PSoC® Board Support Package for CY8CKIT-042 (PSoC 4 Pioneer Kit)

CapSense- and IDAC-enabled Configurations

v1.0

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intl): 408.943.2600
http://www.cypress.com

November 2013 YFS

Copyrights

Disclaimer

Flash Code Protection

Cypress products meet the specifications contained in their particular Cypress PSoC Datasheets. Cypress believes that its family of PSoC products is one of the most secure families of its kind on the market today, regardless of how they are used. There may be methods, unknown to Cypress, that can breach the code protection features. Any of these methods, to our knowledge, would be dishonest and possibly illegal. Neither Cypress nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Cypress is willing to work with the customer who is concerned about the integrity of their code. Code protection is constantly evolving. We at Cypress are committed to continuously improving the code protection features of our products.

Table of Contents

Chapters

Tables

Figures

Supported Design Configurations

This document supports two similar configurations of the PSoC 4 device. Both designs include a core set of functions such as PWMs, a Timer, Analog Comparator and ADC. The difference is restricted to the availability of CapSense or the current DAC (IDAC).

The selection of the CapSense- or IDAC-enabled design impacts the device resources used and the pin choices. These differences are clearly highlighted within the document.

Device Family Overview

The Cypress PSoC 4 is a family of 32-bit devices with the following characteristics.

- High-performance 32-bit ARM Cortex-M0 core with a nested vectored interrupt controller (NVIC)
- Digital system that includes configurable Universal Digital Blocks (UDBs) and specific function peripherals, such as UART, SPI and I2C
- Analog subsystem that includes 12-bit SAR ADC, PWMs, comparators, op amps, CapSense, LCD drive and more
- Several types of memory elements, including SRAM and flash
- Programming and debug system through Serial Wire Debug (SWD)
- Flexible routing to all pins

Figure 1 shows the major components of a typical CY8C42 family member PSoC 4 device.

Figure 1: CY8C42 Device Family Block Diagram

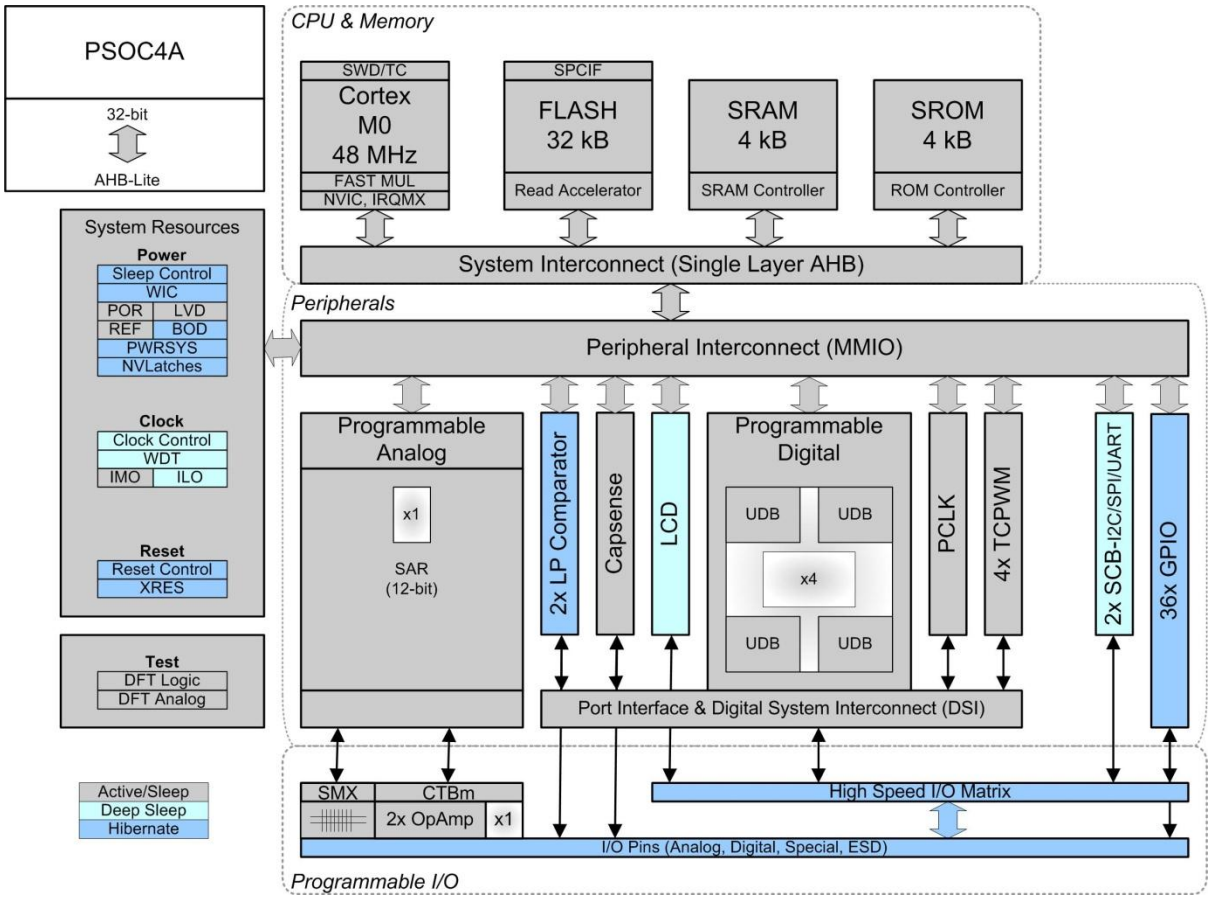


Table 1 lists the key characteristics of this device.

Table 1: Device Characteristics

Name	Value
Device	CY8C4245AXI-483
Architecture	PSoC 4
Family	CY8C42
CPU speed (MHz)	48
Flash size (kBytes)	32
SRAM size (kBytes)	4
Vdd range (V)	1.71 to 5.5
Automotive qualified	No (Industrial Grade Only)

NOTE: The CPU speed noted above is the maximum allowed speed. The CPU is clocked by HFCLK, which is listed in the System Clocks section below (and can be changed by modifying the project in PSoC Creator).

Resources

This design is intended to function like a relatively simple microcontroller, with only a little PSoC-specific functionality (e.g. the hardware-based switch debouncer and the control register being used to drive muxes and inputs to the Timer). As a result not all resources have been used. This is deliberate because it leaves “head-room” for the reader to modify the design to suit their unique needs (using PSoC Creator).

Table 2: Device Resources

	Resources Used		
Name	**CapSense-enabled**	**IDAC-enabled**	**Total Available**
Digital clock dividers	1 (25.0%)	1 (25.0%)	4
Pins	35 (97.2%)	34 (94.4%)	36
UDB Macrocells	7 (21.9%)	7 (21.9%)	32
UDB Unique Pterms	10 (15.6%)	10 (15.6%)	64
UDB Datapath Cells	0 (0.0%)	0 (0.0%)	4
UDB Status Cells	0 (0.0%)	0 (0.0%)	4
UDB Control Cells	1 (25.0%)	1 (25.0%)	4
Interrupts	13 (40.6%)	12 (37.5%)	32
Comparator/Opamp Fixed Blocks	1 (50.0%)	1 (50.0%)	2
SAR Fixed Blocks	1 (100.0%)	1 (100.0%)	1
CSD Fixed Blocks	1 (100.0%)	0 (0.0%)	1
8-bit CapSense IDACs	1 (100.0%)	1 (100.0%)	1
7-bit CapSense IDACs	1 (100.0%)	0 (0.0%)	1
Temperature Sensor	0 (0.0%)	0 (0.0%)	1
Low Power Comparator	1 (50.0%)	1 (50.0%)	2
TCPWM Blocks	4 (100.0%)	4 (100.0%)	4
Serial Communication Blocks	2 (100.0%)	2 (100.0%)	2
Segment LCD Blocks	0 (0.0%)	0 (0.0%)	1

System Settings

The following tables show system settings as configured in PSoC Creator.

Configuration

Table 3: Configuration Settings

Name	Value
Device Configuration Mode	Compressed
Unused Bonded IO	Allow but warn
Heap Size (bytes)	256
Stack Size (bytes)	1024
Include CMSIS Core Peripheral Library Files	True

Debug

Table 4: Debug Settings

Name	Value
Chip Protection	Open
Debug Select	SWD (serial wire debug)

Operating Conditions

Table 5: Operating Conditions

Name	Value
Vddd (V)	3.3
Vdda (V)	3.3
Variable Vdda	True
Temperature Range	-40C - 85C

Pins

PSoC devices include flexible internal routing so that the user can pick the perfect pin mapping. This diagram shows a pinout chosen to suit the Pioneer board. The user can modify the pin selection from PSoC Creator.

Figure 2: Device Pin Layout - CapSense-enabled

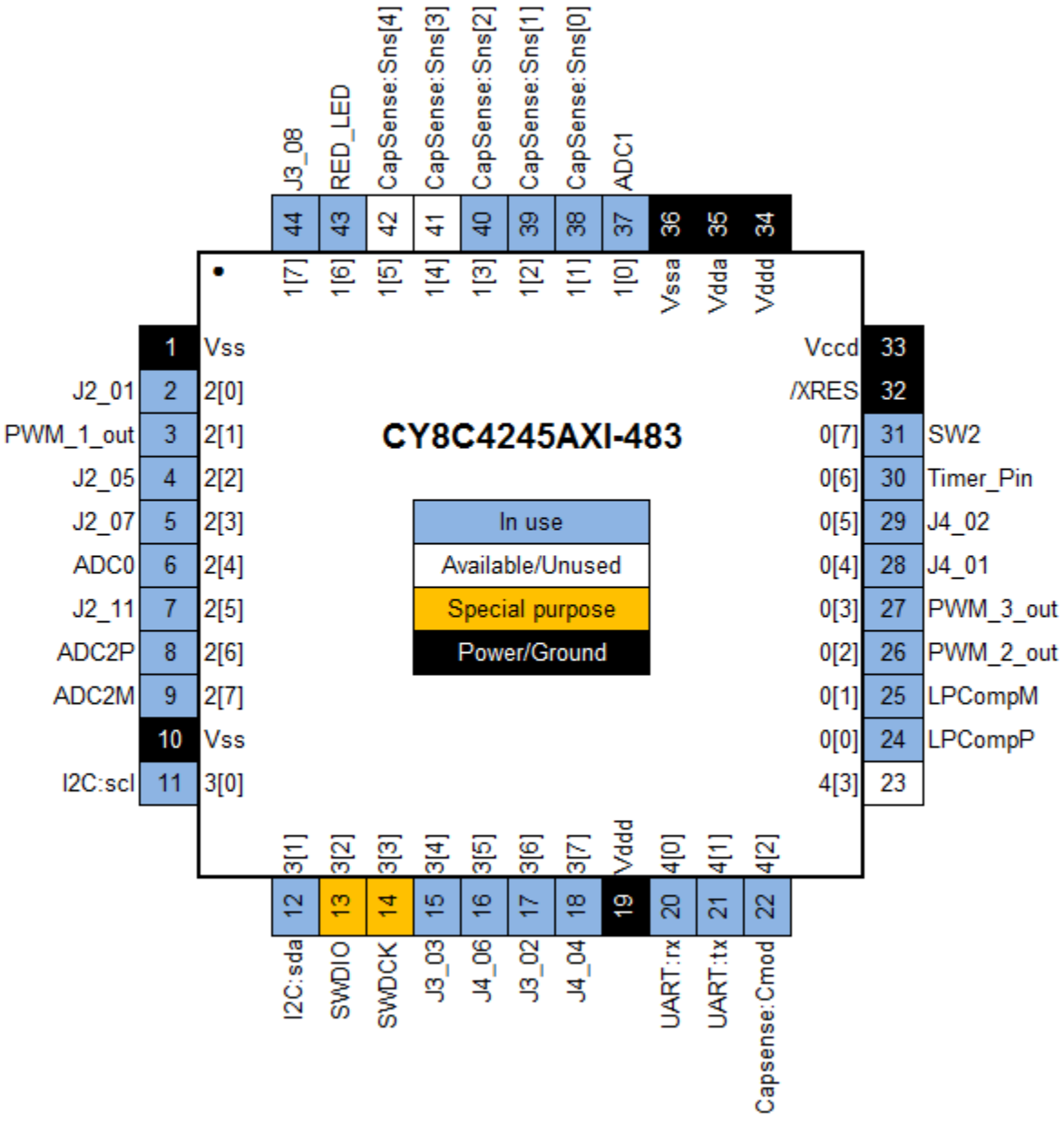
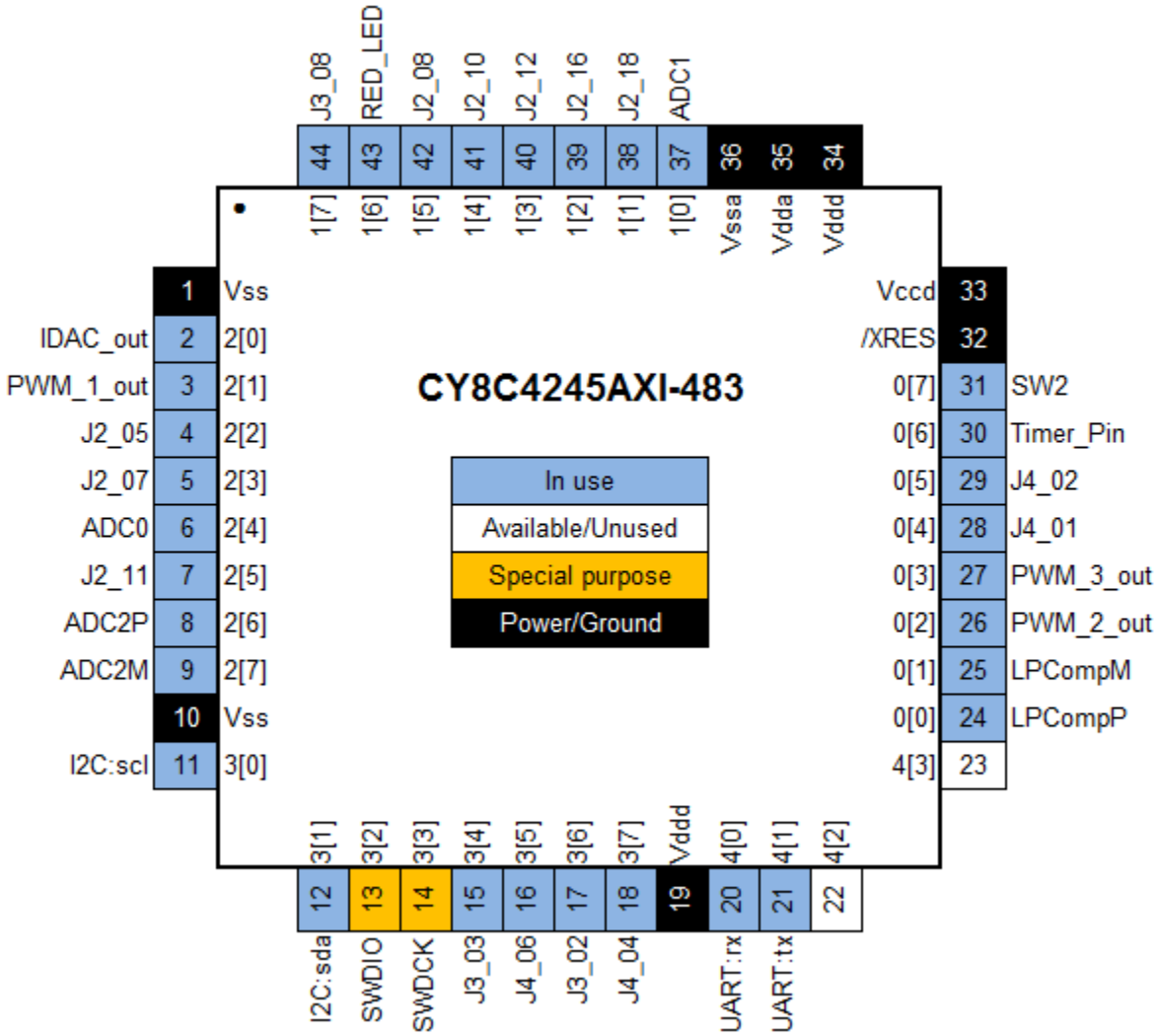


Figure 3: Device Pin Layout - IDAC-enabled



Device Pin Functions

This table lists all pins with their function and configuration. Note that some pins have different uses in the CapSense- and IDAC-enabled designs.

Table 6: Device Pins

Pin	Port	Name	Type	Drive Mode
1	VSS	VSS	Power	
2	P2[0]	Use is dependent upon design (CapSense or IDAC) – see figures below		
3	P2[1]	PWM_1_out	Digital Out	Strong drive
4	P2[2]	J2_05	Software	Strong drive
5	P2[3]	J2_07	Software	Strong drive
6	P2[4]	ADC0	Analog	HiZ analog
7	P2[5]	J2_11	Software	Strong drive
8	P2[6]	ADC2P	Analog	HiZ analog
9	P2[7]	ADC2M	Analog	HiZ analog
10	VSS	VSS	Power	
11	P3[0]	I2C:scl	Digital In	OD, DL
12	P3[1]	I2C:sda	Digital In	OD, DL
13	P3[2]	GPIO [unused]		
14	P3[3]	GPIO [unused]		
15	P3[4]	J3_03	Software	Strong drive
16	P3[5]	J4_06	Software	Strong drive
17	P3[6]	J3_02	Software	Strong drive
18	P3[7]	J4_04	Software	Strong drive
19	VDDD	VDDD	Power	
20	P4[0]	UART:rx	Digital In	HiZ digital
21	P4[1]	UART:tx	Digital Out	Strong
22	P4[2]	Use is dependent upon design (CapSense or IDAC) – see figures below		
23	P4[3]	GPIO [unused]		
24	P0[0]	LPCompP	Analog	HiZ analog
25	P0[1]	LPCompM	Analog	HiZ analog
26	P0[2]	PWM_2_out	Digital Out	Strong
27	P0[3]	PWM_3_out	Digital Out	Strong
28	P0[4]	J4_01	Software	Strong

Pin	Port	Name	Type	Drive Mode
29	P0[5]	J4_02	Software	Strong
30	P0[6]	Timer_Pin	Digital In	Res pull dn
31	P0[7]	SW2	Digital In	Res pull up
32	XRES	XRES	Power	
33	VCCD	VCCD	Power	
34	VDDD	VDDD	Power	
35	VDDA	VDDA	Power	
36	VSSA	VSSA	Power	
37	P1[0]	ADC1	Analog	HiZ analog
38	P1[1]	Use is dependent upon design (CapSense or IDAC) – see figures below		
39	P1[2]	Use is dependent upon design (CapSense or IDAC) – see figures below		
40	P1[3]	Use is dependent upon design (CapSense or IDAC) – see figures below		
41	P1[4]	Use is dependent upon design (CapSense or IDAC) – see figures below		
42	P1[5]	Use is dependent upon design (CapSense or IDAC) – see figures below		
43	P1[6]	RED_LED	Digital Out	Strong drive
44	P1[7]	J3_08	Software	Strong drive

Figure 4: Device Pins continued (CapSense-enabled)

Pin	Port	Name	Type	Drive Mode
2	P2[0]	J2_01	Software	Strong drive
22	P4[2]	CapSense:Cmod	A/D Out	HiZ analog
38	P1[1]	CapSense:Sns[0]	Analog	HiZ analog
39	P1[2]	CapSense:Sns[0]	Analog	HiZ analog
40	P1[3]	CapSense:Sns[0]	Analog	HiZ analog
41	P1[4]	CapSense:Sns[0]	Analog	HiZ analog
42	P1[5]	CapSense:Sns[0]	Analog	HiZ analog

Figure 5: Device Pins continued (IDAC-enabled)

Pin	Port	Name	Type	Drive Mode
2	P2[0]	IDAC_out	A/D Out	HiZ analog
22	P4[2]	GPIO [unused]		
38	P1[1]	J2_18	Software	Strong drive
39	P1[2]	J2_16	Software	Strong drive

Pin	Port	Name	Type	Drive Mode
40	P1[3]	J2_12	Software	Strong drive
41	P1[4]	J2_10	Software	Strong drive
42	P1[5]	J2_08	Software	Strong drive

The reset state of all non-power pins is high impedance analog unbuffered.

Abbreviations used in Table 6 have the following meanings:

- GPIO = General Purpose IO
- SIO = Special IO
- Res pull up = Resistive pull up
- Res pull dn = Resistive pull down
- Strong = Strong drive (digital output)
- HiZ analog = High impedance analog
- OD, DL = Open drain, drives low
- OD, DH = Open drain, drives high

For more information on reading, writing and configuring pins please refer to:

- Application Programming Interface section in the cy_pins component datasheet

Using the cy_pins Component

Some pins are dedicated to hardware functions, like the UART or ADC, and others are accessible from software. Those named after the header to which they are connected, e.g. J2_05, are software pins. Those named after their associated hardware function should (typically) not be accessed using the software APIs. It is not recommended to change the behavior of these pins from software or to attempt to read/write them.

Important: Note that the string "*Jx_yy*" in the following APIs should be replaced with the name of the cy_pins component listed in Table 6, e.g. J2_05_Read().

Table 7: cy_pins APIs

Function	Description
*Jx_yy*_Read()	Reads the physical port and returns the current value for all pins in the component
*Jx_yy*_Write()	Writes the value to the component pins while protecting other pins in the physical port if shared by multiple Pins components
*Jx_yy*_ReadDataReg()	Reads the current value of the port's data output register and returns the current value for all pins in the component
*Jx_yy*_SetDriveMode()	Sets the drive mode for each of the Pins component's pins

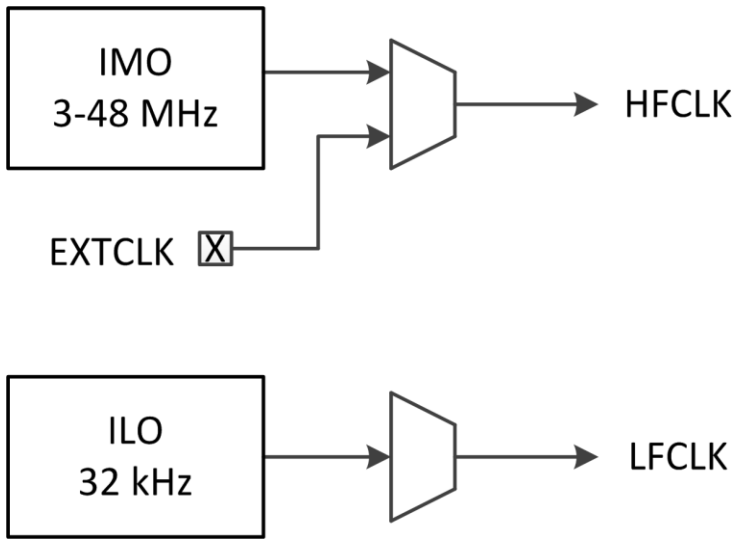
Function	Description
*Jx_yy*_ClearInterrupt()	Clears any active interrupts on the port into which the component is mapped. Returns value of interrupt status register

Clocks

The clock system includes these clock resources.

- Two internal clock sources:
    - 3 to 48 MHz Internal Main Oscillator (IMO) ±2% at 3 MHz
    - 32 kHz Internal Low Speed Oscillator (ILO) output
- HFCLK can be generated using an external signal from EXTCLK pin
- Twelve clock dividers, each with 16-bit divide capability:
    - Eight can be used for fixed-function blocks
    - Four can be used for the UDBs

Figure 6: System Clock Configuration



System Clocks

System clocks are sources that are available from the PSoC 4 clock block. These clocks are used to create clocks used in the actual design.

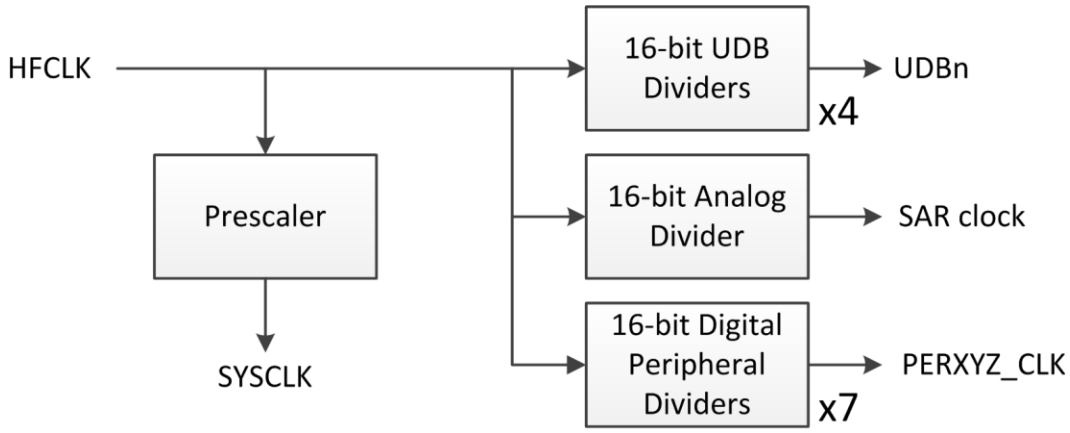
Table 8: System Clocks

Name	Domain	Source	Freq (MHz)	Accuracy (%)	Start at Reset
LFCLK	NONE	ILO	0.032	±30	True
ILO	NONE		0.032	±30	True
SYSCLK	NONE	HFCLK	48	±2	True
EXTCLK	NONE		0	±0	False
IMO	NONE		48	±2	True
HFCLK	NONE	Direct_Sel	48	±2	True

Local Clocks

Local clocks are derived from system clocks and drive individual analog and digital components.

Figure 7: Local Clock Configuration



Note that the two CapSense clocks in the following table only exist in the CapSense-enabled design.

Table 9: Local Clocks

Name	Domain	Source	Freq (MHz)	Accuracy (%)	Start at Reset
ADC_intClock	FIXED_FUNCTION	HFCLK	2.1818	±2	True
UART_SCBCLK	FIXED_FUNCTION	HFCLK	0.1151	±2	True
I2C_SCBCLK	FIXED_FUNCTION	HFCLK	1.6	±2	True
PWM_Clock	FIXED_FUNCTION	HFCLK	1	±2	True
Debounce_Clock	DIGITAL	HFCLK	0.0001	±2	True
Timer_Clock	FIXED_FUNCTION	HFCLK	0.001	±2	True
CapSense_SenseClk	FIXED_FUNCTION	HFCLK	1	±2	True
CapSense_SampleClk	FIXED_FUNCTION	HFCLK	24	±2	True

For more information on clocking resources, please refer to:

- Clocking System chapter in the PSoC 4 Technical Reference Manual
- Clocking chapter in the System Reference Guide
    - CyIMO API routines
    - CyILO API routines

Using the cy_clock Component

The local clocks listed in Table 9 can be managed from software using the following APIs. Note that changing the ADC or I2C clocks is very likely to have unintended consequences on component behavior, and is not recommended.

Important: Note that the string "*Clockname*" in the following APIs should be replaced with the name of the cy_clock component listed in the Design Contents chapters, e.g. PWM_Clock_Start().

Table 10: cy_clock APIs

Function	Description
*Clock*_Start()	Enables the clock.
*Clock*_Stop()	Disables the clock.
*Clock*_SetDivider()	Sets the divider of the clock and restarts the clock divider immediately.
*Clock*_SetDividerValue()	Sets the divider of the clock and restarts the clock divider immediately.
*Clock*_GetDividerRegister()	Gets the clock divider register value.
*Clock*_SetFractionalDividerRegister()	Sets the fractional divider of the clock and restarts the clock divider immediately.
*Clock*_GetFractionalDividerRegister()	Gets the fractional clock divider register value.

Interrupts

This design contains the following interrupt components: (0 is the highest priority).

Note that some components, such as I2C, include “buried” interrupts. These are not shown in the Design Contents section below and are typically enabled/disabled by the component Start/Stop APIs.

Note that the CapSense ISR in the following table only exists in the CapSense-enabled design.

Table 11: Interrupts

Name	Priority	Vector	Integrated into Component
ADC_EOC_Int	3	1	No
ADC_IRQ	3	14	Yes
I2C_SCB_IRQ	3	11	Yes
LPComp_Active_Int	3	2	No
LPComp_Wakeup_Int	3	8	No
PWM_1_Int	3	19	No
PWM_2_Int	3	17	No
PWM_3_Int	3	18	No
SW2_Active_Int	3	3	No
SW2_Wakeup_Int	3	0	No
Timer_Int	3	16	No
UART_Int	3	10	No
CapSense_ISR	3	15	Yes

For more information on interrupts, please refer to:

- Interrupt Controller chapter in the PSoC 4 Technical Reference Manual
- Interrupts chapter in the System Reference Guide
    - CyInt API routines and related registers
- Datasheet for cy_isr component

Global Interrupt Control

The following macros enable/disable all interrupts in the system. Note that some RTOS implementations enable interrupts in the OS startup code.

- CyGlobalIntEnable
- CyGlobalIntDisable

Note that these macros do not require trailing parentheses (i.e. the instruction "CyGlobalIntEnable;" will enable interrupts).

Using the cy_isr Component

The interrupts listed in Table 11 can be managed from software. In some cases the interrupts have been integrated into the component that uses them (e.g. I2C) and in others they are independent (e.g. SW2_Interrupt). The integrated interrupts are handled by the component APIs and do not need to be set up and enabled in application code. The independent interrupts (marked "No" in the "Integrated into Component" column of Table 11) are visible in the schematic sheets below and need to be set up by the user (if required).

The following macros are useful for creating ISR routines that are compatible with the cy_isr APIs. These macros provide consistent definition of interrupt service routines across compilers and platforms. Note that the macro to use is different between the function definition and the function prototype.

Function prototype example:

```
CY_ISR_PROTO(MyISR);
```

Function definition example:

```
CY_ISR(MyISR)
{
    /* ISR Code here */
}
```

ISRs that are declared and defined in this way can be installed using the *ISRname*_StartEx() API below.

Important: Note that the string "*ISRname*" in the following APIs should be replaced with the name of the cy_isr component listed in Table 11, e.g. SW2_Active_Int_StartEx().

Table 12: cy_isr APIs

Function	Description
*ISRname*_Start()	Sets up the interrupt to function.
*ISRname*_StartEx()	Sets up the interrupt to function and sets address as the ISR vector for the interrupt.
*ISRname*_Stop()	Disables and removes the interrupt.
*ISRname*_Interrupt()	The default interrupt handler for ISR.
*ISRname*_SetVector()	Sets address as the new ISR vector for the Interrupt.
*ISRname*_GetVector()	Gets the address of the current ISR vector for the interrupt.
*ISRname*_SetPriority()	Sets the priority of the interrupt.
*ISRname*_GetPriority()	Gets the priority of the interrupt.
*ISRname*_Enable()	Enables the interrupt to the interrupt controller.
*ISRname*_GetState()	Gets the state (enabled, disabled) of the interrupt.

Function	Description
*ISRname*_Disable()	Disables the interrupt.
*ISRname*_SetPending()	Causes the interrupt to enter the pending state, a software method of generating the interrupt.
*ISRname*_ClearPending()	Clears a pending interrupt.

Flash Memory

PSoC 4 devices offer a host of Flash protection options and device security features that you can leverage to meet the security and protection requirements of an application. These requirements range from protecting configuration settings or Flash data to locking the entire device from external access.

Table 13: Flash Protection Settings

Start Address	End Address	Protection Level
0x0	0x7FFF	U - Unprotected

Flash memory is organized as rows with each row of flash having 128 bytes. Each flash row can be assigned one of four protection levels:

- U - Unprotected
- F - External read protect (Factory upgrade)
- R - External write protect (Field upgrade)
- W - Full Protection

For more information on Flash memory and protection, please refer to:

- Flash Protection chapter in the PSoC 4 Technical Reference Manual
- Flash and EEPROM chapter in the System Reference Guide
    - CyFlash API routines
    - CyWrite API routines

Design Contents

This chapter describes how the PSoC was configured. You may change, add to, or delete from, this configuration by editing the *TopDesign.cycsh* file in PSoC Creator.

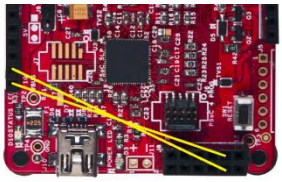
This design's schematic content consists of the following schematic sheets.

UART

The UART supports two-way serial communication through a pair of Tx and Rx pins (P4[1:0]). These pins are connected to J3_09 (Rx) and J3_10 (Tx) on the Pioneer kit.

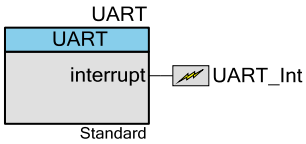
The UART can also be accessed from a PC running a terminal emulator, through the PSoC 5LP USB-UART bridge. This is described in detail in the Pioneer Kit Guide. Note that the kit guide expects the UART pins to be on P0[5:4] but this design uses P4[1:0]. This means that the wiring is different. To enable the bridge with this design make the following connections.

- UART:rx – J3_09 to J8_10
- UART:tx – J3_10 to J8_09



The UART is implemented in a hardware block called an SCB – Serial Communication Block. An SCB can be configured as a UART, I2C or SPI using PSoC Creator. Note that some APIs are rather long, and refer to the different uses of the block, in order to support all modes of the SCB hardware.

Figure 8: UART Schematic



This schematic contains the following component instances (click links for details on features and APIs).

- Instance UART (type: SCB_P4_v1_10)
- Instance UART_Int (type: cy_isr_v1_70)
    - See the chapter on Using the cy_isr Component for API information.

Table 14: UART Parameters

Parameter Name	Value	Description
Data Rate	9600	UART baud rate in kbps.
Direction	TX + RX	Enables RX and TX direction.
Data Bits	8 bits	Number of data bits inside the UART byte/word.
Parity	None	Parity check as Odd or Even or none.

Parameter Name	Value	Description
Stop Bits	1 bit	Number of Stop bits.
Rx Buffer Size	8	Size of the RX buffer. The value 8 implies the usage of hardware RX FIFO. Greater values imply usage of internal software buffer along with RX FIFO.
Rx Trigger Level	7	Number of entries in the RX FIFO to trigger the SCB.INTR_RX.TRIGGER interrupt event.
Tx Buffer Size	8	Size of the TX buffer. The value 8 implies usage of hardware TX FIFO. Greater values imply the usage of internal software buffer along with TX FIFO.
Tx Trigger Level	0	Number of entries in the TX FIFO to trigger the SCB.INTR_TX.TRIGGER interrupt event.
Wake Enable	false	Enables the wakeup from Deep Sleep on start bit event. The actual wakeup source is RX GPIO. The skip start UART feature allows it to continue receiving bytes.
Drop On Frame Err	false	Determines whether the data is dropped from RX FIFO on a frame error event.
Drop On Parity Err	false	Determines whether the data is dropped from RX FIFO on a parity error event.
Oversampling Factor	12	Oversampling factor of SCBCLK.
Median Filter Enable	false	Applies a digital 3 tap median filter to the UART input line.

Table 15: UART Interrupt Sources

Parameter Name	Value	Description
Rx Frame Err	true	SCB.INTR_RX.FRAME_ERROR: frame error in received data frame.
Rx Full	true	SCB.INTR_RX.FULL: RX FIFO is full.
Rx Not Empty	true	SCB.INTR_RX.NOT_EMPTY: RX FIFO is not empty. There is at least one entry to get data from.
Rx Overflow	true	SCB.INTR_RX.OVERFLOW: attempt to write to a full RX FIFO.
Rx Parity Err	true	SCB.INTR_RX.PARITY_ERROR: parity error in received data frame.
Rx Trigger	false	SCB.INTR_RX.TRIGGER: RX FIFO has more entries than the value specified by UartRxTriggerLevel.
Rx Underflow	true	SCB.INTR_RX.UNDERFLOW: attempt to read from an empty RX FIFO.
Tx Empty	true	SCB.INTR_TX.EMPTY: TX FIFO is empty.

Parameter Name	Value	Description
Tx Not Full	true	SCB.INTR_TX.NOT_FULL: TX FIFO is not full. There is at least one entry to put data.
Tx Overflow	true	SCB.INTR_TX.OVERFLOW: attempt to write to a full TX FIFO.
Tx Trigger	false	SCB.INTR_TX.TRIGGER: TX FIFO has fewer entries than the value specified by UartTxTriggerLevel.
Tx Uart Done	true	SCB.INTR_TX.UART_DONE: all data are sent in to TX FIFO and the transmit FIFO and the shifter register are emptied.
Tx Underflow	true	SCB.INTR_TX.UNDERFLOW: attempt to read from an empty TX FIFO.

Table 16: UART APIs

Function	Description
UART_Init()	Initialize the SCB component according to defined parameters in the customizer.
UART_Enable()	Enables SCB component operation.
UART_Start()	Starts the SCB.
UART_Stop()	Disable the SCB component.
UART_Sleep()	Prepares component to enter Deep Sleep.
UART_Wakeup()	Prepares component to exit Deep Sleep.
UART_UartInit()	Configures the SCB for SPI operation.
UART_UartPutChar()	Places a byte of data in the transmit buffer to be sent at the next available bus time.
UART_UartPutString()	Places a NULL terminated string in the transmit buffer to be sent at the next available bus time.
UART_UartPutCRLF()	Places byte of data followed by a carriage return (0x0D) and line feed (0x0A) to the transmit buffer
UART_UartGetChar()	Retrieves next data element from receive buffer.
UART_UartGetByte()	Retrieves next data element from the receive buffer
UART_UartSetRxAddress()	Sets the hardware detectable receiver address for the UART in Multiprocessor mode.
UART_UartSetRxAddressMask()	Sets the hardware address mask for the UART in Multiprocessor mode.
UART_SpiUartWriteTxData()	Places a data entry into the transmit buffer to be sent at the next available bus time. This function is common for SPI and UART.
UART_SpiUartPutArray()	Places an array of data into the transmit buffer to be sent.

Function	Description
	This function is common for SPI and UART.
UART_SpiUartGetTxBufferSize()	Returns the number of elements currently in the transmit buffer. This function is common for SPI and UART.
UART_SpiUartClearTxBuffer()	Clears the transmit buffer and TX FIFO. This function is common for SPI and UART.
UART_SpiUartReadRxData()	Retrieves the next data element from the receive buffer. This function is common for SPI and UART.
UART_SpiUartGetRxBufferSize()	Returns the number of received data elements in the receive buffer. This function is common for SPI and UART
UART_SpiUartClearRxBuffer()	Clear the receive buffer and RX FIFO. This function is common for SPI and UART.

Table 17: UART Interrupt APIs

Function	Description
UART_EnableInt()	When using an Internal interrupt, this enables the interrupt in the NVIC.
UART_DisableInt()	When using an Internal interrupt, this disables the interrupt in the NVIC.
UART_GetInterruptCause()	Returns a mask of bits showing what the source of the current triggered interrupt.
UART_SetCustomInterruptHandler()	Registers a function to be called by the internal interrupt handler.
UART_SetTxInterruptMode()	Configures which bits of TX interrupt request register will trigger an interrupt event.
UART_GetTxInterruptMode()	Returns TX interrupt mask
UART_GetTxInterruptSourceMasked()	Returns TX interrupt request register masked by interrupt mask
UART_GetTxInterruptSource()	Returns the bit-mask of pending TX interrupt sources
UART_ClearTxInterruptSource()	Clears the bit-mask of pending TX interrupt sources
UART_SetTxInterrupt()	Generates interrupt event from bit-mask of TX interrupt sources
UART_SetRxInterruptMode()	Configures which bits of RX interrupt request register will trigger an interrupt event
UART_GetRxInterruptMode()	Returns RX interrupt mask
UART_GetRxInterruptSourceMasked()	Returns RX interrupt request register masked by interrupt mask
UART_GetRxInterruptSource()	Returns the bit-mask of pending RX interrupt sources
UART_ClearRxInterruptSource()	Clears the bit-mask of pending RX interrupt sources

Function	Description
UART_SetRxInterrupt()	Generates interrupt event from bit-mask of RX interrupt sources

Table 18: UART_Int Parameters

Parameter Name	Value	Description
Interrupt Type	LEVEL	IRQ source is sticky and remains active until firmware clears the source of the request with an action.

I2C

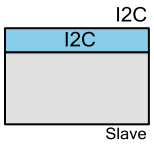
The I2C supports two-way serial communication through a pair data (sda) and clock (scl) pins (P3[1:0]). These pins are connected to J3_04 (scl) and J3_05 (sda) on the Pioneer kit.

The I2C protocol is implemented in ISRs that are installed and managed from the APIs below. These APIs maintain the content of read and write buffers so that I2C communication is possible with the minimum of user-provided firmware. You should install custom interrupt handlers if you wish to act on I2C events.

I2C can also be accessed from a PC running a terminal emulator, through the PSoC 5LP USB-UART bridge. This is described in detail in the Pioneer Kit Guide.

I2C is implemented in a hardware block called an SCB – Serial Communication Block. An SCB can be configured as a UART, I2C or SPI using PSoC Creator. Note that some APIs are rather long, and refer to the different uses of the block, in order to support all modes of the SCB hardware.

Figure 9: I2C Schematic



This schematic contains the following component instances (click links for details on features and APIs).

- Instance I2C (type: SCB_P4_v1_10)

Table 19: I2C Parameters

Parameter Name	Value	Description
Mode	Slave	Defines the I2C operation mode as: Slave, Master, Multi-Master or Multi-Master-Slave.
Data Rate	100	Data rate in kbps. The standard data rates are: 50, 100, 400 and 1000 kbps.
Slave Address	0x8	The 7-bit slave address (MSB ignored).

Parameter Name	Value	Description
Slave Address Mask	0xFE	Slave address mask. Bit value 0 – excludes bit from address comparison. Bit value 1 – the bit needs to match with the corresponding bit of the I2C slave address.
I2cWakeEnable	true	Enables wakeup from Deep Sleep on an I2C address match event.
Accept Address	false	Specifies whether to accept a match I2C slave address in the RX FIFO or not. This option could be used for software address matching.
Median Filter Enable	false	Applies a digital 3 tap median filter to the I2C lines.
Oversampling Factor	16	Oversampling factor of SCBCLK.

Table 20: I2C APIs

Function	Description
I2C_Init()	Initialize the SCB component according to defined parameters in the customizer.
I2C_Enable()	Enables the SCB component operation.
I2C_Start()	Starts the SCB component.
I2C_Stop()	Disable the SCB component.
I2C_Sleep()	Prepares the SCB component to enter Deep Sleep.
I2C_Wakeup()	Prepares the SCB component to exit Deep Sleep.
I2C_I2CInit()	Configures the SCB component for operation in I2C mode.
I2C_I2CSlaveStatus()	Returns slave status flags.
I2C_I2CSlaveClearReadStatus()	Returns read status flags and clears slave read status flags.
I2C_I2CSlaveClearWriteStatus()	Returns the write status and clears the slave write status flags.
I2C_I2CSlaveSetAddress()	Sets slave address, a value between 0 and 127 (0x00 to 0x7F).
I2C_I2CSlaveSetAddressMask()	Sets slave address mask, a value between 0 and 254 (0x00 to 0xFE).
I2C_I2CSlaveInitReadBuf()	Sets up the slave receive data buffer (master <- slave).
I2C_I2CSlaveInitWriteBuf()	Sets up the slave write buffer (master -> slave).
I2C_I2CSlaveGetReadBufSize()	Returns the number of bytes read by the master since I2C_I2CSlaveClearReadBuf() was called.
I2C_I2CSlaveGetWriteBufSize()	Returns the number of bytes written by the master since I2C_I2CSlaveClearWriteBuf() was called.
I2C_I2CSlaveClearReadBuf()	Resets the read buffer counter to zero.

Function	Description
I2C_I2CSlaveClearWriteBuf()	Resets the write buffer counter to zero.

Table 21: I2C Interrupt APIs

Function	Description
I2C_EnableInt()	When using an Internal interrupt, this enables the interrupt in the NVIC.
I2C_DisableInt()	When using an Internal interrupt, this disables the interrupt in the NVIC.
I2C_GetInterruptCause()	Returns a mask of bits showing what the source of the current triggered interrupt.
I2C_SetCustomInterruptHandler()	Registers a function to be called by the internal interrupt handler.
I2C_SetTxInterruptMode()	Configures which bits of TX interrupt request register will trigger an interrupt event.
I2C_GetTxInterruptMode()	Returns TX interrupt mask
I2C_GetTxInterruptSourceMasked()	Returns TX interrupt request register masked by interrupt mask
I2C_GetTxInterruptSource()	Returns the bit-mask of pending TX interrupt sources
I2C_ClearTxInterruptSource()	Clears the bit-mask of pending TX interrupt sources
I2C_SetTxInterrupt()	Generates interrupt event from bit-mask of TX interrupt sources
I2C_SetRxInterruptMode()	Configures which bits of RX interrupt request register will trigger an interrupt event
I2C_GetRxInterruptMode()	Returns RX interrupt mask
I2C_GetRxInterruptSourceMasked()	Returns RX interrupt request register masked by interrupt mask
I2C_GetRxInterruptSource()	Returns the bit-mask of pending RX interrupt sources
I2C_ClearRxInterruptSource()	Clears the bit-mask of pending RX interrupt sources
I2C_SetRxInterrupt()	Generates interrupt event from bit-mask of RX interrupt sources
I2C_SetMasterInterruptMode()	Configures which bits of Master interrupt request register will trigger an interrupt event
I2C_GetMasterInterruptMode()	Returns Master interrupt mask
I2C_GetMasterInterruptSourceMasked()	Returns Master interrupt request register masked by interrupt mask
I2C_GetMasterInterruptSource()	Returns the bit-mask of pending Master interrupt sources
I2C_ClearMasterInterruptSource()	Clears the bit-mask of pending Master interrupt sources

Function	Description
I2C_SetMasterInterrupt()	Generates interrupt event from bit-mask of Master interrupt sources
I2C_SetSlaveInterruptMode()	Configures which bits of Slave interrupt request register will trigger an interrupt event
I2C_GetSlaveInterruptMode()	Returns Slave interrupt mask
I2C_GetSlaveInterruptSourceMasked()	Returns Slave interrupt request register masked by interrupt mask
I2C_GetSlaveInterruptSource()	Returns the bit-mask of pending Slave interrupt sources
I2C_ClearSlaveInterruptSource()	Clears the bit-mask of pending Slave interrupt sources
I2C_SetSlaveInterrupt()	Generates interrupt event from bit-mask of Slave interrupt sources

PWMs

Three PWMs, all driven from the same 1MHz clock, are used to drive the RGB LED on the Pioneer kit. PWM_1 drives the red LED, PWM_2 the green LED and PWM_3 the blue LED.

In addition, the PWM outputs are routed to the kit headers. For PWM_2 and PWM_3 the kit routes the signal directly to both the LED and the header. For PWM_1 the kit does not route the signal to the header. To match the functionality to the other PWMs the signal is routed inside the PSoC device to another pin, which is available on a header.

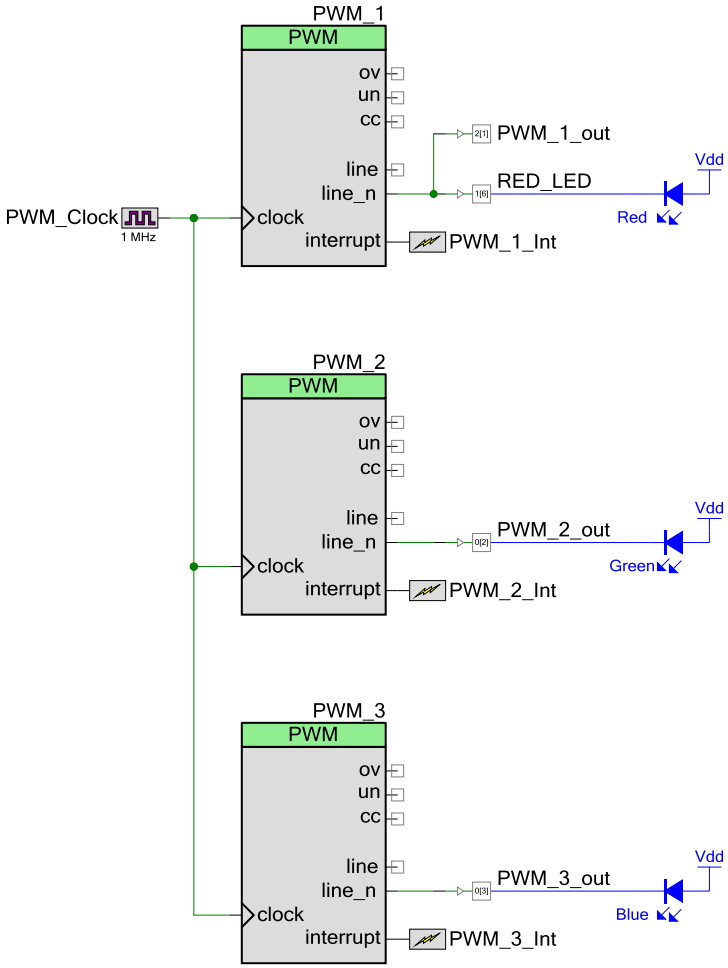
These are the pin mappings for the PWMs.

- RED_LED (from PWM_1) connects to P1[6], which is connected the red LED.
- PWM_1_out also connects to P2[1], which is routed to header J2_03.
- PWM_2_out connects to P0[2], which is routed to the green LED and header J2_02.
- PWM_3_out connects to P0[3], which is routed to the blue LED and header J2_04.

The PWMs are implemented in a hardware block called a TCPWM – Timer/Counter/PWM. A TCPWM can be configured as a Timer/Counter, PWM or Quadrature Decoder using PSoC Creator.

Note that, in the figure below the blue components are documentation. They show the (active low) connections of the LEDs on the Pioneer kit.

Figure 10: PWM Schematic



This schematic contains the following component instances (click links for details on features and APIs).

- Instance PWM_1, PWM_2 and PWM_3 (type: TCPWM_v1_0)
- Instance PWM_Clock (type: cy_clock_v2_10)
    - See the chapter on Using the cy_clock Component for API information.
- Instance RED_LED, PWM_1_out, PWM_2_out and PWM_3_out (type: cy_pins_v1_90)
    - See the chapter on Using the cy_pins Component for API information.
- Instance PWM_1_Int, PWM_2_Int and PWM_3_Int (type: cy_isr_v1_70)
    - See the chapter on Using the cy_isr Component for API information.

Table 22: PWM APIs

Function	Description
PWM_n_Init()	Initialize/Restore default TCPWM configuration
PWM_n_Enable()	Enables the TCPWM. TCPWM will be started if the Start terminal is not present
PWM_n_Start()	Initializes the TCPWM with default customizer values when called the first time and enables the TCPWM. TCPWM will be started if the Start terminal is not present
PWM_n_Stop()	Disables the TCPWM
PWM_n_SetMode()	Sets the operational mode of the TCPWM
PWM_n_SetPrescaler()	Sets the prescaler value that is applied to the clock input
PWM_n_TriggerCommand()	Triggers the designated command to occur on the designated TCPWM instances
PWM_n_SetPWMMode()	Writes the control register that determines what mode of operation the TCPWM output lines are driven in
PWM_n_SetPWMSyncKill()	Writes the register that controls whether the TCPWM kill signal (stop input) causes an asynchronous or synchronous kill operation
PWM_n_SetPWMStopOnKill()	Writes the register that controls whether the TCPWM kill signal (stop input) causes the TCPWM counter to stop
PWM_n_SetPWMInvert()	Writes the bits that control whether the line and line_n outputs are inverted from their normal output values
PWM_n_SetInterruptMode()	Sets the interrupt mask to control which interrupt requests generate the interrupt signal
PWM_n_GetInterruptSourceMasked()	Gets the interrupt requests masked by the interrupt mask
PWM_n_GetInterruptSource()	Gets the interrupt requests (without masking)
PWM_n_ClearInterrupt()	Clears the interrupt request
PWM_n_SetInterrupt()	Sets a software interrupt request
PWM_n_WriteCounter()	Writes a new 16 bit counter value directly into the counter register
PWM_n_ReadCounter()	Reads the current counter value
PWM_n_SetCounterMode()	Sets the counter mode
PWM_n_SetPeriodSwap()	Writes the register that controls whether the period registers are swapped
PWM_n_SetCompareSwap()	Writes the register that controls whether the compare registers are swapped
PWM_n_WritePeriod()	Writes the 16 bit period register with the new period value
PWM_n_ReadPeriod()	Reads the 16 bit period register

Function	Description
PWM_n_WritePeriodBuf()	Writes the 16 bit period buffer register with the new period value
PWM_n_ReadPeriodBuf()	Reads the 16 bit period buffer register
PWM_n_WriteCompare()	Writes the 16 bit compare register with the new compare value
PWM_n_ReadCompare()	Reads the compare register
PWM_n_WriteCompareBuf()	Writes the 16 bit compare buffer register with the new compare value
PWM_n_ReadCompareBuf()	Reads the compare buffer register
PWM_n_SetCaptureMode()	Sets the capture trigger mode
PWM_n_SetReloadMode()	Sets the reload trigger mode
PWM_n_SetStartMode()	Sets the start trigger mode
PWM_n_SetStopMode()	Sets the stop trigger mode
PWM_n_SetCountMode()	Sets the count trigger mode
PWM_n_ReadStatus()	Reads the status of the TCPWM

Table 23: PWM Interrupt Sources

Parameter Name	Value	Description
Terminal count	true	Interrupt when the timer hits its terminal count value
Compare/Capture	true	Interrupt when the compare or capture inputs are asserted.

Table 24: PWM_Clock Parameters

Parameter Name	Value	Description
Frequency	1MHz	Clock frequency.
Source	HFCLK	Source clock, from which this clock is derived.
Divider	48	Local divider value to obtain desired frequency from source clock.
Accuracy	+/- 2%	Accuracy of the source clock.

Table 25: RED_LED and PWM_*n*_out Parameters

Parameter Name	Value	Description
Type	Digital Output	Output pin.
Drive Mode	Strong	Strong drive.
Initial State	1	Initial value written to the pin's data register after power-on reset (POR).

Parameter Name	Value	Description
Slew Rate	Fast	Rise and fall ramp rate for the pin as it changes output logic levels. Fast mode is required for signals that switch at greater than 1 MHz.
Drive level	Vddio	Output drive voltage supply sourced by the pin.
Current	4mA source 8mA sink	The amount of current that can be sourced/sunk at the pin.

Table 26: PWM_*n*_Int Parameters

Parameter Name	Value	Description
Interrupt Type	LEVEL	IRQ source is sticky and remains active until firmware clears the source of the request with an action.

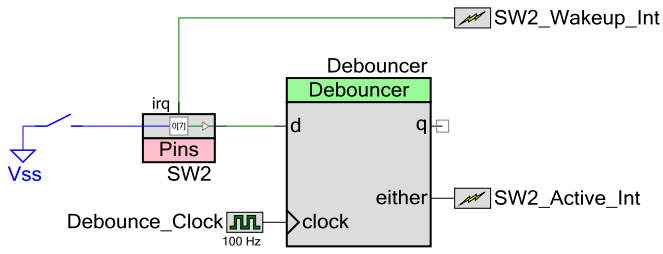
Debounced Switch

SW2 is a switch on the Pioneer kit. It is connected to P0[7] on the PSoC device. The input signal is connected to a debouncer component that filters out button “ringing”.

Note that, in the figure below the blue components are documentation. They show the (active low) connection of the switch on the Pioneer kit.

There are two interrupts associated with the switch. The SW2_Wakeup_Int is intended to be used to wake the device from sleep states. This interrupt can be used in active state but is not debounced and so can generate multiple IRQs for every button press. The SW2_Active_Int is debounced and is intended to be used when the device is in the active state. SW2_Active_Int cannot wake the part from a sleep state.

Figure 11: Debounced Switch Schematic



This schematic contains the following component instances (click links for details on features and APIs).

- Instance Debouncer (type: Debouncer_v1_0)
- Instance Debounce_Clock (type: cy_clock_v2_10)
  - See the chapter on Using the cy_clock Component for API information.
- Instance SW2 (type: cy_pins_v1_90)
  - See the chapter on Using the cy_pins Component for API information.

- Instance SW2_Wakeup and SW2_Active_Int (type: cy_isr_v1_70)
    - See the chapter on Using the cy_isr Component for API information.

Table 27: Debouncer Parameters

Parameter Name	Value	Description
Edge Type	Either	Assert an IRQ on rising, falling or both (either) edges

Table 28: Debounce_Clock Parameters

Parameter Name	Value	Description
Frequency	100Hz	Clock frequency.
Source	HFCLK	Source clock, from which this clock is derived.
Divider	480000[1]	Local divider value to obtain desired frequency from source clock.
Accuracy	+/- 2%	Accuracy of the source clock.

Table 29: SW2 Parameters

Parameter Name	Value	Description
Type	Digital Input	Input pin.
Drive Mode	Res pull up	Resistive pull-up. Reads high when input is not driven strong.
Initial State	1	Initial value written to the pin's data register after power-on reset (POR).
Threshold	CMOS	Threshold levels that define a logic high level (1) and a logic low level (0).
Interrupt	Both edges	Selects whether the pin can generate an interrupt and, if selected, the interrupt type.

Table 30: SW2_Active_Int Parameters

Parameter Name	Value	Description
Interrupt Type	LEVEL	IRQ source is sticky and remains active until firmware clears the source of the request with an action.

Table 31: SW2_Wakeup_Int Parameters

Parameter Name	Value	Description

[1] This exceeds the maximum possible divider value (16-bit) for a single clock. The division is obtained by chaining two clock dividers. The clock APIs that modify the divider value will, as a result, likely not have the intended result. Avoid using these APIs on this clock or use PSoC Creator to reconfigure the clock.

Parameter Name	Value	Description
Interrupt Type	RISING-EDGE	IRQ is triggered from an edge-detect circuit that generates a synchronous one-cycle pulse. Firmware is not required to clear the interrupt.

Timer

The Timer is configured to measure events with a 1ms granularity. The input clock frequency can be adjusted by software to change the timer resolution. The Timer_Clock is sourced from a design-wide clock – Clock_1MHz – to enable the local clock APIs to easily change the clock divider without impacting other clocks in the system.

The Timer is controlled by a combination of firmware-driven signals, via the Timer_Ctl_Reg, and/or an input pin, Timer_Pin. Start, reload and capture inputs are available. Timer_Pin is P0[6], which is on header J3_06. Timer_Pin is active high.

The register provides two functions. The upper three bits control the muxes that select the input sources to the Timer component. Each bit corresponds to one of the inputs; start, reload and capture. When the register bit is high the Timer_Pin is routed to the associated input. When low, inputs are driven by software, via the lower three bits.

The Timer is implemented in a hardware block called a TCPWM – Timer/Counter/PWM. A TCPWM can be configured as a Timer/Counter, PWM or Quadrature Decoder using PSoC Creator.

Figure 12: Timer Control Register Schematic

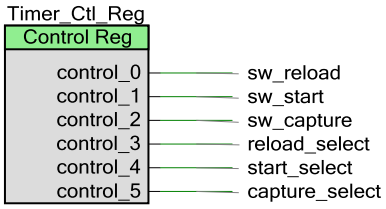
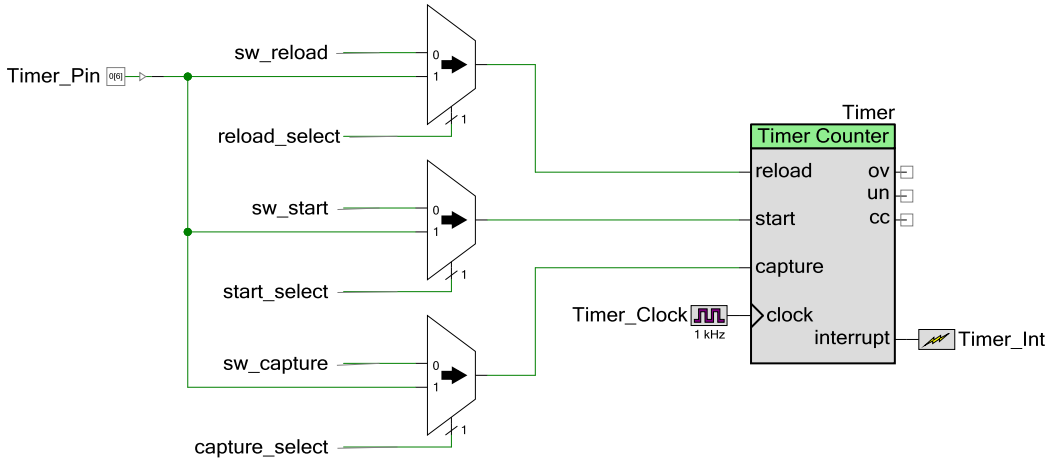


Figure 13: Timer Schematic



This schematic contains the following component instances (click links for details on features and APIs).

- Instance Timer (type: TCPWM_v1_0)
- Instance Timer_Clock (type: cy_clock_v2_10)
    - See the chapter on Using the cy_clock Component for API information.
- Instance Timer_Pin (type: cy_pins_v1_90)
    - See the chapter on Using the cy_pins Component for API information.
- Instance Timer_Int (type: cy_isr_v1_70)
    - See the chapter on Using the cy_isr Component for API information.

Table 32: Timer_Clock Parameters

Parameter Name	Value	Description
Frequency	1kHz	Clock frequency.
Source	HFCLK	Source clock, from which this clock is derived.
Divider	48000	Local divider value to obtain desired frequency from source clock.
Accuracy	+/- 2%	Accuracy of the source clock.

Table 33: Timer_Pin Parameters

Parameter Name	Value	Description
Type	Digital Input	Input pin.
Drive Mode	Res pull dn	Resistive pull-down. Reads low when input is not driven strong.
Initial State	0	Initial value written to the pin's data register after power-on reset (POR).
Threshold	CMOS	Threshold levels that define a logic high level (1) and a logic low level (0).
Interrupt	None	Selects whether the pin can generate an interrupt and, if selected, the interrupt type.

Table 34: Timer_Int Parameters

Parameter Name	Value	Description
Interrupt Type	LEVEL	IRQ source is sticky and remains active until firmware clears the source of the request with an action.

Table 35: Timer APIs

Function	Description
Timer_Init()	Initialize/Restore default TCPWM configuration
Timer_Enable()	Enables the TCPWM. TCPWM will be started if the Start terminal is not present

Function	Description
Timer_Start()	Initializes the TCPWM with default customizer values when called the first time and enables the TCPWM. TCPWM will be started if the Start terminal is not present
Timer_Stop()	Disables the TCPWM
Timer_SetMode()	Sets the operational mode of the TCPWM
Timer_SetPrescaler()	Sets the prescaler value that is applied to the clock input
Timer_TriggerCommand()	Triggers the designated command to occur on the designated TCPWM instances
Timer_SetOneShot()	Writes the register that controls whether the TCPWM runs continuously or stops after terminal count is reached
Timer_SetInterruptMode()	Sets the interrupt mask to control which interrupt requests generate the interrupt signal
Timer_GetInterruptSourceMasked()	Gets the interrupt requests masked by the interrupt mask
Timer_GetInterruptSource()	Gets the interrupt requests (without masking)
Timer_ClearInterrupt()	Clears the interrupt request
Timer_SetInterrupt()	Sets a software interrupt request
Timer_WriteCounter()	Writes a new 16 bit counter value directly into the counter register
Timer_ReadCounter()	Reads the current counter value
Timer_SetCounterMode()	Sets the counter mode
Timer_ReadCapture()	Reads the captured counter value
Timer_ReadCaptureBuf()	Reads the capture buffer register
Timer_WritePeriod()	Writes the 16 bit period register with the new period value
Timer_ReadPeriod()	Reads the 16 bit period register
Timer_SetCaptureMode()	Sets the capture trigger mode
Timer_SetReloadMode()	Sets the reload trigger mode
Timer_SetStartMode()	Sets the start trigger mode
Timer_SetStopMode()	Sets the stop trigger mode
Timer_SetCountMode()	Sets the count trigger mode
Timer_ReadStatus()	Reads the status of the TCPWM

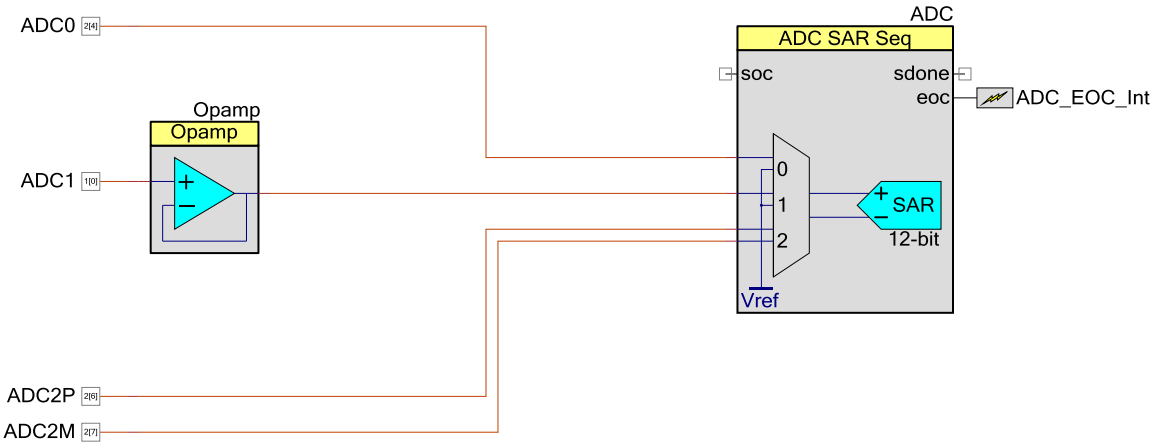
ADC

The ADC is configured with three channels, connected to pins ADC0, ADC1 and ADC3P (plus) and ADC2M (minus). It can generate an IRQ after every conversion.

- Channel 0 is a single-ended input to pin ADC0 (P2[4]) on header J2_09

- Channel 1 is a single-ended input to pin ADC1 (P1[0]) on headers J2_17 and J4_07 via a signal-boosting Opamp
- Channel 2 is a differential input to pins ADC2P and ADC2M
    - The positive input is ADC2P (P2[6]) on headers J2_13 and J4_05
    - The negative input is ADC2M (P2[7]) on header J2_15

Figure 14: ADC Schematic



This schematic contains the following component instances (click links for details on features and APIs).

- Instance ADC (type: ADC_SAR_SEQ_P4_v1_10)
- Instance Opamp (type: OpAmp_P4_v1_0)
- Instances ADC0, ADC1, ADC2P and ADC2M (type: cy_pins_v1_90)
    - See the chapter on Using the cy_pins Component for API information.
- Instance ADC_EOC_Int (type: cy_isr_v1_70)
    - See the chapter on Using the cy_isr Component for API information.

Table 36: ADC Parameters

Parameter Name	Value	Description
Sample Rate	5050sps	Number of samples taken per second.
Sampling Mode	Free Running	Controls whether each scan must be triggered by the SOC terminal or the ADC runs continuously.
Vref Source	Vdda / 2	Internal reference voltage.
Vref Value	1.65V	Actual voltage of Vref source.
Single-Ended Range	0V to 3.3V	Measurable range for single-ended channels.
Single-Ended Result Format	signed	Format of returned value for all single-ended channels.
Differential Range	-1.65V to 1.65V	Measurable range for differential channels.

Parameter Name	Value	Description
Differential Result Format	signed	Format of returned value for all differential channels.
Samples Averaged	8	Number of samples averaged for each conversion.
Interrupt High Limit	0x7FF	Upper limit for Compare Mode.
Interrupt High Limit	0x0	Lower limit for Compare Mode.
Interrupt Compare Mode	(Result < Low Limit) or (High Limit <= Result)	Limit condition that will trigger a maskable range detect interrupt.

Table 37: ADC APIs

Function	Description
ADC_Start()	Performs all required initialization for this component and enables the power. The power will be set to the appropriate power based on the clock frequency.
ADC_Stop()	This function stops ADC conversions and puts the ADC into its lowest power mode.
ADC_StartConvert()	For free running mode, this API starts the conversion process and it runs continuously. In a triggered mode, this routine triggers every conversion.
ADC_StopConvert()	Forces the ADC to stop conversions. If a conversion is currently executing, that conversion will complete, but no further conversions will occur.
ADC_IRQ_Enable()	Enables interrupts to occur at the end of a conversion. Global interrupts must also be enabled for the ADC interrupts to occur.
ADC_IRQ_Disable()	Disables interrupts at the end of a conversion.
ADC_IsEndConversion()	Immediately returns the status of the conversion or does not return (blocking) until the conversion completes, depending on the retMode parameter.
ADC_GetResult16()	Gets the data available in the SAR result register.
ADC_SetChanMask()	Sets the channel enable mask. Sets which channels that will be scanned.
ADC_EnableInjection()	Enables the injection channel for the next scan only.
ADC_SetLowLimit()	This parameter sets the low limit for a limit compare.
ADC_SetHighLimit()	This parameter sets the high limit for a limit compare.
ADC_SetLimitMask()	Sets which channels may cause a limit condition interrupt.
ADC_SetSatMask()	Sets which channels may cause a saturation event interrupt.
ADC_SetOffset()	Sets the offset of the ADC channel.

Function	Description
ADC_SetGain()	Sets the gain in counts per 10 volt for the ADC channel.
ADC_CountsTo_Volts()	Converts the ADC output to volts as a floating point number.
ADC_CountsTo_mVolts()	Converts the ADC output to millivolts.
ADC_CountsTo_uVolts()	Converts the ADC output to microvolts.
ADC_Sleep()	Stops the ADC operation and saves the configuration registers and component enable state.
ADC_Wakeup()	Restores the component enable state and configuration registers.
ADC_SaveConfig()	Save the current configuration of ADC non-retention registers.
ADC_RestoreConfig()	Restores the configuration of ADC non-retention registers.

Table 38: Opamp Parameters

Parameter Name	Value	Description
Mode	Follower	Selects the 2-input (Opamp) or 1-input (Follower) configuration. In follower mode, the inverting input is internally connected to the output to create a voltage follower.
Power	High	Selects the operating power level: High Power, Medium Power, or Low Power. Higher operating current increases the Opamp bandwidth.
Output Current	1mA	Selects output mode: 1mA or 10mA.
Compensation	High Stability	Compensation is used to prevent unwanted oscillations in the output

Table 39: Opamp APIs

Function	Description
Opamp_Init()	Initializes or restores the component according to the customizer Configure dialog settings.
Opamp_Enable()	Activates the hardware and begins component operation.
Opamp_Start()	Performs all of the required initialization for the component and enables power to the block.
Opamp_Stop()	Turns off the Opamp block.
Opamp_SetPower()	Sets the drive power to one of three settings; LOWPOWER, MEDPOWER, HIGHPOWER.
Opamp_PumpControl()	Turn the boost pump on or off.
Opamp_Sleep()	This is the preferred API to prepare the component for

Function	Description
	sleep.
Opamp_Wakeup()	This is the preferred API to restore the component to the state when Opamp_Sleep() was called.
Opamp_SaveConfig()	Saves the configuration of the component.
Opamp_RestoreConfig()	Restores the configuration of the component.

Table 40: ADC0, ADC1, ADC2P and ADC2M Parameters

Parameter Name	Value	Description
Type	Analog	Analog pin.
Drive Mode	HiZ Analog	High Impedance - Analog.

Table 41: ADC_EOC_Int Parameters

Parameter Name	Value	Description
Interrupt Type	RISING-EDGE	IRQ source is sticky and remains active until firmware clears the source of the request with an action.

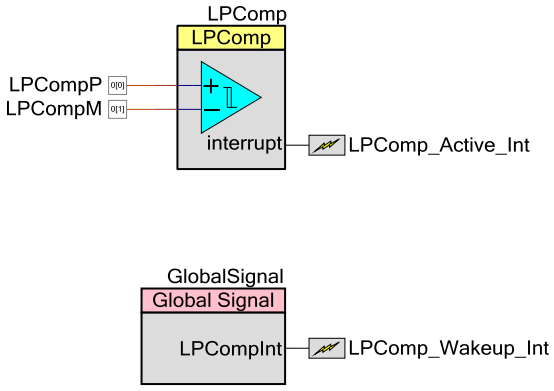
Comparator

The Comparator simply compares the input voltages and stores the result in a register. It is configured to generate an interrupt – LPComp_Active_Int – when there is a change in the compare value.

- The positive input voltage is LPCompP (P0[0]) on headers J2_13 and J4_05
- The negative input voltage is LPCompM (P0[1]) on header J2_15

The GlobalSignal component remains active when the device is in a sleep mode. It can assert the interrupt – LPComp_Wakeup_Int – from both the Sleep and DeepSleep states.

Figure 15: Comparator Schematic



This schematic contains the following component instances (click links for details on features and APIs).

- Instance LPComp (type: LPComp_P4_v1_0)

- Instance GlobalSignal (type: cy_gsref_v2_0)
- Instances LPCompM and LPCompP (type: cy_pins_v1_90)
    - See the chapter on Using the cy_pins Component for API information.
- Instances LPComp_Active_Int and LPComp_Wakeup_Int (type: cy_isr_v1_70)
    - See the chapter on Using the cy_isr Component for API information.

Table 42: LPComp Parameters

Parameter Name	Value	Description
DigitalFilter	Disable	Determines if the output is synced to the system clock
Hysteresis	Enable	Enables the output hysteresis
Interrupt	Both edges	Selects the detection setting for the output and the interrupt
Speed	Slow/Ultra low power	Component power setting. This setting is required for sleep mode wakeup.

Table 43: LPComp APIs

Function	Description
LPComp_Start()	Performs all of the required initialization for the component and enables power to the block.
LPComp_Init()	Initializes or restores the component according to the customizer settings.
LPComp_Enable()	Activates the hardware and begins component operation.
LPComp_Stop()	Turns off the LP Comparator block.
LPComp_GetCompare()	This function returns a nonzero value when the voltage connected to the positive input is greater than the negative input voltage.
LPComp_SetSpeed()	Sets the power and speed to one of three settings.
LPComp_SetInterruptMode()	Sets the interrupt edge detect mode.
LPComp_GetInterruptSource()	Gets the interrupt requests.
LPComp_ClearInterrupt()	Clears the interrupt request.
LPComp_SetInterrupt()	Sets a software interrupt request.
LPComp_SetHysteresis()	Enables or disables the hysteresis setting.
LPComp_SetFilter()	Enables or disables the digital filter setting.
LPComp_ZeroCal()	Performs custom calibration of the input offset to minimize error for a specific set of conditions.
LPComp_LoadTrim()	Writes a value into the comparator offset trim register.

Table 44: GlobalSignal Parameters

Parameter Name	Value	Description
Signal	LPCompInt	Selected global signal. LPCompInt triggers when any enabled low power comparator generates an IRQ.

Table 45: LPComp_Active_Int Parameters

Parameter Name	Value	Description
Interrupt Type	RISING-EDGE	IRQ source is sticky and remains active until firmware clears the source of the request with an action.

Table 46: LPComp_Wakeup_Int Parameters

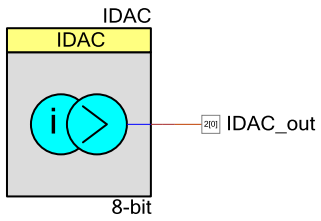
Parameter Name	Value	Description
Interrupt Type	LEVEL	IRQ source is sticky and remains active until firmware clears the source of the request with an action.

IDAC

This component is only available in the IDAC-enabled design. If CapSense is present in the system, the IDAC is disabled.

The IDAC can source current to the IDAC_out pin. The maximum current is 612µA and the resolution is 8-bit (2.4µA/bit).

Figure 16: IDAC Schematic



This schematic contains the following component instances (click links for details on features and APIs).

- Instance IDAC (type: IDAC_P4_v1_0)
- Instance IDAC_out (type: cy_pins_v1_90)
    - See the chapter on Using the cy_pins Component for API information.

Table 47: IDAC Parameters

Parameter Name	Value	Description
Type	Source	Mode of operation. Negative/Sink (default) or Positive/Source.

Parameter Name	Value	Description
Range	High	Output range of the IDAC. Can be either 0-306uA (Low) or 0-612uA (High).
Resolution	8-bit	Number of bits in the IDAC. Current granularity or 1.2uA/bit (low range) or 2.4uA/bit (high range).
Value	0	Initial output value of the IDAC.

Table 48: IDAC APIs

Function	Description
IDAC_Start()	Performs all of the required initialization for the component and enables power to the block.
IDAC_Stop()	Turn off the IDAC block.
IDAC_Init()	Initializes or restores the component according to the customizer Configure dialog settings.
IDAC_Enable()	Activates the hardware and begins component operation.
IDAC_SetValue(uint32 value)	Sets the DAC's output value.
IDAC_Sleep()	This is the preferred API to prepare the component for sleep.
IDAC_Wakeup()	This is the preferred API to restore the component to the state when IDAC_Sleep() was called.
IDAC_SaveConfig()	Saves the configuration of the component.
IDAC_RestoreConfig()	Restores the configuration of the component.

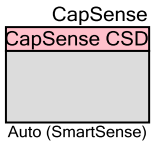
Table 49: IDAC_out Parameters

Parameter Name	Value	Description
Type	Analog	Analog pin.
Drive Mode	HiZ Analog	High Impedance - Analog.

CapSense

This component is only available in the CapSense-enabled design. If the IDAC is present in the system, CapSense is disabled.

CapSense is configured to support the 5-element slider on the kit.



Capsense Configuration
5-sensor slider
Cmod (modulation capacitor)

CapSense Connections
Sensor 0 - P1.1
Sensor 1 - P1.2
Sensor 2 - P1.3
Sensor 3 - P1.4
Sensor 4 - P1.5
Cmod - P4.2

This schematic contains the following component instances (click links for details on features and APIs).

- Instance CapSense (type: CapSense_CSD_P4_v1_11)

Table 50: CapSense Parameters

Parameter Name	Value	Description
SensorNumber	5	Total sensors count

Table 51: CapSense APIs

Function	Description
CapSense_Start()	Preferred method to start the component. Initializes registers and enables active mode power template bits of the subcomponents used within CapSense.
CapSense_Stop()	Disables component interrupts, and calls CapSense_ClearSensors() to reset all sensors to an inactive state.
CapSense_Sleep()	Prepares the component for the device entering a low-power mode. Disables Active mode power template bits of the sub components used within CapSense, saves nonretention registers, and resets all sensors to an inactive state.
CapSense_Wakeup()	Restores CapSense configuration and nonretention register values after the device wake from a low power mode sleep mode.
CapSense_Init()	Initializes the default CapSense configuration provided with the customizer.
CapSense_Enable()	Enables the Active mode power template bits of the subcomponents used within CapSense.
CapSense_SaveConfig()	Saves the configuration of CapSense.
CapSense_RestoreConfig()	Restores CapSense configuration.
CapSense_ScanSensor()	Sets scan settings and starts scanning a sensor or group of combined sensors.
CapSense_ScanEnabledWidgets()	The preferred scanning method. Scans all of the enabled widgets.
CapSense_IsBusy()	Returns the status of sensor scanning.
CapSense_SetScanSlotSettings()	Sets the scan settings of the selected scan slot (sensor).
CapSense_ClearSensors()	Resets all sensors to the nonsampling state.
CapSense_EnableSensor()	Configures the selected sensor to be scanned during the next scanning cycle.
CapSense_DisableSensor()	Disables the selected sensor so it is not scanned in the next scanning cycle.

CapSense_ReadSensorRaw()	Returns sensor raw data from the CapSense_SensorResult[ ] array.
CapSense_InitializeSensorBaseline()	Loads the CapSense_sensorBaseline[sensor] array element with an initial value by scanning the selected sensor.
CapSense_InitializeEnabledBaselines()	Loads the CapSense_sensorBaseline[] array with initial values by scanning enabled sensors only. This function is available only for two-channel designs.
CapSense_InitializeAllBaselines()	Loads the CapSense_sensorBaseline[] array with initial values by scanning all sensors.
CapSense_UpdateSensorBaseline()	The historical count value, calculated independently for each sensor, is called the sensor's baseline. This baseline updated uses a low-pass filter with k = 256.
CapSense_UpdateEnabledBaselines()	Checks the CapSense_sensorEnableMask []array and calls the CapSense_UpdateSensorBaseline() function to update the baselines for enabled sensors.
CapSense_EnableWidget()	Enables all sensor elements in a widget for the scanning process.
CapSense_DisableWidget()	Disables all sensor elements in a widget from the scanning process.
CapSense_CheckIsWidgetActive()	Compares the selected of widget to the CapSense_Signal[] array to determine if it has a finger press.
CapSense_CheckIsAnyWidgetActive()	Uses the CapSense_CheckIsWidgetActive() function to find if any widget of the CapSense CSD component is in active state.
CapSense_GetCentroidPos()	Checks the CapSense_sensorSignal[] array for a finger press in a linear slider and returns the position.
CapSense_GetRadialCentroidPos()	Checks the CapSense_sensorSignal[] array for a finger press in a radial slider widget and returns the position.
CapSense_GetTouchCentroidPos()	If a finger is present, this function calculates the X and Y position of the finger by calculating the centroids within the touchpad.
CapSense_GetMatrixButtonPos()	If a finger is present, this function calculates the row and column position of the finger on the matrix buttons.

Software Pins

This schematic shows the pinouts for the three headers that are accessible from the PSoC 4 device; J2, J3, and J4.

The pins for all the components discussed above are labeled here for convenience. Note that some pins are routed to multiple headers.

The cy_pins component instances, named “JX_YY”, are software pins. By default they are configured as digital outputs (Strong drive) but can be re-configured as inputs from software.

Figure 17: Software Pin Schematic - CapSense-enabled

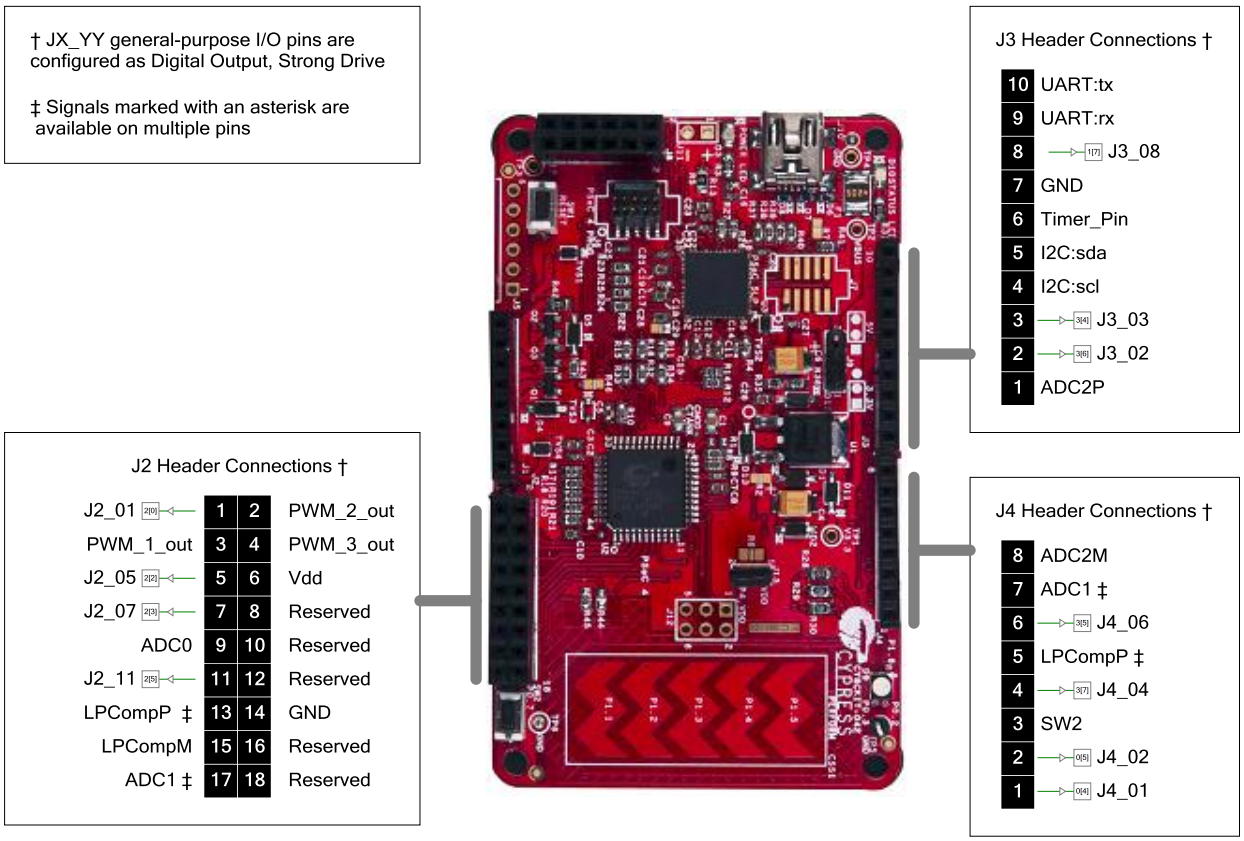
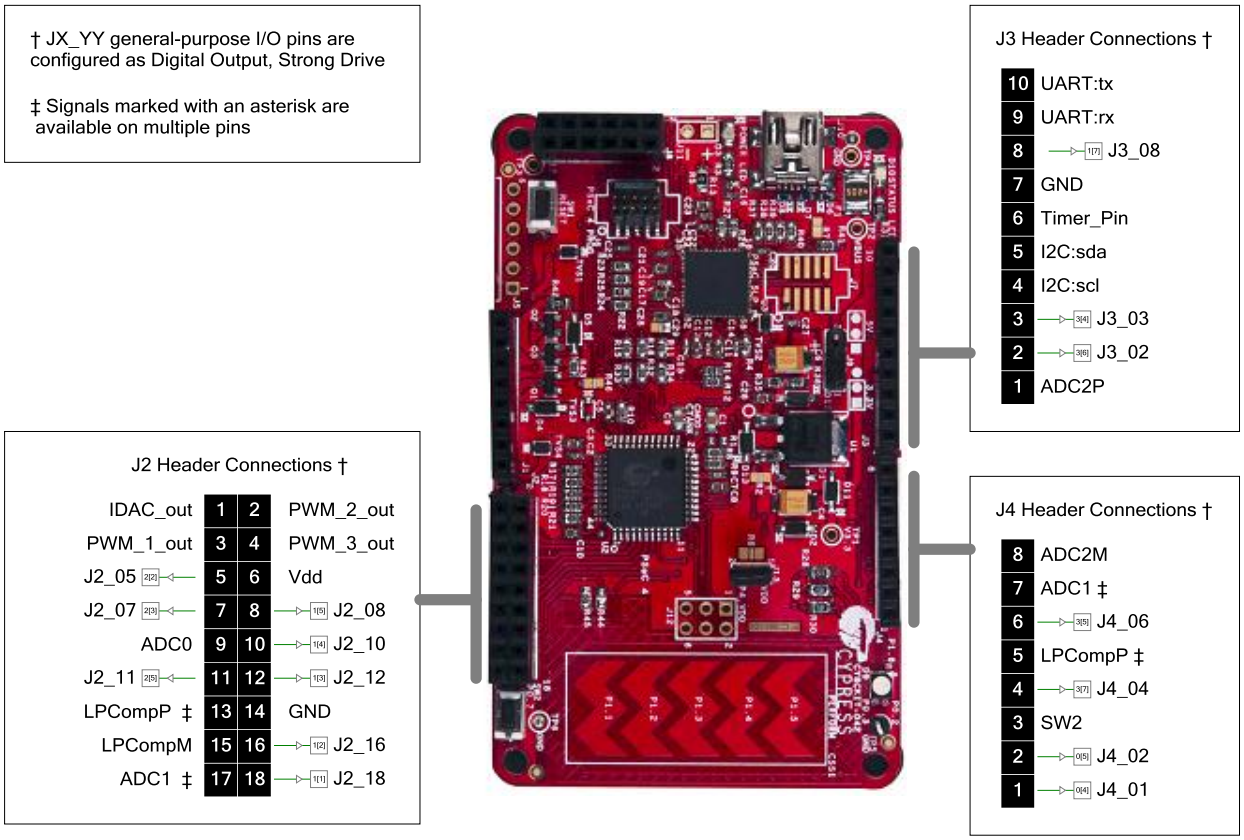


Figure 18: Software Pin Schematic – IDAC-enabled



This schematic contains the following component instances (click links for details on features and APIs).

- Instance J2_**, J3_** and J4_** (type: cy_pins_v1_90)
    - See the chapter on Using the cy_pins Component for API information.

Table 52: J2_**, J3_** and J4_** Parameters

Parameter Name	Value	Description
Type	Digital Output	Output pin.
Drive Mode	Strong	Strong drive.

Other Resources

The following documents contain important information on Cypress software APIs that might be relevant to this design:

- Standard Types and Defines chapter in the System Reference Guide
    - Software base types
    - Hardware register types
    - Compiler defines
    - Cypress API return codes
    - Interrupt types and macros
- Registers
    - The full PSoC 4 register map is covered in the PSoC 4 Registers Technical Reference Manual
    - Register Access chapter in the System Reference Guide
    - CY_GET API routines
    - CY_SET API routines
- System Functions chapter in the System Reference Guide
    - General API routines
    - CyDelay API routines
    - CyVd Voltage Detect API routines
- Power Management
    - Power Supply and Monitoring chapter in the PSoC 4 Technical Reference Manual
    - Low Power Modes chapter in the PSoC 4 Technical Reference Manual
    - Power Management chapter in the System Reference Guide
        - CyPm API routines
- Watchdog Timer chapter in the System Reference Guide
    - CyWdt API routines

Revision History

Version	Changes
1.0	New document