SPI Master Datasheet SPIM V 3.00

Copyright © 2002-2013 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	CapSense®	I2C/SPI	Timer	Flash	RAM	
CY8C20x34, CY8C20x24, CY8C20x66, CY8C20x36, CY8C20336AN, CY8C20436AN, CY8C20636AN, CY8C20xx6AS, CY8C20XX6L, CY8C20x46, CY8C20x96, CY7C604xx, CY7C643xx, CYONS2010, CYONS2011, CYONSFN2051, CYONSFN2053, CYONSFN2061, CYONSFN2151, CYONSFN2161, CYONSFN2162, CYONSFN2010-BFXC, CYONSCN2024-BFXC, CYONSCN2028-BFXC, CYONSCN2020-BFXC, CYONSKN2033-BFXC, CYONSKN2035-BFXC, CYONSKN2030-BFXC, CYONSTN2040, CY8CTST200, CY8CTMG2xx, CY8C20xx7/7S, CYRF89x35, CY8C20065, CY8C24x93, CY7C69xxx						
		X		27	0	3 - 4

Features and Overview

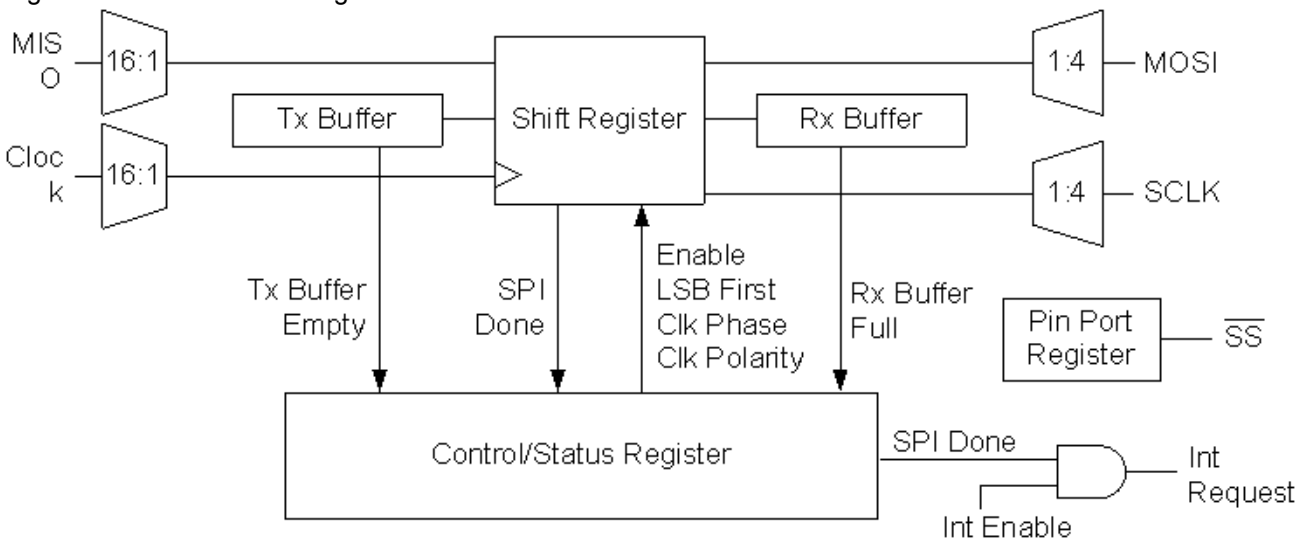
- Supports Serial Peripheral Interconnect (SPI) Master protocol
- Supports SPI clocking modes 0, 1, 2, and 3
- Selectable input sources for clock and MISO
- Selectable output routing for MOSI and SCLK
- Programmable interrupt on SPI done condition
- SPI Slave devices are independently selected

For more details on how SPI works in PSoC 1 devices, read the application note AN51234 - Getting Started with SPI in PSoC® 1.

The SPIM User Module is a Serial Peripheral Interconnect Master. It performs full duplex synchronous 8-bit data transfers. SCLK phase, SCLK polarity, and LSB First are available to accommodate most SPI clocking modes. Controlled by user supplied software, the slave select signal is able to control one or

more SPI Slave devices. The SPIM PSoC block has selectable routing for the input and output signals and programmable interrupt driven control.

Figure 1. SPIM Block Diagram



Functional Description

SPIM is a user module that implements a Serial Peripheral Interconnect Master. It uses the Tx Buffer, Rx Buffer, Control, and Configuration registers of I2C/SPI PSoC block, a data register for data shifting, and one or more Pin Port registers.

The Control register is initialized and configured using the Device Editor and/or the SPIM User Module firmware Application Programming Interface (API) routines. Initialization includes setting the LSB First configuration and the SPI transmission/receive clocking modes. SPI modes 0, 1, 2, and 3 are supported. Both the SPI master and slaves must be set with the same clock mode and bit configuration in order to properly communicate. The SPI modes are defined as follows.

Table 1. SPI Modes

Mode	SCLK Edge Performing Data Latch	Clock Polarity	Notes
0	Leading	Noninverting	Leading edge latches data. Data changes on trailing edge of clock.
1	Leading	Inverted	
2	Trailing	Noninverting	Trailing edge latches data. Data changes on leading edge.
3	Trailing	Inverted	

Note Check to ensure that the clock polarity and phase for the selected mode match the settings used for any attached SPI devices, as mode numbers may not be the same for all devices.

The active low slave select signal(s), ~SS, must be set with user supplied software routines to control selected Pin Port register bits, enabling the SPI Slave devices properly. One or more slave select signals are configurable, but only one slave select signal is active at a time.

To accommodate all of the SPI clocking modes, the slave select signal should be asserted and de-asserted for each byte of data transmitted from the SPI Master to the selected SPI Slave device. However,

this is not a strict requirement, since there are variations in the specific byte transport protocols used for SPI communications between SPI Master and Slave devices.

The SCLK signal is the SPI transmit and receive clock. It is one half the clock rate of the input clock signal. The effective transmit and receive bit rate is the input clock divided by two. The input clock is specified using the Device Editor.

The MOSI signal is the Master Out Slave In data signal that transmits the data from the master to a slave SPI protocol-compliant device. The MISO signal is the Master In Slave Out data signal that transmits the data from the slave SPI device to this user module.

The SPIM hardware transmits data from the master SPI device on the MOSI signal and simultaneously receives data from the selected slave SPI device on the MISO signal. The same SCLK signal is used for both transmit and receive of the master and slave data.

The SPI protocol is a master only initiated response protocol. It is the master's responsibility to determine that the selected slave device is ready for a command or has data ready to be received.

The SPIM User Module is enabled for operation when the SPI enable bit is set in the Control register using an API routine. At this time, all slave select signals should be asserted high.

Before transmitting a byte to a selected SPI Slave device, the specified slave select signal should be asserted low.

The transmitted data byte is written to the Tx Buffer register. This clears the Tx Buffer Empty status bit in the Control register. On the next clock, the data in the Tx Buffer is transferred to the Shift register and the Tx Buffer Empty status bit is set. To prevent a transmit overrun condition check the Buffer Empty status bit before writing a byte to the Tx Buffer register. Another data byte to transmit is written (preloaded), to the Tx Buffer register at this time. Upon completion of the transmission of the current byte, this data is transmitted without delay on the next SCLK signal.

For each bit of the data byte in the Shift register, the SCLK output signal is generated, the data in the Shift register is shifted to the MOSI output, and the input data from the slave SPI is shifted into the Shift register from the MISO input. The specific timing of the SCLK, MOSI, and MISO signals are based upon the SPI clock mode configuration.

After all of the bits are transmitted and simultaneously received, the received data is transferred from the Shift register to the Rx Buffer register, and the Tx Buffer register is transferred to the Shift register. The Rx Buffer Full and the SPI Done status bits are set. The SPI Done status bit causes an interrupt to trigger, if you enable interrupts.

If the SPI Done interrupt condition is not used to retrieve the data byte from the Rx Buffer register, poll the Control register to monitor the Rx Buffer Full status bit. Read the received data from the Rx Buffer register before the next data byte is fully received or the Overrun Error status bit is set.

If a pending byte of data was preloaded into the Tx Buffer register, then this byte restarts the SPI state machine and transmission starts immediately.

Two methods are available to control the slave select signal after each data byte was transmitted: noninterrupt or interrupt level. The selection of which method to use depends upon the SPI clocking mode, the byte transport protocol, and the required bit rate speed of the transmitted data. SPI clocking mode 0 and 1 require that you toggle the slave select off and then on again before the next byte of data is transmitted. The byte transport protocol used for multiple byte transmissions to the same slave device may or may not require that you toggle the slave select. Toggling the slave select may also have a slave device ramification, in that a set up time may be required to allow the slave device to assert its data on the MISO signal before the SCLK signal is initiated.

Controlling the slave select signal at non interrupt level requires that you dedicate the microprocessor to monitoring or sampling the SPI Done status bit, while SPI data is transmitted. If the data bit rate is substantially high, on the order of 1 MHz or faster, the overhead in monitoring the status bit is very limited.

For slower bit rates where the overhead in monitoring the SPI Done status bit may tie up the microprocessor from doing any other required operations, performing the slave select control is done at interrupt level. There is a minimum interrupt latency of 833 ns (at a 24 MHz clock rate), plus the time it takes to exercise the instructions to manage the slave select.

Multiple byte data transfers, where the Tx Buffer is preloaded with data to transmit, may be a cause for concern. However, it may be possible to achieve a 1 MHz or better bit transfer rate if minimal or no inter-byte processing is required to manage the selected slave signal, and when multiple byte transmissions are performed to the same SPI Slave device.

When the final data is transmitted, monitor the SPI Done bit to determine when to disable the SPIM User Module. This guarantees that all of the clocking signals are complete between the master and slave SPI devices.

DC and AC Electrical Characteristics

Table 2. SPIM DC and AC Electrical Characteristics

Parameter	Conditions and Notes	Typical	Limit	Units
F_{max}	Maximum bit rate	--	12	MHz

Placement

SPIM maps onto a single PSoC block and may be placed in the I2C/SPI block.

Reserve the Pin Port register bit(s) for use by the slave select signal(s) to control SPI Slave devices. Configure these port bits as standard CPU port pins.

Timing

Set the clock rate to two times the desired bit rate.

The timing of the SPI data transmission must also account for the delay caused by processing the slave select signal. It is important to take into account the SPI Slave device’s select signal set up time.

Parameters and Resources

Interrupt Mode

This option determines when an interrupt is generated for the TX block. The "TxRegEmpty" option generates an interrupt 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 of the Shift register. This second option is useful when it is important to know when the data byte was 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.

Clock Select

This parameter sets the operating frequency of the SPI Master. It is based upon the SysClk setting in Global Resources. For example, if the SysClk frequency is set to 12 MHz by either selecting a Power

Setting of "3.3V/12MHz" or "5.0V/12MHz" in Global Resources and a Clock Select of "SysClk/4" is selected in user module parameters, then the bitrate of the SPI Master is 3MBits/sec. Global Resources and User Module Parameters are found in the Interconnect View of the Device Editor.

The Clock Select options are:

SysClk/2

SysClk/4

SysClk/8

SysClk/16

SysClk/32

SysClk/64

SysClk/128

SysClk/256

Data Order

This option determines how the serial data is shifted out, either LSb or MSb first.

Interrupt Generation Control

The following parameter is only accessible when the **Enable interrupt generation control** check box in PSoC Designer is checked. This is available under **Project > Settings > Chip Editor**.

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 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.

Here is a list of SPIM supplied API functions:

SPIM_Start

Description:

Sets the mode configuration of the SPI interface and enables the SPIM module by setting the proper bits in the Control register.

Before calling this function, assert all of the slave select signal(s) high to deselect connected SPI Slave devices. Do this in a user supplied routine.

C Prototype:

```
void  SPIM_Start(BYTE bConfiguration)
```

Assembly:

```
mov   A, SPIM_SPIM_MODE_2 | SPIM_SPIM_LSB_FIRST
lcall  SPIM_Start
```

Parameters:

bConfiguration: One byte that specifies the SPI mode and LSB First configurations. It is passed in the Accumulator. Symbolic names provided in C and assembly, and their associated values, are given in the following table. Note that the symbolic names can be OR'ed together to form the configuration of the SPI interface.Also note that the instance name of the user module is prepended to the symbolic name listed in the following table. For example, if you named the user module SPIM1 when you placed it, the symbolic name of the first mode is SPIM1_SPIM_MODE_0.

Symbolic Name	Value
SPIM_SPIM_MODE_0	0x00
SPIM_SPIM_MODE_1	0x02
SPIM_SPIM_MODE_2	0x04
SPIM_SPIM_MODE_3	0x06
SPIM_SPIM_LSB_FIRST	0x80
SPIM_SPIM_MSB_FIRST	0x00

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_Stop

Description:

Disables the SPIM module by clearing the enable bit in the Control register.

After a call to this function is made, assert all of the slave select signal(s) high to disable all of the connected SPI Slave devices. Do this in a user supplied routine.

C Prototype:

```
void  SPIM_Stop(void)
```

Assembly:

```
lcall  SPIM_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_EnableInt

Description:

Enables the SPIM interrupt on the SPI Done condition. The placement location of the SPIM determines the specific interrupt vector and priority.

C Prototype:

```
void  SPIM_EnableInt(void)
```

Assembly:

```
lcall  SPIM_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_DisableInt

Description:

Disables the SPIM interrupt on the SPI Done condition.

C Prototype:

```
void  SPIM_DisableInt(void)
```

Assembly:

```
lcall  SPIM_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_SendTxData

Description:

Initiates the SPI transmission to a slave SPI device.

Before this call, assert the specified SPI slave device's signal low. Do this in a user supplied routine.

C Prototype:

```
BOOL  SPIM_SendTxData(BYTE bSPIMData)
```

Assembly:

```
mov   A, bSPIMData
lcall  SPIM_SendTxData
```

Parameters:

BYTE bSPIMData: Data to send to the SPI slave device. It is passed in Accumulator.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_bReadRxData

Description:

Returns a received data byte from a slave device. Check the Rx Buffer Full flag before calling this routine, to verify that a data byte was received.

C Prototype:

```
BYTE  SPIM_bReadRxData(void)
```

Assembly:

```
lcall  SPIM_bReadRxData
mov   bRxData, A
```

Parameters:

None

Return Value:

Data byte received from the slave SPI and returned in the Accumulator.

Side Effects:

The A and X registers may be altered by this function.

SPIM_bReadStatus

Description:

Reads and returns the current SPIM Control/Status register.

C Prototype:

```
BYTE  SPIM_bReadStatus(void)
```

Assembly:

```
lcall  SPIM_bReadStatus
and   A, SPIM_SPIM_SPI_COMPLETE | SPIM_SPIM_RX_BUFFER_FULL
jnz   SpimCompleteGetRxData
```

Parameters:

None

Return Value:

Returns status byte read and is returned in the Accumulator.

Use defined masks to test for specific status conditions. Note that masks can be OR'ed together to test for multiple conditions. Also note that the instance name of the user module is prepended to the symbolic name listed in the following table. For example, if you named the user module SPIM1 when you placed it, the symbolic name of the first mask is SPIM1_SPIM_SPI_COMPLETE.

SPIM Status Masks	Value
SPIM_SPIM_SPI_COMPLETE	0x20
SPIM_SPIM_RX_OVERRUN_ERROR	0x40
SPIM_SPIM_TX_BUFFER_EMPTY	0x10
SPIM_SPIM_RX_BUFFER_FULL	0x08

Side Effects:

The status bits are cleared after this function is called. The A and X registers may be altered by this function.

Sample Firmware Source Code

In the following C and Assembly sample code, The SPI Master transmits a zero terminated string stored in RAM. The function loads bytes, one at a time, into the Digital PSoC block TX register using the SPIM API. The I2C/SPI block begins transmission by transferring its TX register to its Shift register. The digital block begins transmission by transferring its TX register to its Shift register. Repeating the API call, this function immediately reloads the transmit register with the next byte. This way, the transmission proceeds at the maximum continuous rate. In this simple version, any data received from this slave is discarded. This function returns just after the final byte is loaded in the TX register and while the next to last byte is still shifting out on the MOSI line. Subsequent calls to this function do not corrupt an on going transmission, since the status is always checked before loading a value into the TX register.

```
#include <m8c.h>        // part specific constants and macros
```

```
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

CHAR Message[] = "Hello World.";
CHAR *pbStrPtr = Message;

void main(void)
{
    SPIM_Start(SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST);

while( *pbStrPtr != 0 )  /* While data remains to be sent */
    {
/* Ensure the transmit buffer is free */
while( ! (SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY ) );

SPIM_SendTxData( *pbStrPtr );  /* load the next byte */
pbStrPtr++;
    }
}
```

The equivalent code, written in Assembly, is:

```
include "m8c.inc"          ; part specific constants and macros
include "memory.inc"    ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"    ; PSoC API definitions for all User Modules

export _main
export Message

AREA text (ROM, REL)

.LITERAL
Message:
    ASCIZ "Hello World."
.ENDLITERAL

_main:
    mov   A, SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST
lcall SPIM_Start           ;   Initialize the digital PSoC Block
    ; [...]                 ;   Set SlaveSelect signal low to enable slave
    mov   X, 0              ;   Set index to beginning of string
.TxNextByteLoop:
mov   A, X
    index Message
    jz    .Finish           ;    No, bail out of the loop.
push  A
.WaitForTxEmpty:
lcall SPIM_bReadStatus     ;   Transmit buffer free? ...
and   A, SPIM_SPIM_TX_BUFFER_EMPTY
jz    .WaitForTxEmpty      ;   No,  keep checking the status
pop   A
    lcall SPIM_SendTxData  ;   Yes, load TX with the next byte, ...
inc   X                    ;       Advance the pointer, ...
jmp   .TxNextByteLoop      ;        and repeat until done.
```

```
.Finish:                        ;   Transmit complete!
lcall SPIM_bReadStatus      ;   Buffer empty for last time? ...
    and   A, SPIM_SPIM_TX_BUFFER_EMPTY
jz    .Finish               ;   No,  keep checking the status
.WaitForTxComplete:
    lcall SPIM_bReadStatus      ;   Last byte transmission complete? ...
and   A, SPIM_SPIM_SPI_COMPLETE
jz    .WaitForTxComplete    ;   No,  keep checking the status
; [...]                     ;   (Reset SlaveSelect signal back high)
.AllDone:
    ; Endless loop
    jmp   .AllDone
```

Configuration Registers

The I2C/SPI PSoC block registers used to configure this User Module are described here. Only the parameterized symbols are explained.

Table 3. Block SPIM, Configuration Register

Bit	7	6	5	4	3	2	1	0
Value	Clock Sel			Bypass	SS_	SS_EN_	Int Sel	Slave

Clock Sel - Clock Selection. These bits determine the operating frequency of the SPI Master.

000b - SysClk / 2

001b - SysClk / 4

010b - SysClk / 8

011b - SysClk / 16

100b - SysClk / 32

101b - SysClk / 64

110b - SysClk / 128

111b - SysClk / 256

Bypass - Bypass Synchronization. This bit determines if the inputs are synchronized to SYSCLK.

SS_ - Slave Select. This bit determines the logic value of the SS_ signal when the SS_EN_ signal is asserted (SS_EN_ = 0).

SS_EN_ - Slave Select Enable. This active low bit determines if the slave select (SS_) signal is driven internally. If it is driven internally, its logic level is determined by the SS_ bit. If it is driven externally, its logic level is determined by the external pin.

Int Sel - Interrupt Select. This bit selects which condition produces an interrupt, whether it is based off of the TX Reg Empty condition or the SPI Complete condition.

Slave - this bit determines whether the block functions as a master or slave.

Table 4. Block SPIM, TX Data Buffer Register

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

TX Buffer Register: data written to this buffer is transferred to the Shift register when you enable the PSoC block.

Table 5. Block SPIM, RX Data Buffer Register

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

RX Buffer Register: data received in the Shift register is transferred to this register after completion of the SPI transmit cycle.

Table 6. Block SPIM, Control Register

Bit	7	6	5	4	3	2	1	0
Value	LSB First	RX Overrun Error	SPI Done	TX Buffer Empty	RX Buffer Full	Clock Phase	Clock Polarity	Enable

LSB First - specifies that the LSB bit should be transmitted first.

RX Overrun Error - a flag that indicates that the previously received data byte was not read before the next byte was received.

SPI Done - a flag that indicates that the complete SPI transmit/receive cycle is complete.

TX Buffer Empty - a flag that indicates that the TX buffer is empty.

RX Buffer Full - a flag that indicates that a byte of data was received from the Shift register.

Clock Phase - indicates the phase of the SCLK signal. It is one of the parameters that defines the SPI mode.

Clock Polarity - indicates the polarity of the SCLK signal. It is one of the parameters that defines the SPI mode.

Enable - enables the SPI PSoC block when set.

Version History

Version	Originator	Description
2.5	DHA	P1[0] was set as SPI CLK pin for CY7C64315 and CY7C64316 PSoC devices.
3.00	DHA	1. Added note to ensure that the clock polarity and phase for the selected mode match the settings used for any attached SPI device. 2. Corrected Drive Mode configuration for the SPI pins for CY7C64315 devices.
3.00.b	HPHA	Added AN51234 reference in Getting Started section.

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 © 2002-2013 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.