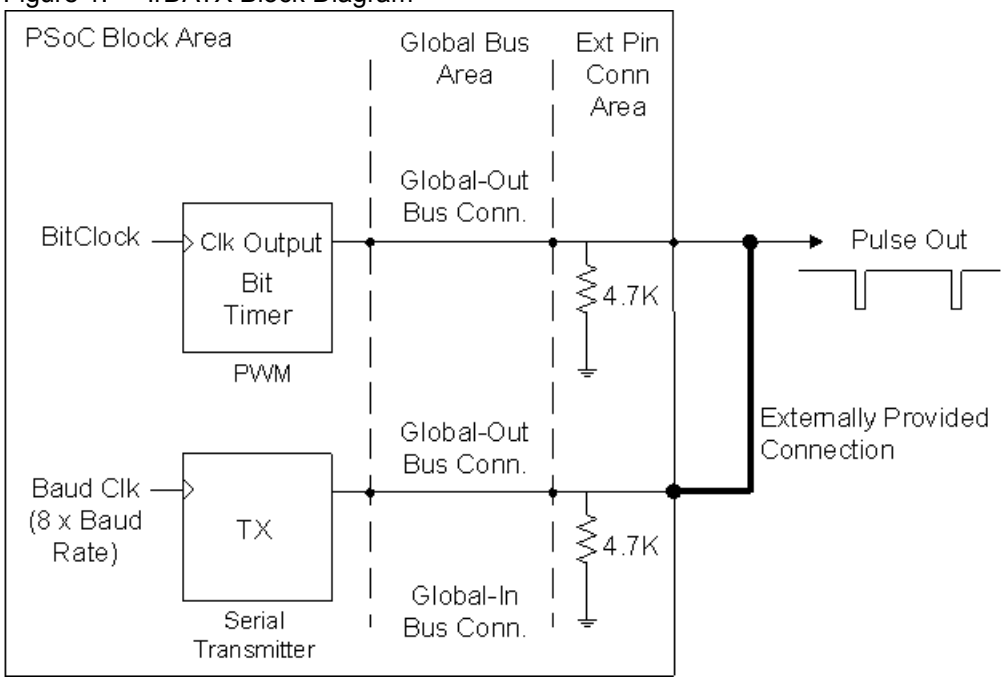
Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	Flash	RAM	
CY8C29/27/24/22/21xxx, CY8C23x33, CYWUSB6953, CY8CLED02/04/08/16, CY8CLED0xD, CY8CLED0xG, CY8C21x45, CY8C22x45, CY8C28x45, CY8CPLC20, CY8CLED16P01, CY8C28xxx						
	2	0	0	51	0	2 for Wired-AND Interface

Features and Overview

- Hardware implementation of IrDA low-speed, physical-layer transmitter
- Data bit rate selectable to a maximum transmit rate of 115.2 kbps
- Optional interrupt on transmit buffer empty

The IrDATX User Module is an 8-bit serial half-duplex transmitter that implements the IrDA low-speed physical layer protocol for infrared communications. Baud rates up to 115.2 kbps can be generated. The data format includes a start bit, 8 bits of data, and a stop bit. Flexible clocking and interrupts are supported. Application Programming Interface (API) firmware routines are provided to initialize, configure, and transmit data. Additional information regarding IrDA is available at http://www.irda.org.

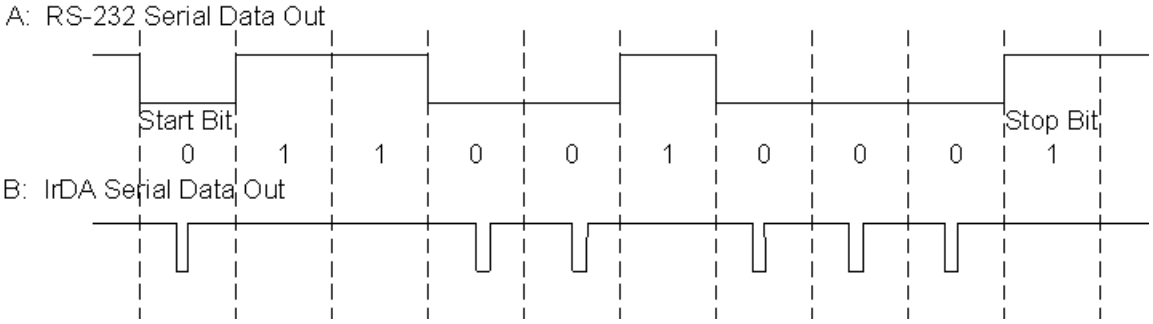
Figure 1. IrDATX Block Diagram



Functional Description

The IrDATX User Module optimizes standard serial communication for optical-wireless transmission. An RS-232-compliant serial signal consists of an electrical signal idling at a non-zero (logic 1) voltage and data bits of a predefined duration framed by a start (logic 0), parity bit, and stop (logic 1) bit. To aid in detection of correct logic levels, voltages are held stable for the entire duration of a bit and detected by measuring voltage during the middle of a bit. This is shown in the following figure.

Figure 2. IrDA Serial Data versus RS-232



The IrDA protocol uses brief duration, high-intensity optical pulses to set timing for transmitted bits. Pulses are only transmitted on data values of logic 0. The stop bit, logic 1, acts as a quiet bit, setting a minimum delay before the next start bit. This is shown in the previous figure.

The IrDA specification allows data pulses between 1.67 microseconds and a maximum of 3/16 of one bit width at the operating baud rate. There is a firm minimum pulse width, but a variable maximum pulse width.

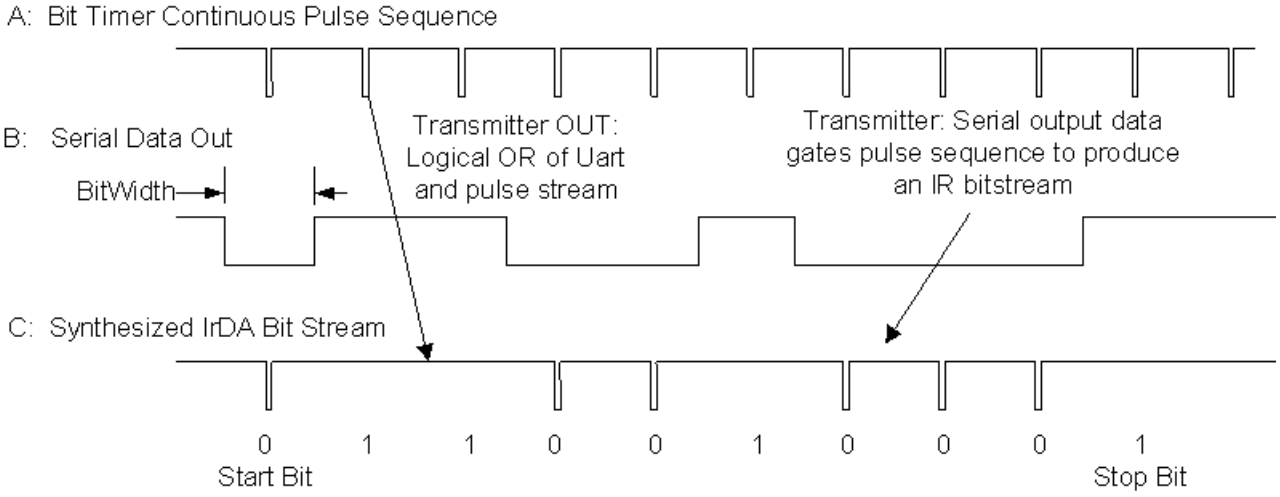
Leveraging the fact that the RS-232 and IrDA byte protocol are the same, the IrDATX User Module operates by logically combining serial data from an internal serial transmitter (TX) with a pulse-width modulator (BitTimer), generating a continuous pulse stream of the correct timing, width, and synchronization.

TX is a communications PSoC block configured as an RS-232-compliant transmitter. When a byte is transmitted, it generates a start bit, eight data bits, and a trailing stop bit. BitTimer is a digital PSoC block configured as a PWM that generates the correct IrDA bit-width pulses, in synchronization with the TX serial output stream.

When a byte of data is to be transmitted, it is loaded into the TX transmit buffer register. The data is transferred to the TX Shift register. On successive positive BaudClock clock edges, the start bit is transmitted, followed by the data bits and a final stop bit.

Referring back to the block diagram, note that the Output and the BitTimer Output pins are connected. *The user must provide this external connection*. The pins are internally set up as resistive pull downs. In this configuration, the TX serial output signal acts as a ‘gate’ for the BitTimer pulse stream. High logic levels block the pulse stream during transmission of data and when no data is being transmitted. As the following figure illustrates, this effectively creates a logical “Wired OR” between the BitTimer output and the TX output data pulses.

Figure 3. Detailed Timing of IrDA Output Data Logic



The design choice, to provide negative output pulses, is due to the ability of the PSoC chip to sink more current (50 mA max) than source it. This provides the ability to apply more drive to the optical transmitter device. An IrDA transmitter device can be connected directly to the output pin. However, if more power is required to drive the device, then buffering circuitry must be provided. When connecting directly to the IrDA output node, sink current is limited by the parallel combination of the pull-down resistors. Connecting an internal digital inverter to the output node of the IrDA User Module, and configuring the optical transmitter to accept an inverted version of the signal shown in the previous figure, can provide higher output drive current.

The BaudClk input signal must be set to eight times the desired baud rate. If the BitTimerClk input signal is set to the same clock as the BaudClk, then an output pulse of 1/8 a bit period is generated.

For a baud rate of 9600, this will create pulses that are 1/9600 * 1/8 = 13 us. For all baud rates, except for 115.2 kbps, the 1/8 output pulse meets the specified pulse width timing of 1.67 us. At 115.2 kbps, the pulse width must be modified using the supplied API. To modify pulse-width timing, the BitTimerClk clock input and the BitTimerConfigOverride API can be used to adjust the output pulse width at any baud rate.

The IrDA rationale for providing a minimum pulse width of 1.67 us is to reduce current when driving the optical transmitter device. For baud rates in which the 1/8 pulse width generated is too long to support the desired current load, the pulse width output can be modified.

Note that the IrDA protocol is half duplex. To save resources, the IrDARX User Module can be placed in the same PSoC blocks, but in a different configuration. Then, based on the state of the transmission, the specific IrDA User Module, transmitter or receiver, can be activated.

The Start() API function initializes the IrDATX User Module for operation. It configures and initializes the TX serial transmitter and the BitTimer pulse width generator.

Calling the SendData() API function will load the TX buffer with the data to be transmitted. The rising edge of the next bit clock transfers the data to the Shift register and sets the Buffer Empty bit of the Status register. If the interrupt enable mask is enabled, an interrupt will be triggered. This interrupt enables the queuing of the next byte to transmit so that upon completion of transmission of the current data byte, the new byte will be transmitted on the next available transmit clock.

When the data byte is completely transmitted, the TX Complete status bit is set in the Status register. Reading the Status register is performed by way of an API call to ReadStatus().

Pin Configuration

The IrDATX User Module works well when configured as a loadable configuration. Take care to set pin drive settings in other loadable configurations so that external devices are driven (or not driven) when the IrDATX module is not in use. Although certain pins are set with specific drive settings when the IrDATX module is loaded, the pin settings are set to their ‘base’ configuration when the IrDATX module is unloaded.

Because this user module uses pull-down pin drive settings, care should be taken when using ‘read-modify-write’ commands on the ports where these pins are defined. Ideally, a shadow register should be used to alter data settings on the port.

For example, you may wish to avoid changing bit-0 of portX. If the pin is configured as a pull up or pull down but and external source is driving it to a different state than the output register, the external data may be read and then re-written to the port, resulting in un-intended alteration of the output data.

Condition:

PRTXDR is configured with bit 0 driven with a pull up. Data in the Data register is set to bit-0 = 1. An external source is pulling the output of bit 1 low. When the port is read, a 0 will be read at bit 0, rather than the 1 in the Data register. A read-modify-write operation on the Data register will then read the zero in bit 0 and write it back to the Data register.

In C:

```
BYTE portXShadow;
//make sure that any changes to PRTXDR are recorded in portXShadow
portXShadow = 0x80 | portXShadow;
PRTXDR = portXShadow;
```

In Assembly:

```
portXShadow: blk 1
;make sure any changes to PRTXDR are recorded in portXShadow
or   [portXShadow], 0x80
mov  A, [portXShadow]
mov  reg[PRTXDR], A
```

DC and AC Electrical Characteristics

Table 1. IrDATX DC and AC Electrical Characteristics

Parameter	Conditions and Notes	Typical	Limit	Units
F_{max}	Maximum character transmission frequency	--	115.2	Kbits/s

Timing

The BaudClk must be set to eight times the desired bit transmit rate. Under most situations, the BitTimerClk can be set to the same clock as the BaudClk. It is imperative that these two clocks be in phase with each other. By default, the BitTimer is set up with a period of eight clocks and a compare value to provide a 1/8 bit pulse. This satisfies the IrDA specification.

To adjust the output pulse width to other than the nominal 1/8 bit period, perform the following.

- The relationship between the pulse width output and the TX bit period is a multiple of eight. When the BitTimerClk and BaudClk are set equal, the output from the TX is divided by N/8, where N can be from 1 to 8. In the default case, N is set to 1 to generate an output pulse that is 1/8 of the bit period.
- To increase the output bit period resolution, the BitTimer pulse width modulator can be re-configured to divide the output pulse to any ratio. However, the BitTimer *must* be set with a clock input that is an integer multiple of the BaudClk and is input to the TX transmitter block.
- For example, to create an output pulse width of 3/16 of a bit period, the BitTimerClk could be set to twice the BaudClk. Then the API function BitTimerConfigOverride() is called with the parameters 16 and 3. The correct period and duty cycle will be configured.
- To create the minimum pulse width of 1.67 us at 9600 baud, the output pulse width would be approximated by 1/9600 * 1/(8*8). The BitTimerClk would be set to eight times the BaudClk clock input or 614.4 kHz. BitTimerConfigOverride() would be called with the parameters of 64 and 1. This would generate a pulse width of 1.627 us – nominally meeting the specification.
- To create a 19.2 Kbits/s data rate with a 3/16 bit pulse width using the global 24V1 or 24V2 clock and a single Counter8 User Module, proceed as follows.

1. Set the 24V1 or 24V2 global clock with a divisor of 12, to generate a 2-MHz output.
2. Set up the Counter8 with a Period count of 12 and a CompareValue of 6. This creates an output frequency of (24 MHz/12)/(12+1) = 153.846 kHz.
3. Set the BitTimerClk to the 2-MHz clock.
4. Set the BaudRateClk to the 153.846 kHz clock.
5. Call BitTimerConfigOverride() with the parameters of 104 and 20, to correctly set the pulse width.

Placement

The IrDATX User Module is composed of two digital blocks. The TX block must be placed in a PSoC Digital Communication block and the BitTimer block may be placed in any digital PSoC block.

Parameters and Resources

BaudClk

This is the TX block clock. It may be clocked by one of 16 possible sources. The Global I/O busses may 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 should be turned off for best 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 baud rate. One data or framing bit is transmitted every eight clock cycles.

BitTimerClk

The BitTimerClk and the BaudClk should be set to the same clock source. This will generate an output pulse width of 1/8 of the baud bit rate. It may be clocked by the same clock sources as the BaudClk.

For cases in which the BitTimerClk must be modified, the clock input must be an integer multiple of the BaudClk.

TX and BitTimer ClockSync

In the PSoC devices, digital blocks may provide clock sources in addition to the system clocks. Digital clock sources may even be chained in ripple fashion. This introduces skew with respect to the system clocks. These skews are more critical in the CY8C29/27/24/22xxx PSoC device families because of

various data-path optimizations, particularly those applied to the system busses. These parameter which specify the synchronization clock for each of the TX and the BitTimer blocks may be used to control clock skew and ensure proper operation when reading and writing PSoC block register values. Appropriate values for these parameters 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), 32KHz, 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 when a 24 MHz (SysClk/1) clock is desired. 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, above. 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 when the 48 MHz (SysClk*2) input is selected. Use when unsynchronized inputs are desired. In general this use is advisable only when interrupt generation is the sole application of the Counter.

TX Output Bus

The TX Output Bus defines which global bus is used to connect the TX block to the an output pin configured with ‘pull-down’ drive level.

TX Output Pin

Under programmatic control, the TX Output pin will be configured with ‘pull-down’ drive level. This output pin, together with the BitTimer output pin, forms the logical OR to sum the two outputs and form an IrDA output data stream.

Pin drive configuration for the TX Output Pin is done automatically by the user module and cannot be over-ridden and saved in the PSoC Designer – Interconnect View.

BitTimer Output Bus

The BitTimer Output Bus defines which global bus is used to connect the BitTimer block to the BitTimer output pin which is configured with ‘pull-down’ drive level.

BitTimer Output Pin

Under programmatic control, the BitTimer Output pin will be configured with ‘pull-down’ drive level. This output pin, together with the TX output pin, forms the logical OR to sum the two outputs and form an IrDA output data stream.

Pin drive configuration for the BitTimer Output Pin is done automatically by the user module and cannot be over-ridden and saved in the PSoC Designer – Interconnect View.

Note TX Output Pin and the BitTimer Output Pin must be connected externally.

BitTimer_Start_Delay

This parameter sets the number of CPU clock cycle delays between the start of the TX and the BitTimer blocks. This parameter would only be set to other than NONE, when there are synchroniza-

tion issues concerning the generation of the TX and BitTimer output pulses. See the discussion in the Timing section of this user module.

Interrupt Generation Control

There are two additional parameters that become available when the **Enable interrupt generation control** check box in PSoC Designer is checked. This is available under **Project > Settings > Chip Editor**. Interrupt Generation Control is important when multiple overlays are used with interrupts shared by multiple user modules across overlays:

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. Properly selecting whether an Interrupt API is to be generated is recommended particularly with projects that have multiple overlays where a single block resource is used by the different overlays. By selecting only Interrupt API generation when it is necessary the need to generate an interrupt dispatch code might 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. Selecting "ActiveStatus" causes firmware to test 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. Selecting "OffsetPreCalc" causes firmware to calculate the source of a shared interrupt request only when an overlay is initially loaded. This calculation decreases 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 provided as part of the user module to allow the designer 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 prior to 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 will 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 Application Programming Interface routines will allow programmatic control of the IrDATX User Module.

IrDATX_Start

Description:

Configures, initializes, and enables the IrDATx User Module. Once enabled, data can be transmitted.

C Prototype:

```
void  IrDATX_Start(void)
```

Assembly:

```
lcall  IrDATX_Start
```

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

IrDATX_Stop

Description:

Disables the IrDATX User Module.

C Prototype:

```
void  IrDATX_Stop(void)
```

Assembly:

```
lcall  IrDATX_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). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

IrDATX_EnableInt

Description:

Enables the IrDATX interrupt on BufferEmpty condition.

C Prototype:

```
void  IrDATX_EnableInt(void)
```

Assembly:

```
lcall  IrDATX_EnableInt
```

Parameters

None

Return Value:

None

Side Effects:

If an interrupt is pending and this API is called, an interrupt will be triggered immediately. This call should be made prior to 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). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

IrDATX_DisableInt

Description:

Disables the BufferEmpty interrupt.

C Prototype:

```
void  IrDATX_DisableInt(void)
```

Assembly:

```
lcall  IrDATX_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). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

IrDATX_SendData

Description:

Initiates data transmission by loading the TX Buffer register with the specified data to transmit. The TX Complete status bit in the Status register should be monitored to make sure transmission was initiated.

C Prototype:

```
void  IrDATX_SendData(BYTE bTxData)
```

Assembly:

```
mov   A, bTxData
lcall  IrDATX_SendData
```

Parameters:

bTxData: Data to be transmitted. Data is passed in the Accumulator.

Return Value:

None

Side Effects:

If the interrupt on the Tx Buffer Empty condition is set up, an interrupt will be generated when bTxData is transferred from the TX Buffer register to the TX Shift register. 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). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

IrDATX_bReadTxStatus

Description:

Returns the contents of the Status register.

C Prototype:

```
BYTE  IrDATX_bReadTxStatus(void)
```

Assembly:

```
lcall  IrDATX_bReadTxStatus
and   A, IrDA_TX_COMPLETE
jnz   TxIsComplete
```

Parameters:

None

Return Value:

Returns status byte read. Utilize defined masks to test for specific status conditions. Note that masks can be OR’ed together to check for multiple conditions.

IrDATX Status Masks	Value
IRDA_TX_BUFFER_EMPTY	0x10
IRDA_TX_COMPLETE	0x20

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

IrDATX_BitTimerOverride

Description:

Overrides the default the BitTimer settings to generate specific output-pulse bit periods. This function sets the output baud bit rate fractional multiplier. This function ONLY needs to be called if modification of the output bit pulse is required. Note that the settings of this function are based on the clock input to the BitTimerClk. See the discussion in the Timing section for details and set up requirements.

C Prototype:

```
void  IrDATX_BitTimerOverride(BYTE bDenominator, BYTE bNumerator)
```

Assembly:

```
mov   A, bDenominator
mov   X, bNumerator
lcall  IrDATX_BitTimerOverride
```

Parameters:

bNumerator: The numerator of the bit rate fractional multiplier. bDenominator: The denominator of the bit rate fractional multiplier.

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

Sample Firmware Source Code

```
;*****************************************************************
;  Name:   TxZeroTerminatedRamString
;
;  Description:
;  Transmits a zero terminated RAM based string. This function
;  assumes IrDATX_Start has already been called.
;
;  Parameters:
;  BYTE * pbStrPtr - pointer to the string - passed in the
;  X register.
;
;  Return: None
;
;*****************************************************************
include "IrDATX.inc"
export TxZeroTerminatedRamString

TxZeroTerminatedRamString:
.TxNextByte:
; check to see if end has been reached.
    mov   A, [X]
    jz    .AllDone

; transmit the next data byte
    call  IrDATX_SendData

.WaitForTxStart:
    ; wait for data to start transmitting
    call  IrDATX_bReadTxStatus
    and   A, IrDA_TX_BUFFER_EMPTY
    jz    .WaitForTxStart
    ; increment the pointer to the next byte in the string
```

```
    inc   X
    ; data byte transmitted - send the next one
    jmp   .TxNextByte

.AllDone:
    ; string completely transmitted!
    ret
```

A same code written in C is as follows.

```
#include "IrDATX.h"
#include "m8c.h"

void TxZeroTerminatedRamString( BYTE * pbStrPtr)
{
   /* check for the end condition, before sending the next byte */
   while( *pbStrPtr != 0 )
   {
      /* send the next byte */
      IrDATX_SendData( *pbStrPtr );

      /* Wait for the data to start transmitting */
      while( !( IrDATX_bReadTxStatus() & IrDA_TX_BUFFER_EMPTY ) );

      pbStrPtr++;
   }
}
```

Configuration Registers

This section describes the PSoC block and other system registers used or modified by the IrDATX User Module.

TX Configuration Registers

Table 2. 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 serial transmitter block.

Table 3. Block TX, Register: Input

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

Table 4. Block TX, Register: Output

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	TX Output Bus		

Table 5. Block TX, Data Shift Register: DR0

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

Table 6. Block TX, Data Buffer Register: DR1

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

TX Buffer Register: The data to transmit 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 7. 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 8. Block TX, Control/Status Register: CR0

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

Tx Complete is a flag that indicates if a data byte is in process of being transmitted. This bit is reset when the register is read. It can be accessed via one of the API functions.

Tx Reg Empty is a flag that indicates when the Buffer register is empty. It can be accessed via one of the API functions.

Tx Enable enables or disables the TX transmitter. It can be set via one of the API functions.

BitTimer Configuration Registers

Table 9. Block BitTimer, Register: Function

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

This register configures this PSoC block to function as a pulse width modulator.

Table 10. Block BitTimer, Register: Input

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

Table 11. Block BitTimer, Register: Output

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	BitTimer Output Bus		

Table 12. Block BitTimer, Counter Register: DR0

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

Table 13. Block BitTimer, Period Register: DR1

Bit	7	6	5	4	3	2	1	0
Value	BitTimer Pulse Divisor Denominator							

Table 14. Block BitTimer, CompareValue Register: DR2

Bit	7	6	5	4	3	2	1	0
Value	BitTimer Pulse Divisor Numerator							

Table 15. Block BitTimer, Control Register: CR0

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

Version History

Version	Originator	Description
2.3	DHA	Added Version History.

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 © 2002-2015 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.