PSoC® Board Support Package for CY8CKIT-050 (PSoC 5LP Developers Kit)

v1.1

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

Table of Contents 3
Chapters 3
Tables 4
Figures 4
Device Family Overview 6
Resources 8
System Settings 9
Configuration 9
Debug 9
Operating Conditions 9
Pins 11
Device Pin Functions 12
Accessing Unused Pins 15
Using the cy_pins Component 15
Clocks 17
System Clocks 18
Local and Design Wide Clocks 18
Using the cy_clock Component 19
Interrupts 21
Global Interrupt Control 21
Using the cy_isr Component 22
DMA 23
Design Contents 24
Switches 24
LEDs 25
Communications 29
Timer 33
Analog-Digital Converters (ADC) 35
Digital-Analog Converter (DAC) 39
Comparators 41
CapSense 43
LCD 46
Other Resources 48
Revision History 49

Tables

Table 1: Device Characteristics 7
Table 2: Device Resources 8
Table 3: Configuration Settings 9
Table 4: Debug Settings 9
Table 5: Operating Conditions 9
Table 6: Device Pins 12
Table 7: Per-Pin APIs (for [unused] pins) 15
Table 8: cy_pins APIs 15
Table 9: System Clocks 18
Table 10: Local Clocks 18
Table 11: cy_clock APIs 19
Table 12: Interrupts 21
Table 13: cy_isr APIs 22
Table 14: SW*n*_Debouncer Parameters 24
Table 15: LED*n*_Control Parameters 25
Table 16: LED*n*_Control APIs 26
Table 17: PWM_*n* Parameters 26
Table 18: I2C Parameters 29
Table 19: I2C APIs 30
Table 20: I2C Sleep/Wake APIs 30
Table 21: UART Parameters 30
Table 22: UART APIs 31
Table 23: Timer_1MHz Parameters 33
Table 24: Timer_1MHz APIs 33
Table 25: ADC_SAR_*n* Parameters 36
Table 26: ADC_SAR_*n* APIs 36
Table 27: ADC_DelSig Parameters 37
Table 28: ADC_DelSig APIs 37
Table 29: DAC Parameters 39
Table 30: DAC APIs 39
Table 31: Comp_*n* Parameters 41
Table 32: Comp_*n* APIs 42
Table 33: Comp_*n*_VRef Parameters 42
Table 34: Comp_*n*_VRef APIs 42
Table 35: General APIs 43
Table 36:Scanning APIs 44
Table 37: High-Level APIs 44
Table 38: LCD APIs 46

Figures

Figure 1: CY8C58LP Device Family Block Diagram 6
Figure 2: Device Pin Layout 11
Figure 3: System Clock Configuration 17
Figure 4: Local and Design Wide Clock Configuration 18
Figure 5: Switches 24
Figure 6: LEDs 25
Figure 7: PWM_*n* APIs 26

Figure 8: I2C 29
Figure 9: UART 29
Figure 10: MHz Timer.....33
Figure 11: SAR ADC 1 35
Figure 12: SAR ADC 2 35
Figure 13: Delta-Sigma ADC 35
Figure 14: Voltage DAC.....39
Figure 15: Comparators.....41
Figure 16: CapSense.....43
Figure 17: LCD.....46

Device Family Overview

The Cypress PSoC 5LP is a family of 32-bit devices with the following characteristics.

- High-performance 32-bit ARM Cortex-M3 core with a nested vectored interrupt controller (NVIC) and a high-performance DMA controller
- Digital system that includes configurable Universal Digital Blocks (UDBs) and specific function peripherals, such as USB, I2C and SPI
- Analog subsystem that includes 20-bit Delta Sigma converters (ADC), SAR ADCs, 8-bit DACs that can be configured for 12-bit operation, comparators, op amps and configurable switched capacitor (SC) and continuous time (CT) blocks to create PGAs, TIAs, mixers, and more
- Several types of memory elements, including SRAM, flash, and EEPROM
- Programming and debug system through Serial Wire Debug (SWD), and Single Wire Viewer (SWV)
- Flexible routing to all pins

Figure 1 shows the major components of a typical CY8C58LP device. For details on all the systems listed above, please refer to the PSoC 5LP Technical Reference Manual.

Figure 1: CY8C58LP Device Family Block Diagram

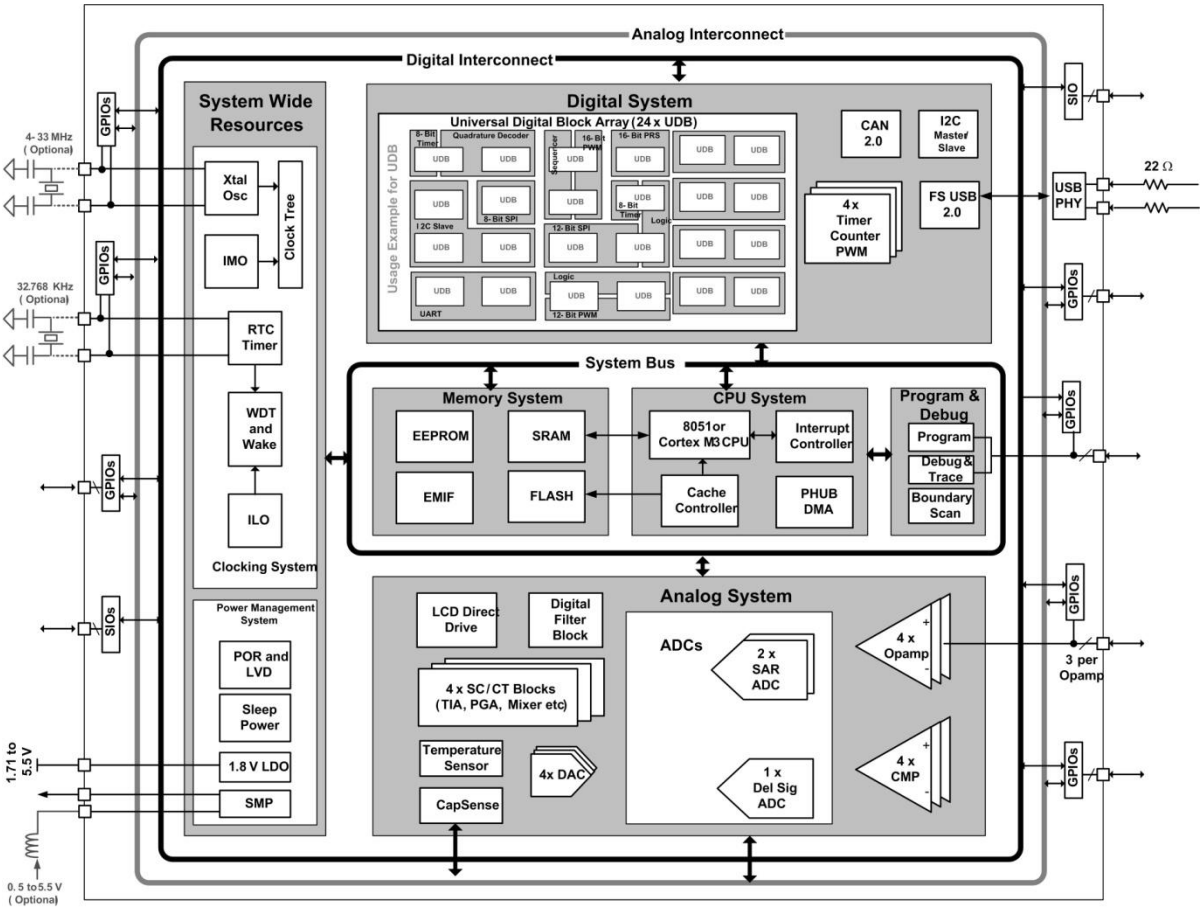


Table 1 lists the key characteristics of this device.

Table 1: Device Characteristics

Name	Value
Device	CY8C5868AXI-LP035
Architecture	PSoC 5LP
Family	CY8C58LP
CPU speed (MHz)	67
Flash size (kBytes)	256
SRAM size (kBytes)	64
EEPROM size (Bytes)	2048
Trace Buffer (kBytes)	0
Vdd range (V)	1.71 to 5.5
Automotive qualified	No (Industrial Grade Only)
Temp range (Celsius)	-40 to 85
JTAG ID	0x2E123069

NOTE: The CPU speed noted above is the maximum available speed. The CPU is clocked by BUS_CLK, listed in the System Clocks section below.

Resources

Table 2 lists the device resources that this design uses.

Table 2: Device Resources

Resource Name	In Use	Total Available
Digital domain clock dividers	7 (87.5%)	8
Analog domain clock dividers	3 (75.0%)	4
Pins	34 (47.2%)	72
UDB Macrocells	52 (27.1%)	192
UDB Unique Pterms	89 (23.2%)	384
UDB Datapath Cells	10 (41.7%)	24
UDB Status Cells	5 (20.8%)	24
UDB Control Cells	6 (25.0%)	24
DMA Channels	0 (0.0%)	24
Interrupts	19 (59.4%)	32
DSM Fixed Blocks	1 (100.0%)	1
VIDAC Fixed Blocks	4 (100.0%)	4
SC Fixed Blocks	0 (0.0%)	4
Comparator Fixed Blocks	4 (100.0%)	4
Opamp Fixed Blocks	0 (0.0%)	4
CapSense Buffers	1 (50.0%)	2
CAN Fixed Blocks	0 (0.0%)	1
Decimator Fixed Blocks	1 (100.0%)	1
I2C Fixed Blocks	1 (100.0%)	1
Timer Fixed Blocks	2 (50.0%)	4
DFB Fixed Blocks	0 (0.0%)	1
USB Fixed Blocks	0 (0.0%)	1
LCD Fixed Blocks	0 (0.0%)	1
EMIF Fixed Blocks	0 (0.0%)	1
LPF Fixed Blocks	0 (0.0%)	2
SAR Fixed Blocks	2 (100.0%)	2

System Settings

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

Configuration

Table 3: Configuration Settings

Name	Value
Device Configuration Mode	Compressed
Enable Error Correcting Code (ECC)	False
Store Configuration Data in ECC Memory	True
Instruction Cache Enabled	True
Enable Fast IMO During Startup	True
Unused Bonded IO	Allow but warn
Heap Size (bytes)	0x1000
Stack Size (bytes)	0x4000
Include CMSIS Core Peripheral Library	True

Debug

Table 4: Debug Settings

Name	Value
Debug Select	SWD+SWV
Enable Device Protection	False
Embedded Trace (ETM)	False
Use Optional XRES	False

Operating Conditions

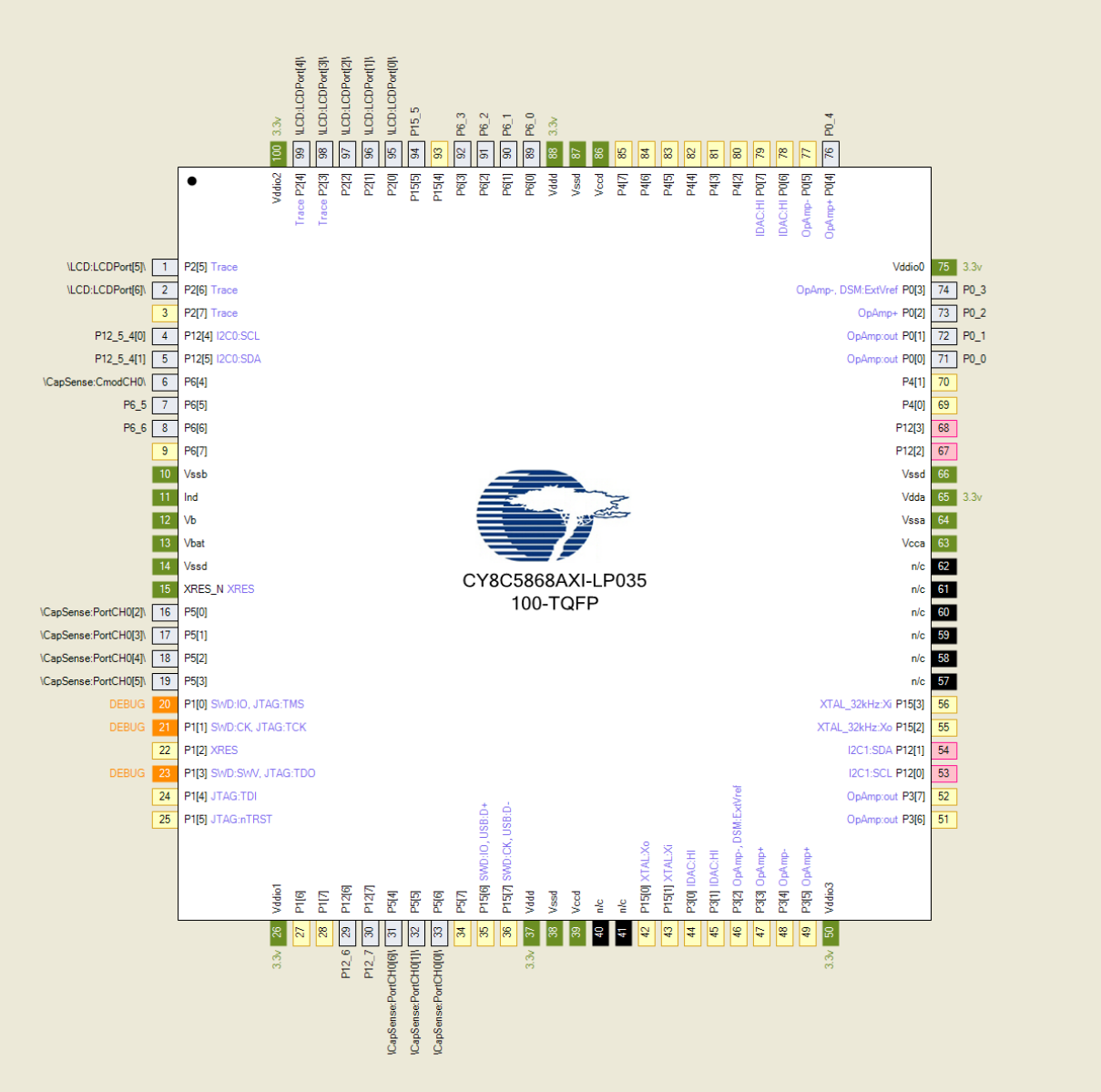
Table 5: Operating Conditions

Name	Value
Vddd (V)	3.3
Vdda (V)	3.3
Vddio0 (V)	3.3
Vddio1 (V)	3.3
Vddio2 (V)	3.3
Vddio3 (V)	3.3
Variable Vdda	False

Pins

Figure 2 shows the pin layout of this device.

Figure 2: Device Pin Layout



Device Pin Functions

Table 6 contains information about the pins on this device in device pin order. (No-connection ["n/c"] pins have been omitted.)

Table 6: Device Pins

Pin	Port	Name	Type	Drive Mode	Reset State
1	P2[5]	LCD:LCDPort[5]		Strong	HiZ Analog Unb
2	P2[6]	LCD:LCDPort[6]		Strong	HiZ Analog Unb
3	P2[7]	GPIO [unused]			HiZ Analog Unb
4	P12[4]	P12_5_4[0]	Digital I/O	OD, DL	HiZ Analog Unb
5	P12[5]	P12_5_4[1]	Digital I/O	OD, DL	HiZ Analog Unb
6	P6[4]	CapSense:CmodCH0	Analog	HiZ Analog	HiZ Analog Unb
7	P6[5]	P6_5	Analog	HiZ Analog	HiZ Analog Unb
8	P6[6]	P6_6	Digital In	HiZ Analog	HiZ Analog Unb
9	P6[7]	GPIO [unused]			HiZ Analog Unb
10	Vssb	Vssb	Power		
11	Ind	Ind	Power		
12	Vb	Vb	Power		
13	Vbat	Vbat	Power		
14	Vssd	Vssd	Power		
15	XRES_N	XRES_N	Power		
16	P5[0]	CapSense:PortCH0[2]	Analog	OD, DL	HiZ Analog Unb
17	P5[1]	CapSense:PortCH0[3]	Analog	OD, DL	HiZ Analog Unb
18	P5[2]	CapSense:PortCH0[4]	Analog	OD, DL	HiZ Analog Unb
19	P5[3]	CapSense:PortCH0[5]	Analog	OD, DL	HiZ Analog Unb
20	P1[0]	GPIO [unused]			HiZ Analog Unb
21	P1[1]	GPIO [unused]			HiZ Analog Unb
22	P1[2]	GPIO [unused]			HiZ Analog Unb
23	P1[3]	GPIO [unused]			HiZ Analog Unb
24	P1[4]	GPIO [unused]			HiZ Analog Unb
25	P1[5]	GPIO [unused]			HiZ Analog Unb
26	Vio1	Vio1	Power		
27	P1[6]	GPIO [unused]			HiZ Analog Unb
28	P1[7]	GPIO [unused]			HiZ Analog Unb
29	P12[6]	P12_6	Digital Out	Strong	HiZ Analog Unb
30	P12[7]	P12_7	Digital Out	Strong	HiZ Analog Unb
31	P5[4]	CapSense:PortCH0[6]	Analog	OD, DL	HiZ Analog Unb
32	P5[5]	CapSense:PortCH0[1]	Analog	OD, DL	HiZ Analog Unb
33	P5[6]	CapSense:PortCH0[0]	Analog	OD, DL	HiZ Analog Unb

Pin	Port	Name	Type	Drive Mode	Reset State
34	P5[7]	GPIO [unused]			HiZ Analog Unb
35	P15[6]	USB [unused]			HiZ Analog Unb
36	P15[7]	USB [unused]			HiZ Analog Unb
37	Vddd	Vddd	Power		
38	Vssd	Vssd	Power		
39	Vccd	Vccd	Power		
42	P15[0]	GPIO [unused]			HiZ Analog Unb
43	P15[1]	GPIO [unused]			HiZ Analog Unb
44	P3[0]	GPIO [unused]			HiZ Analog Unb
45	P3[1]	GPIO [unused]			HiZ Analog Unb
46	P3[2]	GPIO [unused]			HiZ Analog Unb
47	P3[3]	GPIO [unused]			HiZ Analog Unb
48	P3[4]	GPIO [unused]			HiZ Analog Unb
49	P3[5]	GPIO [unused]			HiZ Analog Unb
50	Vio3	Vio3	Power		
51	P3[6]	GPIO [unused]			HiZ Analog Unb
52	P3[7]	GPIO [unused]			HiZ Analog Unb
53	P12[0]	SIO [unused]			HiZ Analog Unb
54	P12[1]	SIO [unused]			HiZ Analog Unb
55	P15[2]	GPIO [unused]			HiZ Analog Unb
56	P15[3]	GPIO [unused]			HiZ Analog Unb
63	Vcca	Vcca	Power		
64	Vssa	Vssa	Power		
65	Vdda	Vdda	Power		
66	Vssd	Vssd	Power		
67	P12[2]	SIO [unused]			HiZ Analog Unb
68	P12[3]	SIO [unused]			HiZ Analog Unb
69	P4[0]	GPIO [unused]			HiZ Analog Unb
70	P4[1]	GPIO [unused]			HiZ Analog Unb
71	P0[0]	P0_0	Analog	HiZ Analog	HiZ Analog Unb
72	P0[1]	P0_1	Analog	HiZ Analog	HiZ Analog Unb
73	P0[2]	P0_2	Analog	HiZ Analog	HiZ Analog Unb
74	P0[3]	P0_3	Analog	HiZ Analog	HiZ Analog Unb
75	Vio0	Vio0	Power		
76	P0[4]	P0_4	Analog	HiZ Analog	HiZ Analog Unb
77	P0[5]	GPIO [unused]			HiZ Analog Unb
78	P0[6]	GPIO [unused]			HiZ Analog Unb
79	P0[7]	GPIO [unused]			HiZ Analog Unb

Pin	Port	Name	Type	Drive Mode	Reset State
80	P4[2]	GPIO [unused]			HiZ Analog Unb
81	P4[3]	GPIO [unused]			HiZ Analog Unb
82	P4[4]	GPIO [unused]			HiZ Analog Unb
83	P4[5]	GPIO [unused]			HiZ Analog Unb
84	P4[6]	GPIO [unused]			HiZ Analog Unb
85	P4[7]	GPIO [unused]			HiZ Analog Unb
86	Vccd	Vccd	Power		
87	Vssd	Vssd	Power		
88	Vddd	Vddd	Power		
89	P6[0]	P6_0	Digital Out	Strong	HiZ Analog Unb
90	P6[1]	P6_1	Digital In	Res pull up	HiZ Analog Unb
91	P6[2]	P6_2	Digital Out	Strong	HiZ Analog Unb
92	P6[3]	P6_3	Digital Out	Strong	HiZ Analog Unb
93	P15[4]	GPIO [unused]			HiZ Analog Unb
94	P15[5]	P15_5	Digital In	Res pull up	HiZ Analog Unb
95	P2[0]	LCD:LCDPort[0]		Strong	HiZ Analog Unb
96	P2[1]	LCD:LCDPort[1]		Strong	HiZ Analog Unb
97	P2[2]	LCD:LCDPort[2]		Strong	HiZ Analog Unb
98	P2[3]	LCD:LCDPort[3]		Strong	HiZ Analog Unb
99	P2[4]	LCD:LCDPort[4]		Strong	HiZ Analog Unb
100	Vio2	Vio2	Power		

Abbreviations used in Table 6 have the following meanings:

- GPIO = General Purpose IO
- SIO = Special IO
- Digital In = Digital input
- Res pull up = Resistive pull up
- Res pull dn = Resistive pull down
- Res pull up/dn = Resistive pull up/down
- Digital Out = Digital output
- Strong = Strong drive (digital output)
- Digital I/O = Bidirectional (I2C)
- HiZ analog = High impedance analog
- HiZ Analog Unb = Hi-Z Analog - unbuffered
- OD, DL = Open drain, drives low
- OD, DH = Open drain, drives high

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

- Pins chapter in the System Reference Guide (CyPins API routines)

- Programming Application Interface section in the cy_pins component datasheet

Accessing Unused Pins

In Table 6, several pins are marked as [unused]. These pins have no specific purpose in the design but can be accessed from firmware by using a library of pin macros is provided in the *cypins.h* file. These macros all make use of the port pin configuration register that is available for every pin on the PSoC 5LP device. The address of that register is provided in the *cydevice_trm.h* file. Each of these pin configuration registers is named:

- CYREG_PRTx_PCy

where x is the port number and y is the pin number within the port.

Table 7: Per-Pin APIs (for [unused] pins)

Function	Description
CyPins_ReadPin()	Reads the current value on the pin (pin state, PS)
CyPins_SetPin()	Set the output value for the pin (data register, DR) to a logic high. Note that this only has an effect for pins configured as software pins that are not driven by hardware.
CyPins_ClearPin()	Clear the output value for the pin (data register, DR) to a logic low. Note that this only has an effect for pins configured as software pins that are not driven by hardware.
CyPins_SetPinDriveMode()	Sets the drive mode for the pin (DM).
CyPins_ReadPinDriveMode()	Reads the drive mode for the pin (DM).

Using the cy_pins Component

Two of the pins listed in Table 6 can be managed from software; the pins for the two switches, SW2 and SW3.

However, in most cases the pins are associated with a specific hardware block such as CapSense, the LCD, or a communications or analog component. It is not recommended to change the behavior of these pins from firmware or to attempt to read/write them.

Important: Note that the string "*Pinname*" in the following APIs should be replaced with the name of the cy_pins component listed in Table 6, e.g. P6_1_Read().

Table 8: cy_pins APIs

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

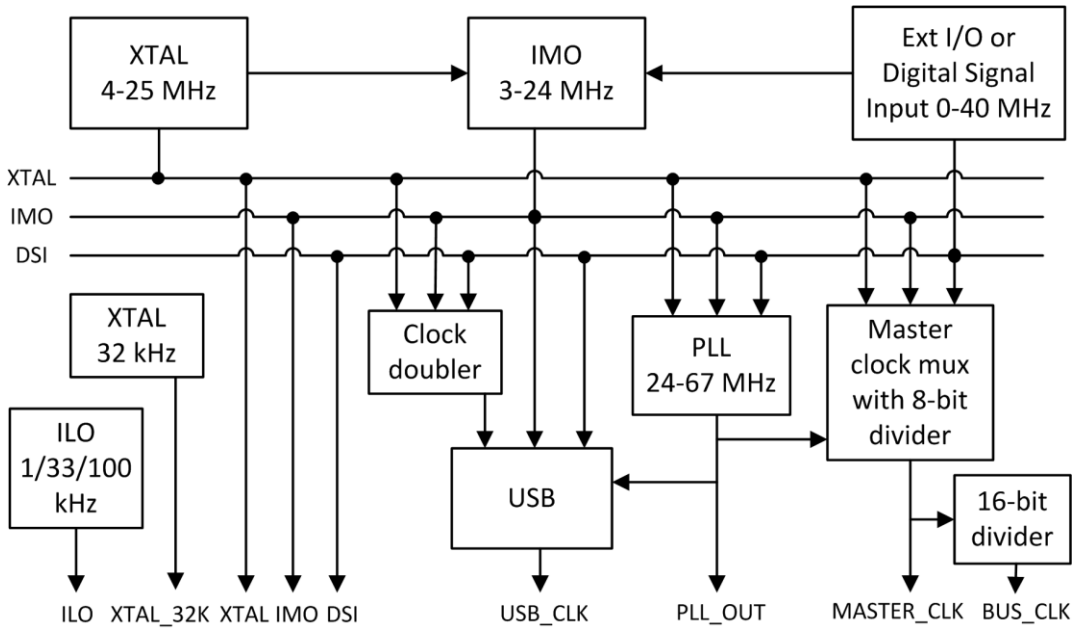
Function	Description
Pinname _SetDriveMode()	Sets the drive mode for each of the Pins component's pins
Pinname _ClearInterrupt()	Clears any active interrupts on the port into which the component is mapped. Returns value of interrupt status register

Clocks

The clock system includes these clock resources:

- Four internal clock sources increase system integration:
  - 3 to 48 MHz Internal Main Oscillator (IMO) ±5% at 3 MHz
  - 1 kHz, 33 kHz, 100 kHz Internal Low Speed Oscillator (ILO) outputs
  - USB Clock Domain, sourced from IMO, MHz External Crystal Oscillator (MHzECO), and Digital System Interconnect (DSI)
  - 24 to 67 MHz fractional Phase-Locked Loop (PLL) sourced from IMO, MHzECO, and DSI
- Clock generated using a DSI signal from an external I/O pin or other logic
- Two external clock sources provide high precision clocks:
  - 4 to 25 MHz External Crystal Oscillator (MHzECO)
  - 32.768 kHz External Crystal Oscillator (kHzECO) for Real Time Clock (RTC)
- Dedicated 16-bit divider for bus clock
- Eight individually sourced 16-bit clock dividers for the digital system peripherals
- Four individually sourced 16-bit clock dividers with skew for the analog system peripherals
- IMO has a USB mode that synchronizes to USB host traffic, requiring no external crystal for USB. (USB equipped parts only)
-

Figure 3: System Clock Configuration



System Clocks

Table 9 lists the system clocks used in this design.

Table 9: System Clocks

Name	Domain	Source	Freq (MHz)	Accuracy (%)	Start at Reset
USB_CLK	DIGITAL	IMO	N/A	±0	False
BUS_CLK	DIGITAL	MASTER_CLK	48	±1	True
MASTER_CLK	DIGITAL	PLL_OUT	48	±1	True
Digital Signal	DIGITAL		N/A	±0	False
XTAL 32kHz	DIGITAL		N/A	±0	False
XTAL	DIGITAL		N/A	±0	False
ILO	DIGITAL		0.001	-50 / +100	True
PLL_OUT	DIGITAL	IMO	48	±1	True
IMO	DIGITAL		3	±1	True

Local and Design Wide Clocks

Local clocks drive individual analog and digital blocks. Design wide clocks are a user-defined optimization, where two or more analog or digital blocks that share a common clock profile (frequency, etc) can be driven from the same clock divider output source.

Figure 4: Local and Design Wide Clock Configuration

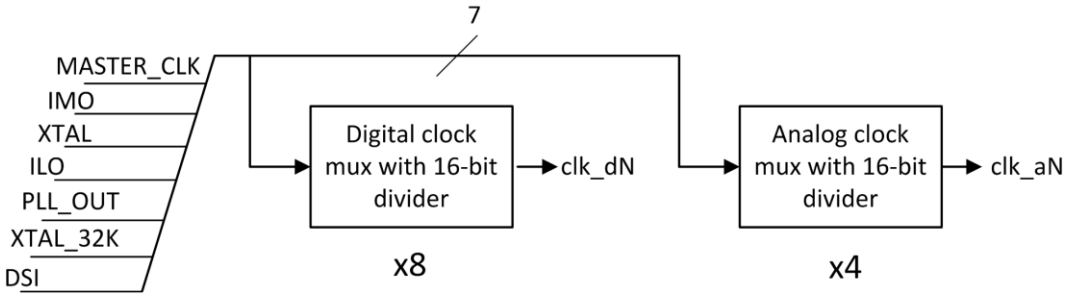


Table 10 lists the local clocks used in this design.

Table 10: Local Clocks

Name	Domain	Source	Freq (MHz)	Accuracy (%)	Start at Reset
UART_IntClock	DIGITAL	MASTER_CLK	0.4615	±1	True
PWM_1_Clock	DIGITAL	IMO	0.025	±1	True
PWM_2_Clock	DIGITAL	IMO	0.025	±1	True
Debouncer_Clock	DIGITAL	IMO	0.01	±1	True
Timer_Clock_1MHz	DIGITAL	MASTER_CLK	1.00	±1	True

Name	Domain	Source	Freq (MHz)	Accuracy (%)	Start at Reset
ADC_DelSig_theACLK	ANALOG	MASTER_CLK	3.0	±1	True
ADC_DelSig_Ext_CP_Clk	DIGITAL	MASTER_CLK	12.0	±1	True
ADC_SAR_1_theACLK	ANALOG	MASTER_CLK	1.7778	±1	True
ADC_SAR_2_theACLK	ANALOG	MASTER_CLK	1.7778	±1	True
CapSense_IntClock	DIGITAL	MASTER_CLK	12.0	±1	True
CapSense_Clock_tmp	DIGITAL	BUS_CLK	48	±1	True

For more information on clocking resources, please refer to:

- Clocking System chapter in the PSoC 5LP Technical Reference Manual
- Clocking chapter in the System Reference Guide
  - CyPLL API routines
  - CyIMO API routines
  - CyILO API routines
  - CyMaster API routines
  - CyXTAL API routines

Using the cy_clock Component

The local clocks listed in Table 10 can be managed from software using the following APIs. Note that changing the ADC and CapSense 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 Table 10, e.g. PWM_1_Start().

Table 11: cy_clock APIs

Function	Description
*Clockname*_Start()	Enables the clock.
*Clockname*_Stop()	Disables the clock.
*Clockname*_StopBlock()	Disables the clock and waits until the clock is disabled.
*Clockname*_StandbyPower()	Selects the power for standby (Alternate Active) operation mode.
*Clockname*_SetDivider()	Sets the divider of the clock and restarts the clock divider immediately.
*Clockname*_SetDividerRegister()	Sets the divider of the clock and optionally restarts the clock divider immediately.
*Clockname*_SetDividerValue()	Sets the divider of the clock and restarts the clock divider immediately.
*Clockname*_GetDividerRegister()	Gets the clock divider register value.
*Clockname*_SetMode()	Sets flags that control the operating mode of the clock.
*Clockname*_SetModeRegister()	Sets flags that control the operating mode of the clock.

Function	Description
*Clockname*_GetModeRegister()	Gets the clock mode register value.
*Clockname*_ClearModeRegister()	Clears flags that control the operating mode of the clock.
*Clockname*_SetSource()	Sets the source of the clock.
*Clockname*_SetSourceRegister()	Sets the source of the clock.
*Clockname*_GetSourceRegister()	Gets the source of the clock.
*Clockname*_SetPhase()	Sets the phase delay of the analog clock (only generated for analog clocks).
*Clockname*_SetPhaseRegister()	Sets the phase delay of the analog clock (only generated for analog clocks).
*Clockname*_SetPhaseValue()	Sets the phase delay of the analog clock (only generated for analog clocks).
*Clockname*_GetPhaseRegister()	Gets the phase delay of the analog clock (only generated for analog clocks).

Interrupts

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

Table 12: Interrupts

Name	Priority	Vector	Integrated into Component
ADC_DelSig_Interrupt	7	29	Yes
ADC_DelSig_IRQ	7	13	Yes
ADC_SAR_1_Interrupt	7	0	Yes
ADC_SAR_1_IRQ	7	14	Yes
ADC_SAR_2_Interrupt	7	1	Yes
ADC_SAR_2_IRQ	7	16	Yes
CapSense_IsrCH0	7	17	Yes
Comp_1_High_ISR	7	2	No
Comp_1_Low_ISR	7	3	No
Comp_2_High_ISR	7	4	No
Comp_2_Low_ISR	7	5	No
Comp_3_High_ISR	7	6	No
Comp_3_Low_ISR	7	7	No
I2C_isr	7	15	Yes
SW2_Interrupt	7	8	No
SW3_Interrupt	7	9	No
Timer_1MHz_Interrupt	7	10	No
UART_RX_Interrupt	7	11	No
UART_TX_Interrupt	7	12	No

For more information on interrupts, please refer to:

- Interrupt Controller chapter in the PSoC 5LP 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 12 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” in Table 12) 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 12, e.g. SW2_Interrupt_StartEx().

Table 13: 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.
*ISRname*_Disable()	Disables the interrupt.
*ISRname*_SetPending()	Causes the interrupt to enter the pending state, a software method of generating the interrupt.

Function	Description
*ISRname*_ClearPending()	Clears a pending interrupt.

DMA

This design contains no DMA components.

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.

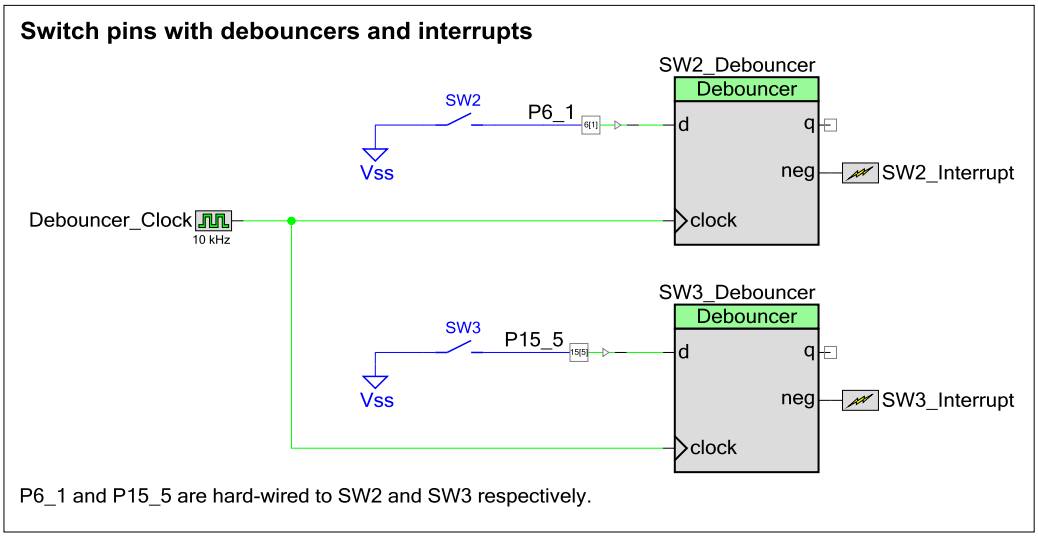
Switches

The CY8CKIT-050 has mechanical switches attached to pins P6_1 (SW2) and P15_5 (SW3).

The cy_pins component APIs can be used to read the state of the switches (e.g. P6_1_Read()). Note that the inputs are grounded on the board, meaning that the *_Read() APIs return 0 when the switch is pressed and 1 when it is open.

The Debouncer components remove "ringing" when the switches are pressed (no need for software debouncing) and assert individual interrupts (on negative edges) for each switch.

Figure 5: Switches



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

- SW1_Debouncer and SW2_Debouncer (type: Debouncer_v1_0)
- Debouncer_Clock (type: cy_clock_v2_10)
- P6_1 and P15_5 (type cy_pins_v1_90)
- SW2_Interrupt and SW3_Interrupt (type cy_isr_v1_70)

Table 14: SW*n*_Debouncer Parameters

Parameter	Value	Description
EitherEdgeDetect	False	Specifies whether positive or negative edge detection is enabled
NegEdgeDetect	True	Specifies whether negative edge detection is enabled
PosEdgeDetect	False	Specifies whether positive edge detection is enabled

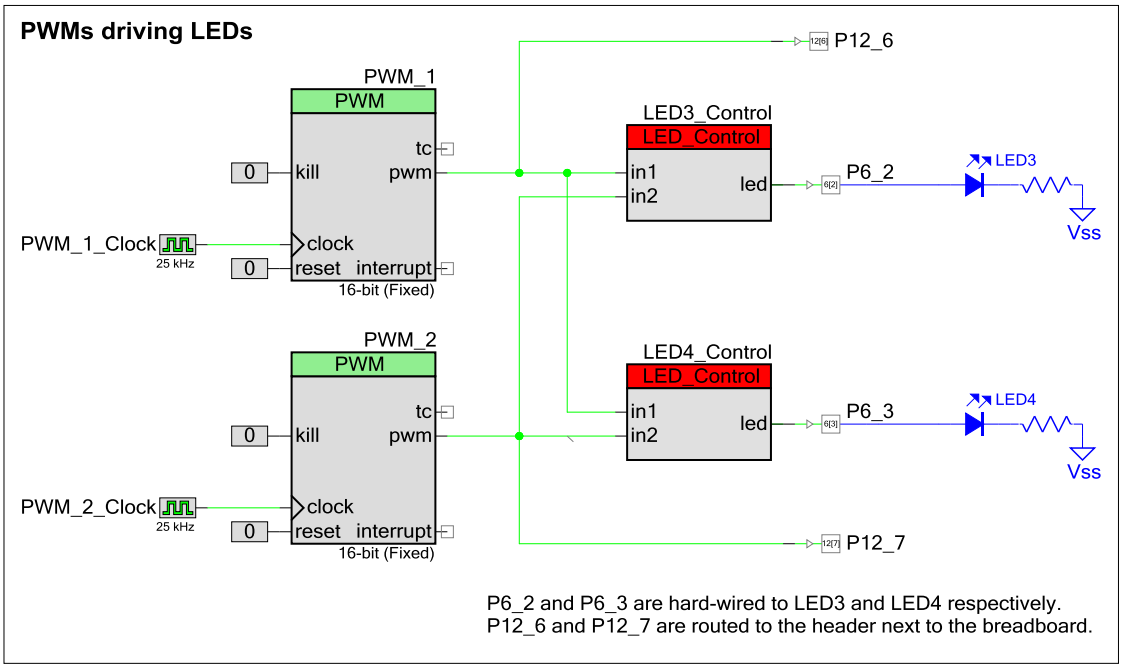
SignalWidth	1	Width of input and output terminals.

LEDs

The CY8CKIT-050 has LEDs attached to pins P6_2 (LED3) and P6_3 (LED4). These pins are driven by custom LED_Control components. The LED_Control components enable the selection of the signal to route to the LEDs (software, in1, in2, in1&in2, in1^in2).

The PWMs are also connected directly to P12_6 and P12_7. This allows the PWM outputs to be available in the breadboard area even when LED3 and LED4 are being driven by software.

Figure 6: LEDs



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

- LED3_Control and LED4_Control (type: LED_Control_v1_00)
  - Note that the LED_Control component was created specifically for this BSP to enable different uses of the LEDs on the board. The datasheet is not available on the web but is included in each of the project files
  - CY8CKIT-050-***\Demo.cydsn\LED_Control_v1_00\LED_Control_v1_00.pdf)
- PWM_1 and PWM_2 (type: PWM_v3_0)
- PWM_1_Clock and PWM_2_Clock (type: cy_clock_v2_10)
- P6_2, P6_3, P12_6 and P12_7 (type cy_pins_v1_90)
- SW2_Interrupt and SW3_Interrupt (type cy_isr_v1_70)

Table 15: LED*n*_Control Parameters

Parameter	Value	Description
channel	Software	Selects the signal driven to the output

Table 16: LED*n*_Control APIs

Function	Description
void LED*n*_Control_Set_Channel(int use)	Chooses which signal to route to the output/LED. Reads the register, modifies the bits that control the signal selection, and writes the new value back to the register.
int LED*n*_Control_Get_Channel(void)	Returns the value of the software bit in the control register. If the output is driven from the schematic the return value is undefined.
void LED*n*_Control_Write(int value)	Writes to the LED when it is driven from software. Reads the register and checks that the selected output is software-driven. If so, it updates the bit that drives the output based on the function argument. If the output is driven from the schematic this API has no effect.
int LED*n*_Control_Read(void)	Returns the value of the software bit in the control register. If the output is driven from the schematic the return value is undefined.

Table 17: PWM_*n* Parameters

Parameter	Value	Description
Resolution	16-bits	The bit-width of the period counter
Period	249 (10ms) for PWM_1 248 (9.96ms) for PWM_2	Initial (and reload) value of the counter
Compare	125	PWM output compare value
CompareType	Greater than or Equal	The PWM output will be high if the Period is greater than or equal to the Compare

Figure 7: PWM_*n* APIs

Function	Description
PWM_*n*_Start()	Initializes the PWM with default customizer values.
PWM_*n* _Stop()	Disables the PWM operation. Clears the enable bit of the control register for either of the software controlled enable modes.
PWM_*n* _SetInterruptMode()	Configures the interrupts mask control of the interrupt source status register.
PWM_*n* _ReadStatusRegister()	Returns the current state of the status register.
PWM_*n* _ReadControlRegister()	Returns the current state of the control register.
PWM_*n* _WriteControlRegister()	Sets the bit field of the control register.

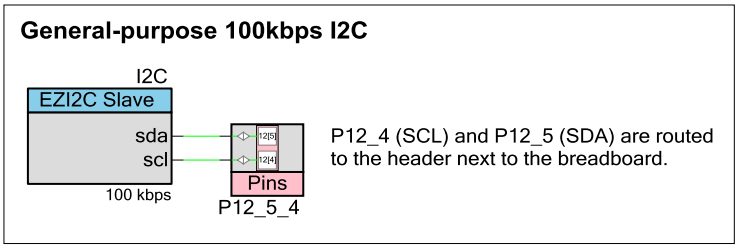
Function	Description
PWM_*n* _SetCompareMode()	Writes the compare mode for compare output when set to Dither mode, Center Align mode or One Output mode.
PWM_*n* _SetCompareMode1()	Writes the compare mode for compare1 output into the control register.
PWM_*n* _SetCompareMode2()	Writes the compare mode for compare2 output into the control register.
PWM_*n* _ReadCounter()	Reads the current counter value (software capture).
PWM_*n* _ReadCapture()	Reads the capture value from the capture FIFO.
PWM_*n* _WriteCounter()	Writes a new counter value directly to the counter register. This will be implemented only for that currently running period.
PWM_*n* _WritePeriod()	Writes the period value used by the PWM hardware.
PWM_*n* _ReadPeriod()	Reads the period value used by the PWM hardware.
PWM_*n* _WriteCompare()	Writes the compare value when the instance is defined as Dither mode, Center Align mode or One Output mode.
PWM_*n* _ReadCompare()	Reads the compare value when the instance is defined as Dither mode, Center Align mode or One Output mode.
PWM_*n* _WriteCompare1()	Writes the compare value for the compare1 output.
PWM_*n* _ReadCompare1()	Reads the compare value for the compare1 output.
PWM_*n* _WriteCompare2()	Writes the compare value for the compare2 output
PWM_*n* _ReadCompare2()	Reads the compare value for the compare2 output.
PWM_*n* _WriteDeadTime()	Writes the dead time value used by the hardware in dead band implementation.
PWM_*n* _ReadDeadTime()	Reads the dead time value used by the hardware in dead band implementation.
PWM_*n* _WriteKillTime()	Writes the kill time value used by the hardware when the kill mode is set as Minimum Time.
PWM_*n* _ReadKillTime()	Reads the kill time value used by the hardware when the kill mode is set as Minimum Time.

Function	Description
PWM_*n* _ClearFIFO()	Clears all capture data from the capture FIFO.
PWM_*n* _Sleep()	Stops and saves the user configuration.
PWM_*n* _Wakeup()	Restores and enables the user configuration.
PWM_*n* _Init()	Initializes component's parameters to those set in the customizer placed on the schematic.
PWM_*n* _Enable()	Enables the PWM block operation.
PWM_*n* _SaveConfig()	Saves the current user configuration of the component.
PWM_*n* _RestoreConfig()	Restores the current user configuration of the component.

Communications

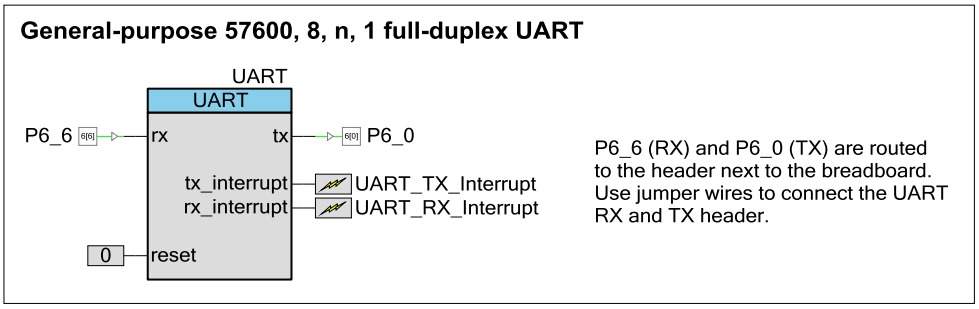
The I2C is connected to P12_4 (SCL) and P12_5 (SDA), which are available on the header next to the breadboard.

Figure 8: I2C



The UART is connected to P6_0 (TX) and P6_6 (RX), which are available on the header next to the breadboard. Use jumper wires to connect to the TX and RX header, next to the D-sub connector.

Figure 9: UART



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

- I2C (type: EZI2C_v1_90)
- UART (type: UART_v2_30)
- P6_6 and P16_0 (type cy_pins_v1_90)
- UART_TX_Interrupt and UART_RX_Interrupt (type cy_isr_v1_70)

Table 18: I2C Parameters

Parameter	Value	Description
Data Rate	100kbps	Used to set the I2C data rate value up to 1000 kbps.The standard data rates are 50, 100 (default), 400, and 1000 kbps.
Slave Address	0x08	I2C slave address - this address is the 7-bit right-justified slave address and does not include the R/W bit.
Sub-address Size	8-bit	Determines what range of data can be accessed.

Table 19: I2C APIs

Function	Description
I2C_Start()	Starts responding to I2C traffic. Enables interrupt.
I2C_Stop()	Stops responding to I2C traffic. Disables interrupt.
I2C_EnableInt()	Enables interrupt, which is required for most I2C operations.
I2C_DisableInt()	Disables interrupt. The I2C_Stop() API does this automatically.
I2C_SetAddress1()	Sets the primary I2C address.
I2C_GetAddress1()	Returns the primary I2C address.
I2C_SetBuffer1()	Sets the buffer pointer for the primary I2C.
I2C_GetActivity(void)	Checks component activity status.
I2C_Sleep()	Stops I2C operation and saves I2C configuration. Disables interrupt.
I2C_Wakeup()	Restores I2C configuration and starts I2C operation. Enables interrupt.
I2C_Init()	Initializes I2C registers with initial values provided from customizer.
I2C_Enable()	Activates the hardware and begins component operation.
I2C_SaveConfig()	Saves the current user configuration of the EZI2C component.
I2C_RestoreConfig()	Restores nonretention I2C registers.

Table 20: I2C Sleep/Wake APIs

Function	Description
I2C_SlaveSetSleepMode()	Deprecated in v1.90. Use I2C_Sleep() instead.
I2C_SlaveSetWakeMode()	Deprecated in v1.90. Use I2C_Wakeup() instead.

Table 21: UART Parameters

Parameter	Value	Description
Mode	Full	Full UART (TX + RX)
Bits per Second	57600	The baud-rate or bit-width configuration of the hardware for clock generation
Data Bits	8	The number of data bits transmitted between start and stop of a single UART transaction
Parity Type	None	Defines the functionality of the parity

		bit location in the transfer
API Control of Parity	Disabled	The UART parity may not be changed by the UART_WriteControlRegister() API.
Stop Bits	1	Defines the number of stop bits implemented in the transmitter
Flow Control	None	CTS and RTS signals are disabled
Rx_Interrrupt	Byte received Parity error Stop error Break Overrun error	Recognized causes of an Rx interrupt
Tx_Interrrupt	Tx complete	Recognized causes of a Tx interrupt

Table 22: UART APIs

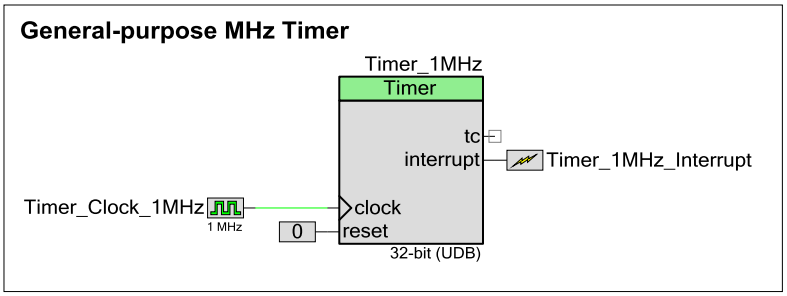
Function	Description
UART_Start()	Initializes and enables the UART operation
UART_Stop()	Disables the UART operation
UART_ReadControlRegister()	Returns the current value of the control register
UART_WriteControlRegister()	Writes an 8-bit value into the control register
UART_EnableRxInt()	Enables the internal interrupt irq
UART_DisableRxInt()	Disables the internal interrupt irq
UART_SetRxInterruptMode()	Configures the RX interrupt sources enabled
UART_ReadRxData()	Returns the data in the RX Data register
UART_ReadRxStatus()	Returns the current state of the status register
UART_GetChar()	Returns the next byte of received data
UART_GetByte()	Reads the UART RX buffer immediately and returns the received character and error condition
UART_GetRxBufferSize()	Returns the number of received bytes remaining in the RX buffer and returns the count in bytes
UART_ClearRxBuffer()	Clears the memory array of all received data
UART_SetRxAddressMode()	Sets the software-controlled Addressing mode used by the RX portion of the UART
UART_SetRxAddress1()	Sets the first of two hardware-detectable addresses
UART_SetRxAddress2()	Sets the second of two hardware-detectable addresses
UART_EnableTxInt()	Enables the internal interrupt irq
UART_DisableTxInt()	Disables the internal interrupt irq
UART_SetTxInterruptMode()	Configures the TX interrupt sources enabled

Function	Description
UART_WriteTxData()	Sends a byte without checking for buffer room or status
UART_ReadTxStatus()	Reads the status register for the TX portion of the UART
UART_PutChar()	Puts a byte of data into the transmit buffer to be sent when the bus is available
UART_PutString()	Places data from a string into the memory buffer for transmitting
UART_PutArray()	Places data from a memory array into the memory buffer for transmitting
UART_PutCRLF()	Writes a byte of data followed by a Carriage Return and Line Feed to the transmit buffer
UART_GetTxBufferSize()	Determines the number of bytes used in the TX buffer. An empty buffer returns 0
UART_ClearTxBuffer()	Clears all data from the TX buffer
UART_SendBreak()	Transmits a break signal on the bus
UART_SetTxAddressMode ()	Configures the transmitter to signal the next bytes as address or data
UART_LoadRxConfig()	Loads the receiver configuration. Half Duplex UART is ready for receive byte
UART_LoadTxConfig()	Loads the transmitter configuration. Half Duplex UART is ready for transmit byte
UART_Sleep()	Stops the UART operation and saves the user configuration
UART_Wakeup()	Restores and enables the user configuration
UART_Init()	Initializes default configuration provided with customizer
UART_Enable()	Enables the UART block operation
UART_SaveConfig()	Save the current user configuration
UART_RestoreConfig()	Restores the user configuration

Timer

The Timer_1MHz is a 32-bit timer with a 1Mhz (1ms precision) input clock. By default the Timer_1MHz_Interrupt is asserted on terminal count.

Figure 10: MHz Timer



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

- Timer_1MHz (type: Timer_v2_50)
- Timer_Clock_1MHz (type: cy_clock_v2_10)
- Timer_1MHz_Interrupt (type cy_isr_v1_70)

Table 23: Timer_1MHz Parameters

Parameter	Value	Description
Resolution	32-bit	Bit-width resolution
Period	4294967296	Defines the period of the counter, which is one greater than the maximum count value (or rollover point)
Trigger Mode	None	Trigger mode is not enabled.
Capture Mode	None	Capture mode is not enabled.
Enable Mode	None	Enable mode is not enabled.
Run Mode	Continuous	Runs continuously while it is enabled
Interrupts	On TC	Generate an interrupt on terminal count

Table 24: Timer_1MHz APIs

Function	Description
Timer_1MHz_Start()	Sets the initVar variable, calls the Timer_Init() function, and then calls the Enable function.
Timer_1MHz_Stop()	Disables the Timer.
Timer_1MHz_SetInterruptMode()	Enables or disables the sources of the interrupt output.
Timer_1MHz_ReadStatusRegister()	Returns the current state of the status register.
Timer_1MHz_ReadControlRegister()	Returns the current state of the control register.
Timer_1MHz_WriteControlRegister()	Sets the bit-field of the control register.

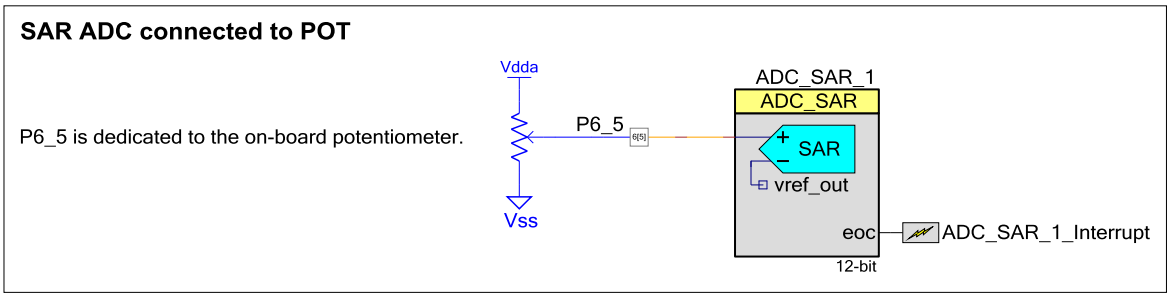
Function	Description
Timer_1MHz_WriteCounter()	Writes a new value directly into the counter register. (UDB only)
Timer_1MHz_ReadCounter()	Forces a capture, and then returns the capture value.
Timer_1MHz_WritePeriod()	Writes the period register.
Timer_1MHz_ReadPeriod()	Reads the period register.
Timer_1MHz_ReadCapture()	Returns the contents of the capture register or the output of the FIFO.
Timer_1MHz_SetCaptureMode()	Sets the hardware or software conditions under which a capture will occur.
Timer_1MHz_SetCaptureCount()	Sets the number of capture events to count before capturing the counter register to the FIFO.
Timer_1MHz_ReadCaptureCount()	Reports the current setting of the number of capture events.
Timer_1MHz_SoftwareCapture()	Forces a capture of the counter to the capture FIFO
Timer_1MHz_SetTriggerMode()	Sets the hardware or software conditions under which a trigger will occur.
Timer_1MHz_EnableTrigger()	Enables the trigger mode of the timer.
Timer_1MHz_DisableTrigger()	Disables the trigger mode of the timer.
Timer_1MHz_SetInterruptCount()	Sets the number of captures to count before an interrupt is triggered.
Timer_1MHz_ClearFIFO()	Clears the capture FIFO.
Timer_1MHz_Sleep()	Stops the Timer and saves its current configuration.
Timer_1MHz_Wakeup()	Restores the Timer configuration and re-enables the Timer.
Timer_1MHz_Init()	Initializes or restores the Timer per the Configure dialog settings.
Timer_1MHz_Enable()	Enables the Timer.
Timer_1MHz_SaveConfig()	Saves the current configuration of the Timer.
Timer_1MHz_RestoreConfig()	Restores the configuration of the Timer.

Analog-Digital Converters (ADC)

There are three ADCs in the design. All are free-running (after the *_StartConvert() API is called) and generate an interrupt at the end of a conversion.

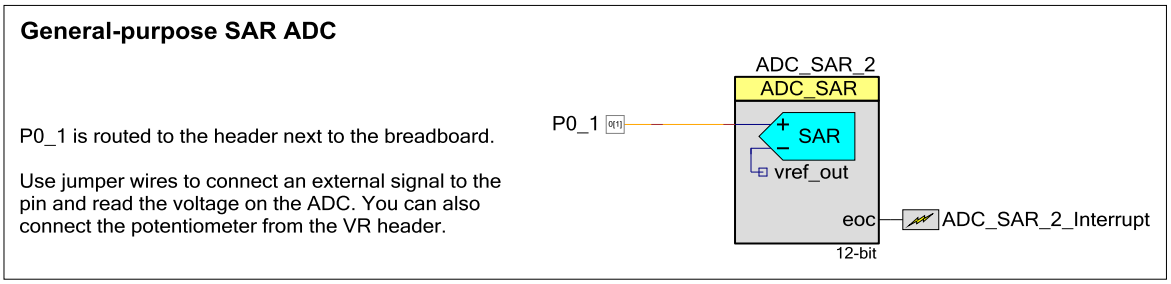
ADC_SAR_1 is connected directly to the on-board potentiometer.

Figure 11: SAR ADC 1



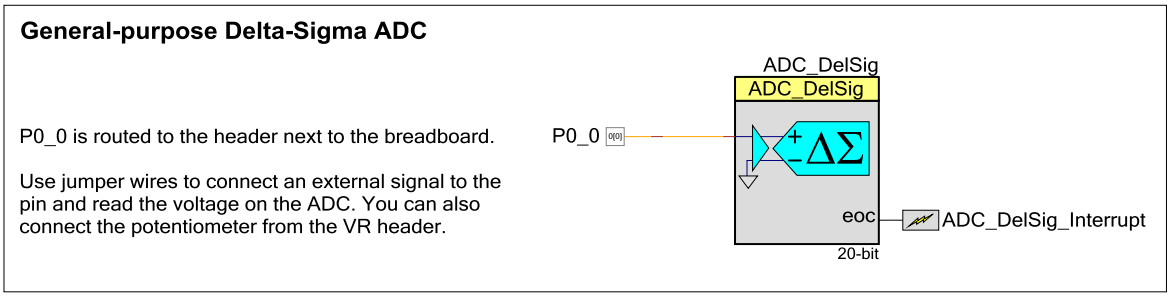
ADC_SAR_2 is connected directly to P0_1, which is near the breadboard area.

Figure 12: SAR ADC 2



ADC_DelSig is connected directly to P0_0, which is near the breadboard area.

Figure 13: Delta-Sigma ADC



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

- ADC_DelSig (type: ADC_DelSig_v3_0)
- ADC_SAR_1 and ADC_SAR_2 (type: ADC_SAR_v2_10)

- P6_5, P0_0 and P0_1 (type cy_pins_v1_90)
- ADC_SAR_1_Interrupt, ADC_SAR_2_Interrupt and ADC_DelSig_Interrupt (type cy_isr_v1_70)

Table 25: ADC_SAR_*n* Parameters

Parameter	Value	Description
Resolution	12-bits	Size of read data
Conversion Rate	100000sps	The conversion rate in samples per second - the conversion time is the inverse of the conversion rate
Sample Mode	Free Running	ADC runs continuously
Input Range	Vssa to Vdda (Single Ended)	Uses the VDDA/2 reference; the usable input range covers the full analog supply voltage. The ADC is put in a single-ended input mode with the – Input connected internally to Vrefhi_out.

Table 26: ADC_SAR_*n* APIs

Function	Description
ADC_SAR_*n*_Start()	Powers up the ADC and resets all states
ADC_SAR_*n*_Stop()	Stops ADC conversions and reduces the power to the minimum
ADC_SAR_*n*_SetPower()	Sets the power mode
ADC_SAR_*n*_SetResolution()	Sets the resolution of the ADC
ADC_SAR_*n*_StartConvert()	Starts conversions
ADC_SAR_*n*_StopConvert()	Stops conversions
ADC_SAR_*n*_IRQ_Enable()	An internal IRQ is connected to the eoc. This API enables the internal ISR.
ADC_SAR_*n*_IRQ_Disable()	An internal IRQ is connected to the eoc. This API disables the internal ISR.
ADC_SAR_*n*_IsEndConversion()	Returns a nonzero value if conversion is complete
ADC_SAR_*n*_GetResult8()	Returns a signed 8-bit conversion result
ADC_SAR_*n*_GetResult16()	Returns a signed16-bit conversion result
ADC_SAR_*n*_SetOffset()	Sets the offset of the ADC
ADC_SAR_*n*_SetGain()	Sets the ADC gain in counts per volt
ADC_SAR_*n*_CountsTo_Volts()	Converts ADC counts to floating-point volts
ADC_SAR_*n*_CountsTo_mVolts()	Converts ADC counts to millivolts
ADC_SAR_*n*_CountsTo_uVolts()	Converts ADC counts to microvolts
ADC_SAR_*n*_Sleep()	Stops ADC operation and saves the user configuration
ADC_SAR_*n*_Wakeup()	Restores and enables the user configuration

Function	Description
ADC_SAR_*n*_Init()	Initializes the default configuration provided with the customizer
ADC_SAR_*n*_Enable()	Enables the clock and power for the ADC
ADC_SAR_*n*_SaveConfig()	Saves the current user configuration
ADC_SAR_*n*_RestoreConfig()	Restores the user configuration

Table 27: ADC_DelSig Parameters

Parameter	Value	Description
Resolution	20-bits	Size of read data
Conversion Rate	187sps	The conversion rate in samples per second - the conversion time is the inverse of the conversion rate
Sample Mode	Continuous	Continuous sample mode operates as a normal delta-sigma converter, taking repeated samples at the conversion rate. There is a latency of three conversion times before the first conversion result is available. This is the time required to prime the internal filter. After the first result, a conversion will be available at the selected sample rate.
Configs	1	The number of different configurations of the ADC – selected by the ADC_SelectConfiguration() API.
Input Range	Vssa to Vdda (Single ended)	This mode is ratiometric of the supply voltage.

Table 28: ADC_DelSig APIs

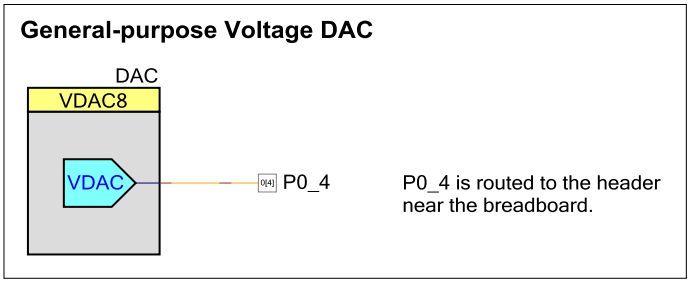
Function	Description
ADC_DelSig_Start()	Sets the initVar variable, calls the ADC_DelSig_Init() function, and then calls the ADC_DelSig_Enable() function.
ADC_DelSig_Stop()	Stops ADC conversions and powers down.
ADC_DelSig_SetBufferGain()	Selects input buffer gain (1,2,4,8)
ADC_DelSig_StartConvert()	Starts conversion.
ADC_DelSig_StopConvert()	Stops conversions
ADC_DelSig_IRQ_Enable()	Enables interrupts at end of conversion.
ADC_DelSig_IRQ_Disable()	Disables interrupts.
ADC_DelSig_IsEndConversion()	Returns a nonzero value if conversion is complete.

Function	Description
ADC_DelSig_GetResult8()	Returns an 8-bit conversion result, right justified.
ADC_DelSig_GetResult16()	Returns a 16-bit conversion result, right justified.
ADC_DelSig_GetResult32()	Returns a 32-bit conversion result, right justified.
ADC_DelSig_Read8()	Returns an 8-bit conversion result. Blocking call that starts a conversion, obtains the result, then stops the conversion.
ADC_DelSig_Read16()	Returns a 16-bit conversion result. Blocking call that starts a conversion, obtains the result, then stops the conversion.
ADC_DelSig_Read32()	Returns a 32-bit conversion result. Blocking call that starts a conversion, obtains the result, then stops the conversion.
ADC_DelSig_SetOffset()	Sets the offset used by the ADC_DelSig_CountsTo_mVolts(), ADC_DelSig_CountsTo_uVolts(), and ADC_DelSig_CountsTo_Volts() functions.
ADC_DelSig_SelectConfiguration()	Sets one of up to four ADC configurations.
ADC_DelSig_SetGain()	Sets the gain used by the ADC_DelSig_CountsTo_mVolts(), ADC_DelSig_CountsTo_uVolts(), and ADC_DelSig_CountsTo_Volts() functions.
ADC_DelSig_CountsTo_mVolts()	Converts ADC counts to mV.
ADC_DelSig_CountsTo_uVolts()	Converts ADC counts to µV.
ADC_DelSig_CountsTo_Volts()	Converts ADC counts to floating point volts.
ADC_DelSig_Sleep()	Stops ADC operation and saves the user configuration.
ADC_DelSig_Wakeup()	Restores and enables the user configuration.
ADC_DelSig_Init()	Initializes or restores the ADC using the Configure dialog settings.
ADC_DelSig_Enable()	Enables the ADC.
ADC_DelSig_SaveConfig()	Saves the current configuration.
ADC_DelSig_RestoreConfig()	Restores the configuration.
ADC_DelSig_SetCoherency()	Sets the coherency register.
ADC_DelSig_SetGCOR()	Calculates a new GCOR value and sets the GCOR registers with this new value.
ADC_DelSig_ReadGCOR()	Returns the normalized GCOR register values.

Digital-Analog Converter (DAC)

A voltage DAC is connected to P0_4, which is near the breadboard area.

Figure 14: Voltage DAC



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

- DAC (type: VDAC8_v1_90)
- P0_4 (type cy_pins_v1_90)

Table 29: DAC Parameters

Parameter	Value	Description
Range	4 Volt	The DAC will generate a voltage between 0.000V and 4.080V, in 16mV increments. Note that Vdda on the kit is 3.3V and so the DAC output will saturate at that voltage.
Initial Value	0	This is the initial value the VDAC8 presents after the DAC_Start() command is executed.
Speed	High	The DAC is set to high speed mode, where the voltage settles quickly, but at a cost of more operating current.

Table 30: DAC APIs

Function	Description
DAC_Start()	Initializes the VDAC8 with default customizer values.
DAC_Stop()	Disables the VDAC8 and sets it to the lowest power state.
DAC_SetSpeed()	Sets DAC speed.
DAC_SetValue()	Sets value between 0 and 255 with the given range.
DAC_SetRange()	Sets range to 1 or 4 volts.
DAC_Sleep()	Stops and saves the user configuration.
DAC_WakeUp()	Restores and enables the user configuration.

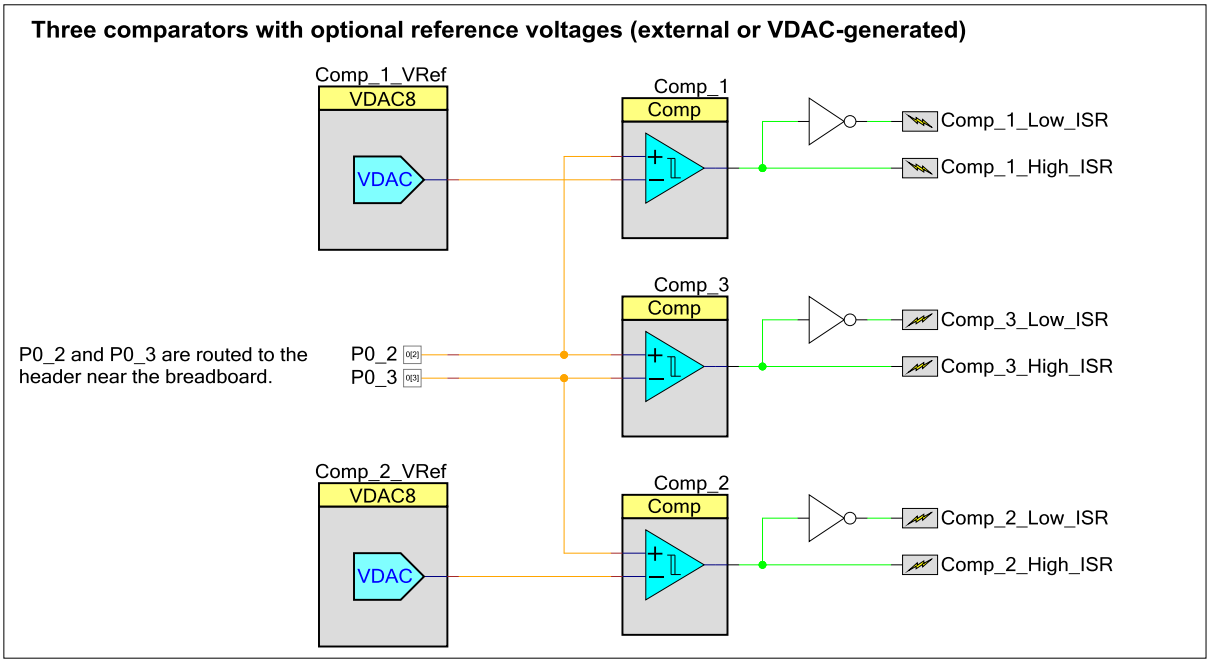
DAC_Init()	Initializes or restores default VDAC8 configuration
DAC_Enable()	Enables the VDAC8.
DAC_SaveConfig()	Saves nonretention DAC data register value.
DAC_RestoreConfig()	Restores nonretention DAC data register value

Comparators

There are three Comparators, each of which is configured with 10mV of hysteresis. Reference voltages can be sourced from P0_3 or one of two DACs (*Comp_n_Ref*). Each one can perform a different comparison and generate interrupts on both a rising (*Comp_n_High_ISR)* or falling edge (*Comp_n_Low_ISR*).

- Comp_1 compares the voltage on P0_2 with the output of the Comp_1_VRef DAC.
- Comp_2 compares the voltage on P0_3 with the output of the Comp_2_VRef DAC.
- Comp_3 compares the voltage on P0_2 with the voltage on P0_3.

Figure 15: Comparators



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

- Comp_1, Comp_2 and Comp_3 (type: Comp_v2_0)
- Comp_1_VRef and Comp_2_VRef (type: VDAC8_v1_90)
- P0_2 and P0_3 (type cy_pins_v1_90)
- Comp_1/2/3_High/Low_ISR (type cy_isr_v1_70)

Table 31: Comp_*n* Parameters

Parameter	Value	Description
Hysteresis	Enabled	Adds approximately 10 mV of hysteresis to the comparator
Speed	Slow	Response time is slower than 80ns
PowerDownOverride	Disabled	The Comparator does not remain active in sleep mode
Polarity	Non-inverting	Output goes high when positive input is greater than negative input

Table 32: Comp_*n* APIs

Function	Description
Comp_*n*_Start()	Initializes the Comparator with default customizer values.
Comp_*n* _Stop()	Turns off the Comparator.
Comp_*n* _SetSpeed()	Sets speed of the Comparator.
Comp_*n* _ZeroCal()	Zeros the input offset of the Comparator.
Comp_*n* _GetCompare()	Returns compare result.
Comp_*n* _LoadTrim()	Writes a value to the Comparator trim register.
Comp_*n* _Sleep()	Stops Comparator operation and saves the user configuration.
Comp_*n* _Wakeup()	Restores and enables the user configuration.
Comp_*n* _Init()	Initializes or restores default Comparator configuration.
Comp_*n* _Enable()	Enables the Comparator.
Comp_*n* _SaveConfig()	Empty function. Provided for future use.
Comp_*n* _RestoreConfig()	Empty function. Provided for future use.

Table 33: Comp_*n*_VRef Parameters

Parameter	Value	Description
Range	4 Volt	The DAC will generate a voltage between 0.000V and 4.080V, in 16mV increments. Note that Vdda on the kit is 3.3V and so the DAC output will saturate at that voltage.
Initial Value	1104mV (0x45) in Comp_1_VRef 2208 (0x8A) in Comp_2_VRef	This is the initial value the VDAC8 presents after the DAC_Start() command is executed.
Speed	Slow	The DAC is set to low speed mode, where the voltage settles slowly, but operating current is reduced.

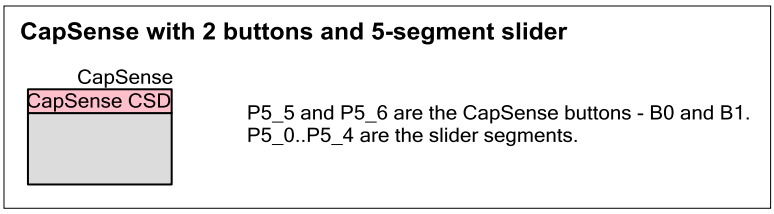
Table 34: Comp_*n*_VRef APIs

Function	Description
Comp_*n*_VRef*()	See Table 30: DAC APIs above for VDAC8 API information. Note that the API prefix “DAC” should be replaced with “Comp_*n*_VRef”.

CapSense

The CapSense buttons are connected to P5_5 (B0) and P5_6 (B1). The Slider is connected to five pins, P5_0..P5_4.

Figure 16: CapSense



This schematic sheet contains the following component instance (click link for online component datasheet with details on features and APIs).

- CapSense (type: CapSense_CSD_v3_40)

Table 35: General 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 nonretention registers. Resets all sensors to an inactive state.
CapSense_RestoreConfig()	Restores CapSense configuration and nonretention register values.

Table 36:Scanning APIs

Function	Description
CapSense_ScanSensor()	Sets scan settings and starts scanning a sensor or group of combined sensors on each channel.
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 or pair of sensors).
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_SetRBleed()	Sets the pin to use for the bleed resistor (Rb) connection if multiple bleed resistors are used.

Table 37: High-Level APIs

Function	Description
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.

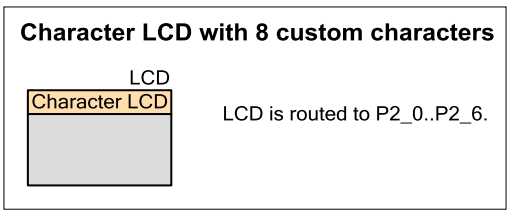
Function	Description
CapSense_EnableWidget()	Enables all sensor elements in a widget for the scanning process.
CapSense_DisableWidget()	Disables all sensor elements in a widget from the scanning process.
CapSense_CheckIsWidgetActive()	Compares the selected of widget to the CapSense_Signal[] array to 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.

LCD

The LCD is connected to P20..P2_6. It is configured with the following special characters.

- CHAR_LO Used by the scope demo to indicate a low signal
- CHAR_HI Used by the scope demo to indicate a high signal
- CHAR_RISE Used by the scope demo to indicate a rising signal
- CHAR_FALL Used by the scope demo to indicate a falling signal
- CHAR_SINEH High phase of a sine wave
- CHAR_SINEL Low phase of a sine wave
- CHAR_MU Greek mu symbol
- CHAR_HEART Cute heart shape

Figure 17: LCD



This schematic sheet contains the following component instance (click link for online component datasheet with details on features and APIs).

- LCD (type: CharLCD_v1_90)

Table 38: LCD APIs

Function	Description
LCD_Start()	Starts the module and loads custom character set to LCD, if it was defined.

Function	Description
LCD_Stop()	Turns off the LCD
LCD_DisplayOn()	Turns on the LCD module's display
LCD_DisplayOff()	Turns off the LCD module's display
LCD_PrintString()	Prints a null-terminated string to the screen, character by character
LCD_PutChar()	Sends a single character to the LCD module data register at the current position.
LCD_Position()	Sets the cursor's position to match the row and column supplied
LCD_WriteData()	Writes a single byte of data to the LCD module data register
LCD_WriteControl()	Writes a single-byte instruction to the LCD module control register
LCD_ClearDisplay()	Clears the data from the LCD module's screen
LCD_IsReady()	Polls LCD until the ready bit is set
LCD_Sleep()	Prepares component for entering sleep mode
LCD_Wakeup()	Restores components configuration and turns on the LCD
LCD_Init()	Performs initialization required for component's normal work
LCD_Enable()	Turns on the display
LCD_SaveConfig()	Empty API provided to store any required data prior entering to a Sleep mode.
LCD_RestoreConfig()	Empty API provided to restore saved data after exiting a Sleep mode.

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 5LP register map is covered in the PSoC 5LP 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 5LP Technical Reference Manual
  - Low Power Modes chapter in the PSoC 5LP 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
- Cache Management
  - Cache Controller chapter in the PSoC 5LP Technical Reference Manual
  - Cache chapter in the System Reference Guide
    - CyFlushCache() API routine

Revision History

Version	Changes
1.0	New document

1.1	Revised document as part of the update to PSoC Creator 3.0.
	Component version updates:
	• I2C from 1.80 to 1.90
	  ○ Deprecated _SlaveSetSleepMode() and _SlaveSetWakeMode()
	• ADC_DelSig from 2.30 to 3.0
	  ○ Added _Read8/16/32() APIs
	• ADC_SAR from 2.0 to 2.10
	• Clock from 2.0 to 2.10
	• PWM from 2.40 to 3.0
	• CapSense from 3.30 to 3.40
	Replaced the slightly blurred image of device pins in Figure 1.
	Removed references to deprecated system settings; “Require XRES Pin” and “Temperature Range”.
	Added (this) Revision History section.