Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

PSoC 6 MCU: CY8C6xx8, CY8C6xxA Architecture Technical Reference Manual (TRM)

PSoC 61, PSoC 62 MCU

Document No. 002-24529 Rev. *G

December 2, 2020

Cypress Semiconductor
An Infineon Technologies Company
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com
www.infineon.com

Copyrights

Content Overview

Contents

Section A: Overview

This section encompasses the following chapters:

- Introduction chapter on page 19
- Getting Started chapter on page 25
- Document Organization and Conventions chapter on page 26

Document Revision History

Revision	Issue Date	Origin of Change	Description of Change
**	July 09, 2018	NIDH	Initial version of CY8C62x8, CY8C62xA TRM
*A	October 04, 2018	NIDH	Minor edits in Non-volatile memory Programming and USB device mode chapters, updated Figure 31-2 and I/O System. Major rewrite in Protection Units chapter. Updated figure 20-1 in Clocking System and figure 24-3 in Peripheral Trigger.
*B	October 31, 2018	AJYA	Initial version of CY8C6xx8, CY8C6xxA TRM for public release
*C	December 14, 2018	AJYA	Added section "Enabling and Disabling the FLL" on page 250 Added the "I2C Master Clock Synchronization" subsection in "Oversampling and Bit Rate" on page 361 Added a few minor updates to the Watchdog chapter Modified the number of backup domain registers to 16 in "Backup Registers" on page 241 Added section "SRAM Usage" on page 169
*D	September 30, 2019	AJYA	Modified the document title Added Boot Code chapter Deleted the Chip Operational Mode chapter Updated the Nonvolatile Memory and SAR ADC chapters Minor updates throughout the document Renamed all instances of energy profiler as "profiler" Updated Secure Digital Host Controller (SDHC), Protection Units, Clocking System, and Interrupts chapters based on review comments
*E	March 27, 2020	VKVK	Updated the Trigger Multiplexer Block, Timer, Counter, and PWM (TCPWM), I/O System, Inter-Processor Communication, DMA Controller (DW), CPU Subsystem (CPUSS), Nonvolatile Memory, and Clocking System chapters as part of the PSoC 6 collateral review effort
*F	June 30, 2020	YEKT	Added the SRAM Controller chapter. Aligned the introduction section with the datasheet. Added information about CY8C61x8 and CY8C61xA devices Updates throughout the document to address review comments.
*G	December 01, 2020	YEKT	Fixed an error in PDF bookmark.

1. Introduction

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC® MCU is a scalable and reconfigurable platform architecture that supports a family of programmable embedded system controllers with Arm® Cortex® CPUs (single and multi-core). The CY8C6xx8/CY8C6xxA product family(CY8C61x8, CY8C62x8, CY8C61xA, and CY8C62xA), based on the PSoC 6 MCU platform, is a combination of a dual-core microcontroller with built-in programmable peripherals. It incorporates integrated low-power flash technology, high-performance analog-to-digital conversion, low-power comparators, touch sensing, serial memory interface with encryption, secure digital host controller (SDHC), and standard communication and timing peripherals.

1.1 Features

32-bit Dual CPU Subsystem

- 150-MHz Arm® Cortex®-M4F (CM4) CPU with single-cycle multiply, floating point, and memory protection unit (MPU)
- 100-MHz Cortex-M0+ (CM0+) CPU with single-cycle multiply and MPU
- User-selectable core logic operation at either 1.1 V or 0.9 V
- Active CPU current slope with 1.1-V core operation
- Active CPU current slope with 0.9-V core operation
- Three DMA controllers

Memory Subsystem

- 2048-KB application flash, 32-KB auxiliary flash (AUXflash), and 32-KB supervisory flash (Sflash); read-while-write (RWW) support. Two 8-KB flash caches, one for each CPU
- 1024-KB SRAM with three independent blocks for power and data retention control
- One-time-programmable (OTP) 1-Kb eFuse array

Low-Power 1.7-V to 3.6-V Operation

- Six power modes for fine-grained power management
- Deep Sleep mode with SRAM retention
- On-chip DC-DC Buck converter
- Backup domain and real-time clock

Flexible Clocking Options

- On-chip crystal oscillators
- Two Phase-locked loops (PLL) for multiplying clock frequency

- Internal main oscillator (IMO)
- Ultra-low-power internal low-speed oscillator (ILO)
- Frequency locked loop (FLL) for multiplying IMO frequency

Quad-SPI (QSPI)/Serial Memory Interface (SMIF)

- Execute-In-Place (XIP) from external Quad SPI Flash
- On-the-fly encryption and decryption
- 4-KB cache for greater XIP performance with lower power
- Supports single, dual, quad, dual-quad, and octal interfaces

Serial Communication

- Thirteen run-time configurable serial communication blocks (SCBs)
  - Eight SCBs: configurable as SPI, I^2C, or UARTs
  - Four SCBs: configurable as I^2C or UART
  - One Deep Sleep SCB: configurable as SPI or I^2C
- USB full-speed device interface
- Two independent SD Host Controller/eMMC/SD controllers

Audio Subsystem

- Two PDM channels and one I^2S channel with TDM mode

Timing and Pulse-Width Modulation

- Thirty-two timer/counter pulse-width modulators (TCPWMs)
- Center-aligned, Edge, and Pseudo-random modes
- Comparator-based triggering of Kill signals

Programmable Analog

- 12-bit 2-Msps SAR ADC with differential and single-ended modes and 16-channel sequencer with result averaging
- Two low-power comparators available in Deep Sleep and Hibernate modes
- Two opamps
- Always-on low frequency Deep Sleep operation
- Built-in temp sensor connected to ADC

Up to 102 Programmable GPIOs

- Two Smart I/O ports (16 I/Os) enable Boolean operations on GPIO pins; available during system Deep Sleep
- Programmable drive modes, strengths, and slew rates
- Six overvoltage-tolerant (OVT) pins

LCD

- LCD segment direct block support up to 61 segments and up to 8 commons
- Operates in Active, Sleep, and Deep Sleep modes

Capacitive Sensing

- CapSense Sigma-Delta (CSD) provides best-in-class SNR, liquid tolerance, and proximity sensing
- Enables dynamic usage of both self and mutual sensing
- Automatic hardware tuning (SmartSense™)

Security Built into Platform Architecture

- ROM-based root of trust via uninterruptible Secure Boot

- Step-wise authentication of execution images
- Secure execution of code in execute-only mode for protected routines
- All Debug and Test ingress paths can be disabled
- Up to eight Protection Contexts

Cryptography Accelerators

- Hardware acceleration for symmetric and asymmetric cryptographic methods and hash functions
- True Random Number Generator (TRNG) function

Profiler

- Eight counters provide event or duration monitoring of on-chip resources

1.2 PSoC 61 and PSoC 62 MCU Series Differences

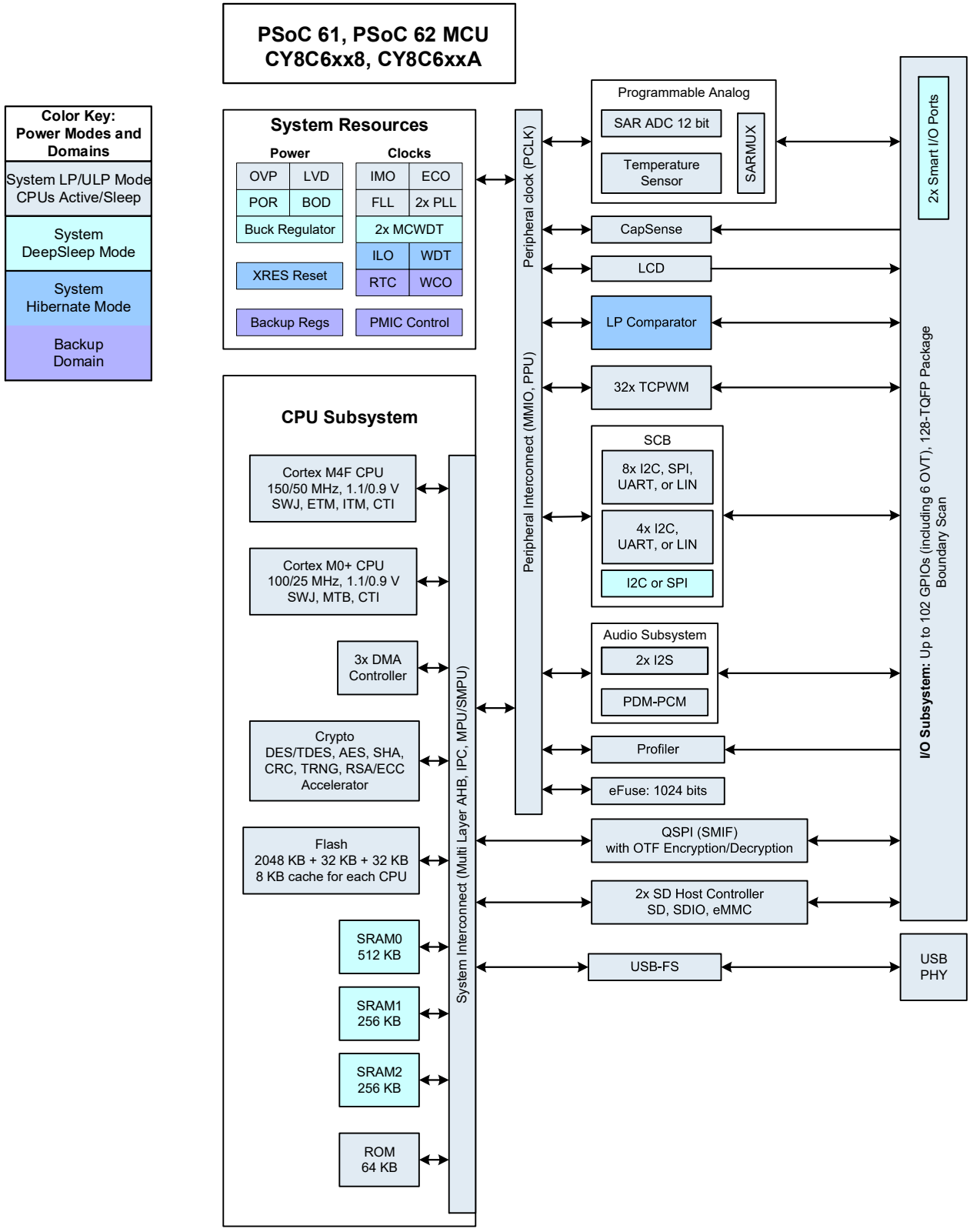
There is only one difference between the PSoC 61 (CY8C61x8, CY8C61xA) and the PSoC 62 (CY8C62x8, CY8C62xA) series MCUs.

- In the PSoC 62 series, both the CPUs (Cortex-M4F and Cortex M0+) are available for applications.
- In the PSoC 61 series, only the Cortex-M4F CPU is available for applications. The Cortex M0+ is reserved for system functions. When not executing the system functions, the CM0+ will be in CPU Deep Sleep mode.

1.3 Architecture

Figure 1-1 shows the major components of the PSoC 6 MCU architecture.

Figure 1-1. PSoC 6 MCU Architecture Block Diagram



The block diagram shows the device subsystems and gives a simplified view of their interconnections. The color-code shows the lowest power mode where the particular block is still functional (for example, LP comparator is functional in Deep Sleep and Hibernate modes).

Note: In the PSoC 61 series (CY8C61x8, CY8C61xA), only the Arm Cortex-M4F CPU is available for applications. The Cortex M0+ is reserved for system functions, and is not available for applications. When not executing the system functions, the CM0+ will be in CPU Deep Sleep mode.

2. Getting Started

2.1 PSoC 6 MCU Resources

This chapter provides the complete list of PSoC 6 MCU resources that helps you get started with the device and design your applications with them. If you are new to PSoC, Cypress provides a wealth of data at www.cypress.com to help you to select the right PSoC device and quickly and effectively integrate it into your design.

The following is an abbreviated list of PSoC 6 MCU resources:

- Overview: PSoC Portfolio, PSoC Roadmap, PSoC 6 MCU webpage
- Product Selectors: See the PSoC 6 MCU Product Selector Guide to choose a part that suits your application. In addition, ModusToolbox includes a similar device selection tool to select devices for ModusToolbox projects.
- Datasheets describe and provide electrical specifications for each device family.
- Application Notes and Code Examples cover a broad range of topics, from basic to advanced level. Many of the application notes include code examples, which can be opened from ModusToolbox.
- Technical Reference Manuals (TRMs) provide detailed descriptions of the architecture and registers in each device family.
- CapSense Design Guide: Learn how to design capacitive touch-sensing applications with PSoC devices.
- Development Tools
  - ModusToolbox is a free integrated design environment (IDE). It enables you to design hardware and firmware systems concurrently with PSoC devices.
  - PSoC 6 Kits offer an easy-to-use, inexpensive platform that enables prototyping of wide variety of designs including IoT applications requiring Wi-Fi/BT/BLE using the PSoC 6 MCU at its center.
- Additional Resources: Visit the PSoC 6 MCU webpage for additional resources such as IBIS, BSDL models, CAD Library Files, and Programming Specifications.
- Technical Support
  - Forum: See if your question is already answered by fellow developers of the PSoC 6 community.
  - Cypress support: Visit our support page or contact a local sales representative.

3. Document Organization and Conventions

This document includes the following sections:

- Section B: CPU Subsystem on page 30
- Section C: System Resources Subsystem (SRSS) on page 217
- Section D: Digital Subsystem on page 305
- Section E: Analog Subsystem on page 496

3.1 Major Sections

For ease of use, information is organized into sections and chapters that are divided according to device functionality.

- Section – Presents the top-level architecture, how to get started, and conventions and overview information of the product.
- Chapter – Presents the chapters specific to an individual aspect of the section topic. These are the detailed implementation and use information for some aspect of the integrated circuit.
- Glossary – Defines the specialized terminology used in this technical reference manual (TRM). Glossary terms are presented in bold, italic font throughout.
- Registers Technical Reference Manual – Supplies all device register details summarized in the technical reference manual. This is an additional document.

3.2 Documentation Conventions

This document uses only four distinguishing font types, besides those found in the headings.

- The first is the use of *italics* when referencing a document title or file name.
- The second is the use of ***bold italics*** when referencing a term described in the Glossary of this document.
- The third is the use of Times New Roman font, distinguishing equation examples.
- The fourth is the use of `Courier New` font, distinguishing code examples.

3.2.1 Register Conventions

Register conventions are detailed in the registers TRM.

3.2.2 Numeric Naming

Hexadecimal numbers are represented with all letters in uppercase with an appended lowercase ‘h’ (for example, ‘14h’ or 3Ah) and hexadecimal numbers may also be represented by a ‘0x’ prefix, the C coding convention. Binary numbers have an appended lowercase ‘b’ (for example, 01010100b or 01000011b’). Numbers not indicated by an ‘h’ or ‘b’ are decimal.

3.2.3 Units of Measure

This table lists the units of measure used in this document.

Table 3-1. Units of Measure

Abbreviation	Unit of Measure
bps	bits per second
°C	degrees Celsius
dB	decibels
dBm	decibels-milliwatts
fF	femtofarads
G	Giga
Hz	Hertz
k	kilo, 1000
K	kilo, 2^10
KB	1024 bytes, or approximately one thousand bytes
Kbit	1024 bits
kHz	kilohertz (32.000)
kΩ	kilohms
MHz	megahertz
MΩ	megaohms
µA	microamperes
µF	microfarads
µs	microseconds
µV	microvolts
µVrms	microvolts root-mean-square
mA	milliamperes
ms	milliseconds
mV	millivolts
nA	nanoamperes
ns	nanoseconds
nV	nanovolts
Ω	ohms
pF	picofarads
pp	peak-to-peak
ppm	parts per million
SPS	samples per second
σ	sigma: one standard deviation
V	volts

3.2.4 Acronyms and Initializations

This table lists the acronyms and initializations used in this document

Table 3-2. Acronyms and Initializations

Acronym	Definition
ABUS	analog output bus
AC	alternating current
ADC	analog-to-digital converter
ADV	advertising
AES	Advanced Encryption Standard
AHB	AMBA (advanced microcontroller bus architecture) high-performance bus, an Arm data transfer bus
API	application programming interface
APOR	analog power-on reset
BC	broadcast clock
BCD	binary coded decimal
BESL	best effort service latency
BOD	brownout detect
BOM	bill of materials
BR	bit rate
BRA	bus request acknowledge
BRQ	bus request
CAN	controller area network
CI	carry in
CIC	cascaded integrator comb
CMAC	cipher-based message authentication code
CMP	compare
CO	carry out
COM	LCD common signal
CPHA	clock phase
CPOL	clock polarity
CPU	central processing unit
CPUSS	CPU subsystem
CRC	cyclic redundancy check
CSD	CapSense sigma delta
CSX	CapSense cross-point
CT	cipher text
CTI	cross triggering interface
CTM	cross triggering matrix
ESR	equivalent series resistance
DAC	digital-to-analog converter
DAP	debug access port
DC	direct current
DES	Data Encryption Standard
DFF	D flip-flop

Table 3-2. Acronyms and Initializations *(continued)*

Acronym	Definition
DI	digital or data input
DL	drive level
DMA	direct memory access
DMIPS	Dhrystone million instructions per second
DNL	differential nonlinearity
DO	digital or data output
DSP	digital signal processing
DSM	Deep Sleep mode
DU	data unit
DW	data wire
ECO	external crystal oscillator
EEPROM	electrically erasable programmable read only memory
EMIF	external memory interface
ETM	embedded trace macrocell
FB	feedback
FIFO	first in first out
FPU	floating point unit
FSR	full scale range
GAP	generic access profile
GATT	generic attribute profile
GFSK	Gaussian frequency-shift keying
GPIO	general-purpose I/O
HCI	host-controller interface
HFCLK	high-frequency clock
HMAC	hashed message authentication code
HPF	high-pass filter
HSIOM	high-speed I/O matrix
I^2C	inter-integrated circuit
I^2S	inter-IC sound
IDE	integrated development environment
ILO	internal low-speed oscillator
ITO	indium tin oxide
IMO	internal main oscillator
INL	integral nonlinearity
I/O	input/output
IOR	I/O read
IOW	I/O write
IPC	inter-processor communication
IPTAT	proportional to absolute temperature
IRES	initial power on reset
IRA	interrupt request acknowledge
IRK	identity resolution key

Table 3-2. Acronyms and Initializations *(continued)*

Acronym	Definition
IRQ	interrupt request
ISA	instruction set architecture
ISR	interrupt service routine
ITM	instrumentation trace macrocell
IVR	interrupt vector read
IZTAT	zero dependency to absolute temperature
JWT	JSON web token
L2CAP	logical link control and adaptation protocol
LCD	liquid crystal display
LFCLK	low-frequency clock
LFSR	linear feedback shift register
LIN	local interconnect network
LJ	left justified
LL	link layer
LNA	low-noise amplifier
LP	system low-power mode
LPCOMP	Low-Power comparator
LPM	link power management
LR	link register
LRb	last received bit
LRB	last received byte
LSb	least significant bit
LSB	least significant byte
LUT	lookup table
MAC	message authentication code
MISO	master-in-slave-out
MMIO	memory mapped input/output
MOSI	master-out-slave-in
MPU	memory protection unit
MSb	most significant bit
MSB	most significant byte
MSP	main stack pointer
MTB	micro trace buffer
NI	next instant
NMI	non-maskable interrupt
NVIC	nested vectored interrupt controller
OE	output enable
OSR	over-sampling ratio
OVP	over-voltage protection
PA	power amplifier
PC	program counter
PCB	printed circuit board
PCH	program counter high

Table 3-2. Acronyms and Initializations *(continued)*

Acronym	Definition
PCL	program counter low
PD	power down
PDU	protocol data unit
PGA	programmable gain amplifier
PHY	physical layer
PLD	programmable logic device
PM	power management
PMA	PSoC memory arbiter
POR	power-on reset
PPOR	precision power-on reset
PPU	peripheral protection units
PRNG	pseudo random number generator
PRS	pseudo random sequence
PSA	Platform Security Architecture
PSoC	Programmable System-on-Chip
PSP	process stack pointer
PSR	program status register
PSRR	power supply rejection ratio
PSSDC	power system sleep duty cycle
PWM	pulse width modulator
RAM	random-access memory
RETI	return from interrupt
RF	radio frequency
RNG	random number generator
ROM	read only memory
ROT	root of trust
RPA	resolvable private address
RMS	root mean square
RW	read/write
SAR	successive approximation register
SARSEQ	SAR sequencer
SEG	LCD segment signal
SE0	single-ended zero
SC	switched capacitor
SCB	serial communication block
SHA-256	Secure Hash Algorithm
SIE	serial interface engine
SIMO	single input multiple output
SIO	special I/O
SNR	signal-to-noise ratio
SMPU	shared memory protection units
SOF	start of frame
SOI	start of instruction

Table 3-2. Acronyms and Initializations *(continued)*

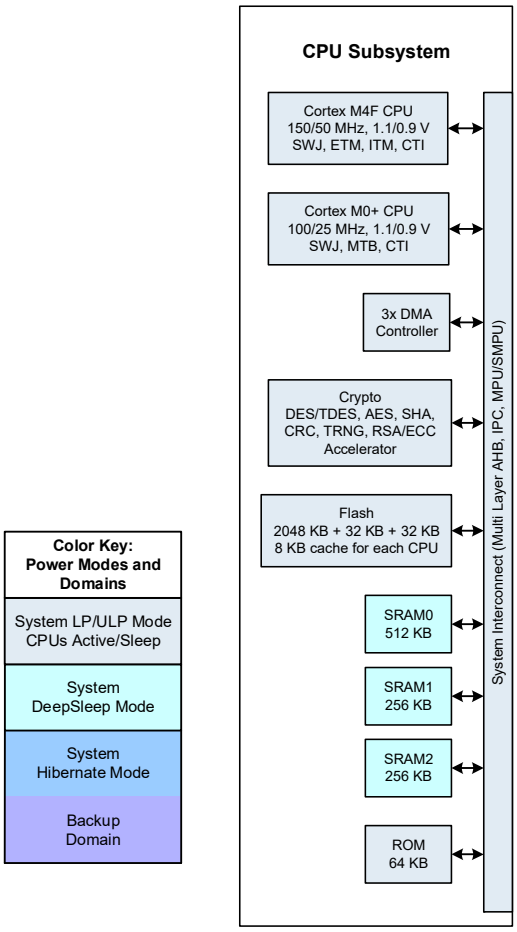
Acronym	Definition
SP	stack pointer
SPD	sequential phase detector
SPI	serial peripheral interconnect
SPIM	serial peripheral interconnect master
SPIS	serial peripheral interconnect slave
SRAM	static random-access memory
SROM	supervisory read only memory
SRSS	system resources subsystem
SSADC	single slope ADC
SSC	supervisory system call
SVCall	supervisor call
SYSCLK	system clock
SWD	single wire debug
SWV	serial wire viewer
TAR	turn-around time
TC	terminal count
TCPWM	timer, counter, PWM
TD	transaction descriptors
TDM	time division multiplexed
TFF	toggle flip-flop
TIA	trans-impedance amplifier
TPIU	trace port interface unit
TRM	technical reference manual
TRNG	True random number generator
UART	universal asynchronous receiver/transmitter
ULB	system ultra low-power mode
USB	universal serial bus
USBIO	USB I/O
VTOR	vector table offset register
WCO	watch crystal oscillator
WDT	watchdog timer
WDR	watchdog reset
WIC	wakeup interrupt controller
XRES	external reset
XRES_N	external reset, active low

Section B: CPU Subsystem

This section encompasses the following chapters:

Top Level Architecture

Figure 3-1. CPU System Block Diagram



4. CPU Subsystem (CPUSS)

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The CPU subsystem is based on dual 32-bit Arm Cortex CPUs, as Figure 4-1 shows. The Cortex-M4 is the main CPU. It is designed for short interrupt response time, high code density, and high 32-bit throughput while maintaining a strict cost and power consumption budget. A secondary Cortex-M0+ CPU implements security, safety, and protection features.

This section provides only an overview of the Arm Cortex CPUs in PSoC 6 MCUs. For details, see the Arm documentation sets for Cortex-M4 and Cortex-M0+.

Some PSoC 6 MCU parts have only one CPU. See the PSoC 61 datasheet/PSoC 62 datasheet for details.

4.1 Features

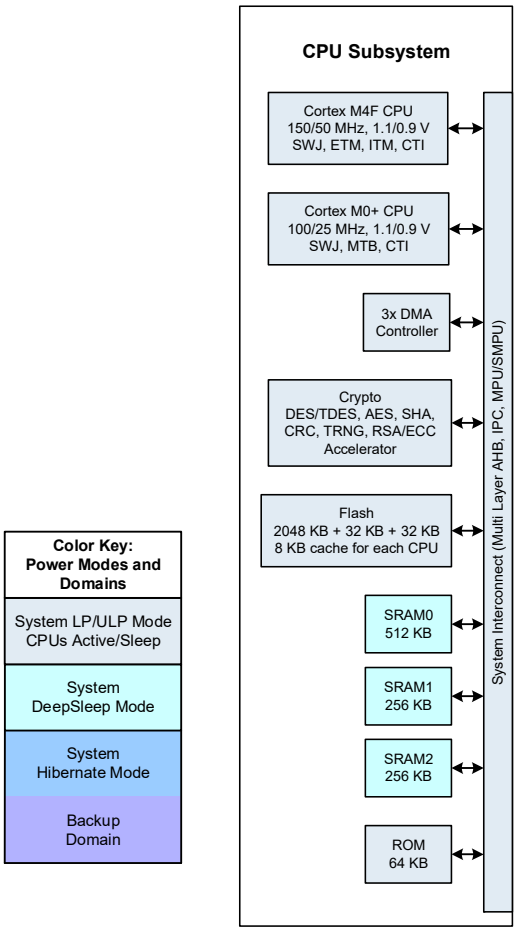
The PSoC 6 MCU Arm Cortex CPUs have the following features:

- Cortex-M4 has a floating-point unit (FPU) that supports single-cycle digital signal processing (DSP) instructions, and a memory protection unit (MPU). Cortex-M0+ has an MPU.
- Both CPUs have 8-KB instruction caches with four-way set associativity.[1]
- Maximum clock frequency of 150 MHz for the Cortex-M4 and 100 MHz for the Cortex-M0+.
- The Cortex-M4 implements a version of the Thumb instruction set based on Thumb-2 technology (defined in the *Armv7-M Architecture Reference Manual*). The Cortex-M0+ supports the Armv6-M Thumb instruction set (defined in the *Armv6-M Architecture Reference Manual*). See “Instruction Set” on page 38.
- Both CPUs have nested vectored interrupt controllers (NVIC) for rapid and deterministic interrupt response. For details, see the Interrupts chapter on page 56
- Both CPUs have extensive debug support. For details, see the Program and Debug Interface chapter on page 154.
  - SWJ: combined serial wire debug (SWD) and Joint Test Action Group (JTAG) ports
  - Serial wire viewer (SWV): provides real-time trace information through the serial wire output (SWO) interface
  - Breakpoints
  - Watchpoints
  - Trace: Cortex-M4: embedded trace macrocell (ETM). Cortex-M0+: 4-KB micro trace buffer (MTB)
- Inter-processor communication (IPC) hardware – see the Inter-Processor Communication chapter on page 41.

1. PSoC 6 does not support cache coherency. As a result when a particular row of flash that executes instructions is written/updated, the updated information will not be reflected in the cache. The cache should be cleared in the firmware during such instances. This is applicable for both CM4 and CM0+ cache – the appropriate (CM0+ and/or CM4) cache should be invalidated.

4.2 Architecture

Figure 4-1. CPU Subsystem Block Diagram



Each CPU is a 32-bit processor with its own 32-bit datapath and a 32-bit memory interface. Each CPU has its own set of 32-bit registers. They support a wide variety of instructions in the Thumb instruction set. They support two operating modes (see “Operating Modes and Privilege Levels” on page 37).

The Cortex-M4 instruction set includes:

- Signed and unsigned, 32×32 → 32-bit and 32×32 → 64-bit, multiply and multiply-accumulate, all single-cycle
- Signed and unsigned 32-bit divides that take two to 12 cycles
- DSP instructions, including single instruction multiple data (SIMD) instructions
- Complex memory-load and store access
- Complex bit manipulation; see the bitfield instructions in Table 4-6

The Cortex-M4 FPU has its own set of registers and instructions. It is compliant with the ANSI/IEEE Std 754-2008, IEEE Standard for Binary Floating-Point Arithmetic.

The Cortex-M0+ has a single cycle 32x32 → 32-bit signed multiplication instruction.

4.2.1 Address and Memory Maps

Both CPUs have a fixed address map, with shared access to memory and peripherals. The 32-bit (4 GB) address space is divided into the regions shown in Table 4-1. Note that code can be executed from the code and SRAM regions.

Table 4-1. Address Map for Cortex-M4 and Cortex-M0+

Address Range	Region Name	Use
0x0000 0000 – 0x1FFF FFFF	Code	Program code region. You can also put data here. It includes the exception vector table, which starts at address 0.
0x2000 0000 – 0x3FFF FFFF	SRAM	Data region. This region is not supported in PSoC 6.
0x4000 0000 – 0x5FFF FFFF	Peripheral	All peripheral registers. Code cannot be executed from this region. Note that the Cortex-M4 bit-band in this region is not supported in PSoC 6.
0x6000 0000 – 0x9FFF FFFF	External RAM	Not used
0xA000 0000 – 0xDFFF FFFF	External Device	Not used
0xE000 0000 – 0xE00F FFFF	Private Peripheral Bus (PPB)	Provides access to peripheral registers within the CPU core.
0xE010 0000 – 0xFFFF FFFF	Device	Device-specific system registers.

The device memory map shown in Table 4-2 applies to both CPUs. That is, the CPUs share access to all PSoC 6 MCU memory and peripheral registers.

Table 4-2. PSoC 6 Memory Map

Address Range	Name	Comments
0x0000 0000 –0x0001 0000	SROM	64 Kbytes
0x0800 0000 – 0x0810 0000	SRAM	Up to 1 Mbyte
0x1000 0000 – 0x1020 0000	User Application Flash	Up to 2 Mbytes
0x1600 0000 – 0x1600 8000	Supervisory Flash (SFlash)	32K for secure access
0x1800 0000 – 0x0800 0000	External memory	128 Mbyte execute-in-place (XIP) region

SRAM is located in the code region for both CPUs (see Table 4-1). This facilitates executing code out of SRAM. There is no physical memory located in the CPUs' SRAM region.

Note: The CPUSS_CM0_VECTOR_TABLE_BASE and CPUSS_CM4_VECTOR_TABLE_BASE registers determine the location of the vector table for each CPU. A number of LS bits in each register are set to 0. As a result, there are restrictions on the location of vector tables – they must be on a 256-byte boundary for CM0+ and a 1024-byte boundary for CM4.

4.2.1.1 Wait State Lookup Tables

The wait state lookup tables show the wait states for Flash, SRAM, and ROM based on the Clk_HF0 frequency and the current power mode. SRAM and ROM have two domains for the wait states – fast clock domain (Clk_Fast) and slow clock domain (Clk_Slow); both domains are based off Clk_HF0. The following tables show the wait states for the slow clock domain. All wait states for the fast clock domain are zero. For more information on clocking see the Clocking System chapter on page 242.

	Ultra-Low Power Mode	Clk_HF0 (MHz)		
		Clk_HF0 ≤ 25	25 < Clk_HF0 ≤ 100	100 < Clk_HF0
ROM/SRAM	True	0	1	1
	False	0	0	1

	Ultra-Low Power Mode	Clk_HF0 (MHz)				
Flash	True	Clk_HF0 ≤ 16		16 < Clk_HF0 ≤ 33		33 < Clk_HF0
		0		1		2
	False	Clk_HF0 ≤ 29	29 < Clk_HF0 ≤ 58	58 < Clk_HF0 ≤ 87	87 < Clk_HF0 ≤ 120	120 < Clk_HF0 ≤ 150
		0	1	2	3	4

4.3 Registers

Both CPUs have sixteen 32-bit registers, as Table 4-3 shows. See the Arm documentation for details.

- R0 to R12 – General-purpose registers. R0 to R7 can be accessed by all instructions; the other registers can be accessed by a subset of the instructions.
- R13 – Stack pointer (SP). There are two stack pointers, with only one available at a time. In thread mode, the CONTROL register indicates the stack pointer to use – Main Stack Pointer (MSP) or Process Stack Pointer (PSP).
- R14 – Link register. Stores the return program counter during function calls.
- R15 – Program counter. This register can be written to control program flow.

Table 4-3. Cortex-M4 and Cortex-M0+ Registers

Name	Type[a]	Reset Value	Description
R0 – R12	RW	Undefined	R0–R12 are 32-bit general-purpose registers for data operations.
MSP (R13) PSP (R13)	RW	[0x0000 0000]	The stack pointer (SP) is register R13. In thread mode, bit[1] of the CONTROL register indicates the stack pointer to use: 0 = Main stack pointer (MSP). This is the reset value. 1 = Process stack pointer (PSP). On reset, the processor loads the MSP with the value from the vector address.
LR (R14)	RW	See note[b]	The link register (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions.
PC (R15)	RW	[0x0000 0004]	The program counter (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value from the vector address plus 0x0000 0004. Bit[0] of the value is loaded into the EPSR T-bit (see Table 4-4) at reset; it must always be 1.
PSR	RW	Undefined	The program status register (PSR) combines: Application Program Status Register (APSR). Execution Program Status Register (EPSR). Interrupt Program Status Register (IPSR).
APSR	RW	Undefined	The APSR contains the current state of the condition flags from previous instruction executions.
EPSR	RO	0x0100 0000	On reset, the EPSR Thumb state bit is loaded with the value bit[0] of the register [0x0000 0004]. It must always be 1. In Cortex-M4, other bits in this register control the state of interrupt-continuable instructions and the if-then (IT) instruction.
IPSR	RO	0	The IPSR contains the current exception number.
PRIMASK	RW	0	The PRIMASK register prevents activation of all exceptions with configurable priority.
CONTROL	RW	0	The CONTROL register controls: - The privilege level in Thread mode; see 4.4 Operating Modes and Privilege Levels. - The currently active stack pointer, MSP or PSP. - Cortex-M4 only: whether to preserve the floating-point state when processing an exception.
FAULTMASK	RW	0	Cortex-M4 only. Bit 0 = 1 prevents the activation of all exceptions except NMI.
BASEPRI	RW	0	Cortex-M4 only. When set to a nonzero value, prevents processing any exception with a priority greater than or equal to the value.

a. Describes access type during program execution in thread mode and handler mode. Debug access can differ.
b. LR reset value is 0xFFFF FFFF in Cortex-M4, undefined in Cortex-M0+.

The Cortex-M4 floating-point unit (FPU) also has the following registers:

- Thirty-two 32-bit single-precision registers, S0 to S31. These registers can also be addressed as sixteen 64-bit double-precision registers, D0 to D15.
- Five FPU control and status registers:
  - CPACR – Coprocessor Access Control Register
  - FPCCR – Floating-point Context Control Register
  - FPCAR – Floating-point Context Address Register
  - FPSCR – Floating-point Status Control Register
  - FPDSCR – Floating-point Default Status Control Register

For more information on how these registers are used, see the Arm Cortex-M4 documentation.

Use the MSR and MRS instructions to access the PSR, PRIMASK, CONTROL, FAULTMASK, and BASEPRI registers. Table 4-4 and Table 4-5 show how the PSR bits are assigned.

Table 4-4. Cortex-M4 PSR Bit Assignments

Bit	PSR Register	Name	Usage
31	APSR	N	Negative flag
30	APSR	Z	Zero flag
29	APSR	C	Carry or borrow flag
28	APSR	V	Overflow flag
27	APSR	Q	DSP overflow and saturation flag
26 – 25	EPSR	IC/IT	Control interrupt-continuable and IT instructions
24	EPSR	T	Thumb state bit. Must always be 1. Attempting to execute instructions when the T bit is 0 results in a HardFault exception.
23 – 20	–	–	Reserved
19 – 16	APSR	GE	Greater than or equal flags, for the SEL instruction
15 – 10	EPSR	IC/TI	Control interrupt-continuable and IT instructions
9	–	–	Reserved
8 – 0	IPSR	ISR_NUMBER	Exception number of current ISR: 0 = thread mode 1 = reserved 2 = NMI 3 = HardFault 4 = MemManage 5 = BusFault 6 = UsageFault 7 – 10 = reserved 11 = SVCall 12 = reserved for debug 13 = reserved 14 = PendSV 15 = SysTick (see “System Tick (SysTick) Exception” on page 62) 16 = IRQ0 … 255 = IRQ240

Table 4-5. Cortex-M0+ PSR Bit Assignments

Bit	PSR Register	Name	Usage
31	APSR	N	Negative flag
30	APSR	Z	Zero flag
29	APSR	C	Carry or borrow flag
28	APSR	V	Overflow flag
27 – 25	–	–	Reserved
24	EPSR	T	Thumb state bit. Must always be 1. Attempting to execute instructions when the T bit is 0 results in a HardFault exception.
23 – 6	–	–	Reserved
5 – 0	IPSR	Exception Number	Exception number of current ISR: 0 = thread mode 1 = reserved 2 = NMI 3 = HardFault 4 – 10 = reserved 11 = SVCall 12, 13 = reserved 14 = PendSV 15 = SysTick 16 = IRQ0 … 47 = IRQ31

4.4 Operating Modes and Privilege Levels

Both CPUs support two operating modes and two privilege levels:

- Operating Modes:
  - Thread Mode – used to execute application software. The processor enters Thread mode when it comes out of reset.
  - Handler Mode – used to handle exceptions. The processor returns to Thread mode when it has finished all exception processing.
- Privilege Levels:
  - Unprivileged – the software has limited access to the MSR and MRS instructions, and cannot use the CPSID and CPSIE instructions. It cannot access the system timer, NVIC, or system control block. It may have restricted access to memory or peripherals.
  - Privileged – the software can use all the instructions and has access to all resources.

In Thread mode, the CONTROL register controls whether software execution is privileged or unprivileged. In Handler mode, software execution is always privileged.

Only privileged software can write to the CONTROL register to change the privilege level. Unprivileged software can use the SVC instruction to transfer control to privileged software.

In Handler mode, the MSP is always used. The exception entry and return mechanisms automatically update the CONTROL register, which may change whether MSP/PSP is used.

In Thread mode, use the MSR instruction to set the stack pointer bit in the CONTROL register. When changing the stack pointer, use an ISB instruction immediately after the MSR instruction. This ensures that instructions after the ISB execute using the new stack pointer.

4.5 Instruction Set

Both CPUs implement subsets of the Thumb instruction set, as Table 4-6 shows. The table does not show the large number of variants and conditions of the instructions. For details, see one of the Arm Cortex Generic User Guides or Technical Reference Manuals.

An instruction operand can be a register, a constant, or another instruction-specific parameter. Instructions act on the operands and often store the result in a destination register. Many instructions have restrictions on using the PC or SP for the operands or destination register. See the Arm documentation for details.

Table 4-6. Instruction Set Summary – Cortex-M4 and Cortex-M0+

Functional Group	Cortex-M4	Cortex-M0+	Brief List of Instruction Mnemonics
Memory access	✔	✔	LDR, STR, ADR, PUSH, POP
General data processing	✔	✔	Cortex-M0+: ADD, ADC, AND, ASR, BICS, CMN, CMP, EOR, LSL, LSR, MOV, MVNS, ORR, REV, ROR, RSB, SBC, SUB, SXT, UXT, TST Cortex-M4 has all of the above plus: CLZ, ORN, RRX, SADD, SAS, SSA, SSUB, TEQ, UADD, UAS, USA, USUB
Multiply and divide	✔	MUL only	MLA, MLS, MUL, SDIV, SMLA, SMLS, SMMLA, SMMLS, SMUA, SMUL, SMUS, UDIV, UMAAL, UMLAL, UMULL
Saturating	✔	–	SSAT, USAT, QADD, QSUB, QASX, QSAX, QDADD, QDSUB, UQADD, UQASX, UQSAX, UQSUB
Packing and unpacking	✔	–	PKH, SXT, SXTA, UXT, UXTA
Bitfield	✔	–	BFC, BFI, SBFX, UBFX
Branch and control	✔	✔	Cortex-M0+: B{cc}, BL, BLX, BX Cortex-M4 has all of the above plus: CBNZ, CBZ, IT, TB
Miscellaneous	✔	✔	CPSID, CPSIE, DMB, DSB, ISB, MRS, MSR, NOP, SEV, SVC, WFE, WFI Cortex-M4 has all of the above plus BKPT
Floating-point	✔	–	VABS, VADD, VCMP, VCVT, VDIV, VFMA, VFNMA, VFMS, VFNMS, VLD, VLMA, VLMS, VMOV, VMRS, VMSR, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VPOP, VPUSH, VSQRT, VST, VSUB

5. SRAM Controller

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

This chapter explains the PSoC 6 MCU SRAM Controller, its features, architecture, and wait states. The SRAM controller enables the CPU to read and write parts of the PSoC 6 SRAM.

5.1 Features

The CPUSS has up to three identical SRAM controllers; see the PSoC 61 datasheet/PSoC 62 datasheet for details.

The SRAM controller has the following features:

- Consists of two AHB-Lite interfaces:
  - An AHB-Lite bus interface on clk_fast that connects to the fast bus infrastructure
  - An AHB-Lite bus interface on clk_slow that connects to the slow bus infrastructure
- Supports programmable number of clk_hf wait states
- Supports 8-, 16-, and 32-bit accesses

5.2 Architecture

The design has two AHB-Lite interfaces that connect to the AHB-Lite infrastructure. Each AHB-Lite interface is connected to a synchronization component that translates between the interface clock (either clk_fast or clk_slow) and the high-frequency clock (clk_hf).

Arbitration is performed on the AHB-Lite transfers from the two ports (AHB-Lite interface). Arbitration uses device-wide bus master specific arbitration priorities. Therefore, although two AHB-Lite interfaces are provided, only one AHB-Lite transfer is accepted by the port arbitration component.

The AHB-Lite transfers are the origin for all SRAM accesses; that is, the write buffer and SRAM repair requests result from AHB-Lite transfers. The SRAM controller differentiates between the following three types of AHB-Lite transfers:

- AHB-Lite read transfers
- 32-bit AHB-Lite write transfers
- 8-bit and 16-bit AHB-Lite write transfers (also referred to as partial AHB-Lite write transfers)

Each type is described in more detail here.

AHB-Lite read transfers. An AHB-Lite read transfer is translated into an SRAM read access. If the read address matches in the write buffer, the SRAM has stale data and the write data provides the requested read data (this functionality is provided by the read merge component).

32-bit AHB-Lite write transfers. A 32-bit AHB-Lite write transfer is translated into an SRAM write access. If the write address matches in the write buffer, the matching write buffer entries have stale data and these entries are invalidated.

Partial AHB-Lite write transfers. A partial AHB-Lite write transfer is translated into an SRAM read access and an SRAM write access. The SRAM read access is the direct result of the partial write transfer and the SRAM write access is the result of a write buffer request. A partial write transfer requires an SRAM read access to retrieve the “missing” data bytes from the SRAM. If the read address matches in the write buffer, the SRAM has stale data and the write data provides the requested read data (this functionality is provided by the read merge component). The requested read data is merged with the partial write data to provide a complete 32-bit data word (this functionality is provided by the write merge component). The address and the merged write data are written to the write buffer. A future write buffer request results in an SRAM write access with the merged write data.

Only the partial AHB-Lite write transfers use the write buffer.

Write buffer. The write buffer is a temporary holding station for future SRAM write accesses.

The buffer allows SRAM write accesses to be postponed. This allows for more performance critical AHB-Lite requests to “overtake” write buffer requests. Memory consistency is guaranteed by matching the SRAM access address with the write buffer entries' addresses: a “matching” SRAM read access uses the read merge component and a matching SRAM write access invalidates the matching write buffer entries.

When the write buffer is full, an entry needs to be freed to accommodate future partial AHB-Lite write transfers. Therefore, a full write buffer raises the priority of the write buffer request path.

The write buffer is constructed as a FIFO with four entries (the order in which entries are written is the same as the order in which entries are read). Each entry consists of:

- A valid field
- An invalidated field
- A word address
- A 32-bit data word

  Note that the merged write data written to the write buffer is always a 32-bit data word. Therefore, no byte mask is required.

When the write buffer is written (an entry is added): the entry valid field is set to ‘1’ and the invalidated field is set to ‘0’.

When the write buffer is read (an entry is removed): the entry valid field is set to ‘0’. If the entry invalidate field is ‘1’, the write buffer request path is selected for an SRAM write access. If the entry valid field is ‘0’, no SRAM access is performed.

On an SRAM read access, a matching entry provides write buffer merge data for the read merge component.

On an SRAM write access resulting from a 32-bit AHB-Lite write transfer, a matching entry invalidated field is set to ‘1’.

The state of the write buffer is reflected by RAMi_STATUS.WB_EMPTY. The write buffer is not retained in Deep Sleep power mode. Therefore, when transitioning to system Deep Sleep power mode, the write buffer should be empty. Note that this requirement is typically met, because a transition to Deep Sleep power mode also requires that there are no outstanding AHB-Lite transfers. If there are no outstanding AHB-Lite transfers, the write buffer gets SRAM access.

5.3 Wait States

The programmable wait states represent the number of clk_hf cycles for a read path through the SRAM memory to flipflops in either the fast domain (CM4 CPU) or slow domain (such as CM0+ CPU, DataWire, and DMA controller).

As the wait states are represented in clk_hf cycles, the wait states do not have to be reprogrammed when the fast clock domain frequency (clk_fast) or slow clock domain frequency (clk_slow) is changed. However, it may be necessary to reprogram the wait states when the high-frequency clock domain (clk_hf) is changed. This means the required number of wait states is a function of the clk_hf frequency.

The fast clock domain is timing closed at a higher frequency than the slow clock domain. Therefore, the read path through the SRAM memory to flipflops in the fast domain is faster than the read path through the SRAM memory to flipflops in the slow domain. In other words, the required number of “fast” wait states (RAMi_CTL.FAST_WS) should be less than or equal to the required number of “slow” wait states (RAMi_CTL.SLOW_WS).

The SRAM controller also has internal SRAM read paths. These paths are to flipflops in the SRAM controller in the high-frequency clock domain (clk_hf). For these SRAM accesses (for example, an SRAM read access to support a partial AHB-Lite write transfer), the fast wait states are used. This is because the maximum fast domain frequency (clk_fast) equals the high-frequency domain frequency (clk_hf).

6. Inter-Processor Communication

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

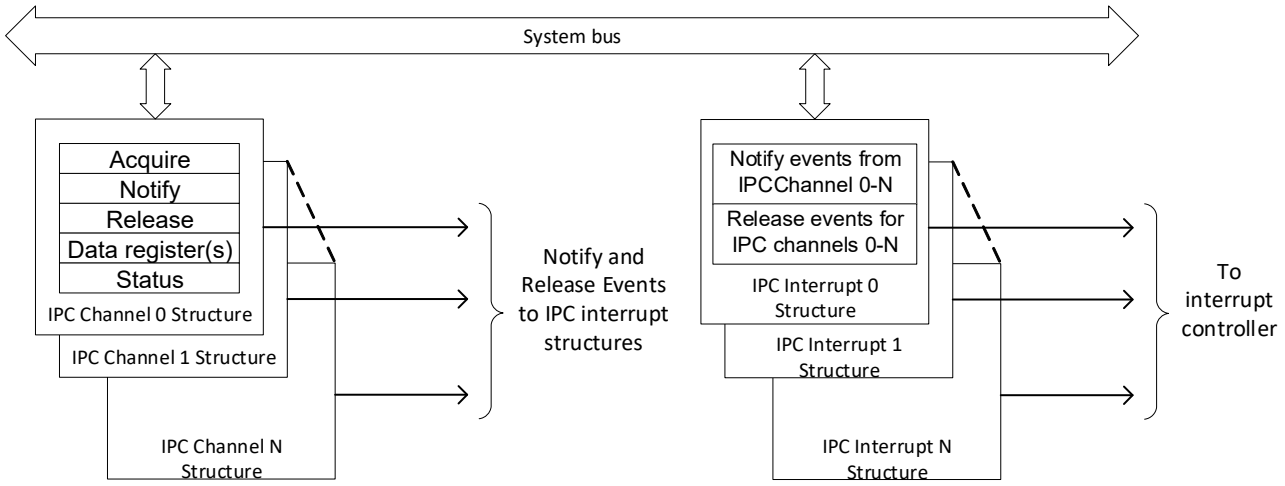
- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

Inter-processor communication (IPC) provides the functionality for multiple processors to communicate and synchronize their activities. IPC hardware is implemented using two register structures.

- IPC Channel: Communication and synchronization between processors is achieved using this structure.
- IPC Interrupt: Each interrupt structure configures an interrupt line, which can be triggered by a 'notify' or 'release' event of any IPC channel.

The Channel and Interrupt structures are independent and have no correlation to each other as shown in Figure 6-1. This allows for building varying models of interface shown in Typical Usage Models on page 46.

Figure 6-1. IPC Register Architecture



6.1 Features

The features of IPC are as follows:

- Implements locks for mutual exclusion between processors
- Allows sending messages between processors
- Supports up to 16 channels for communication
- Supports up to 16 interrupts, which can be triggered using notify or release events from the channels

6.2 Architecture

6.2.1 IPC Channel

An IPC channel is implemented as six hardware registers, as shown in Figure 6-2. The IPC channel registers are accessible to all the processors in the system.

- IPC_STRUCTx_ACQUIRE: This register determines the lock feature of the IPC. The IPC channel is acquired by reading this register. If the SUCCESS field returns a '1', the read acquired the lock.

  If the SUCCESS field returns a '0', the read did not acquire the lock.

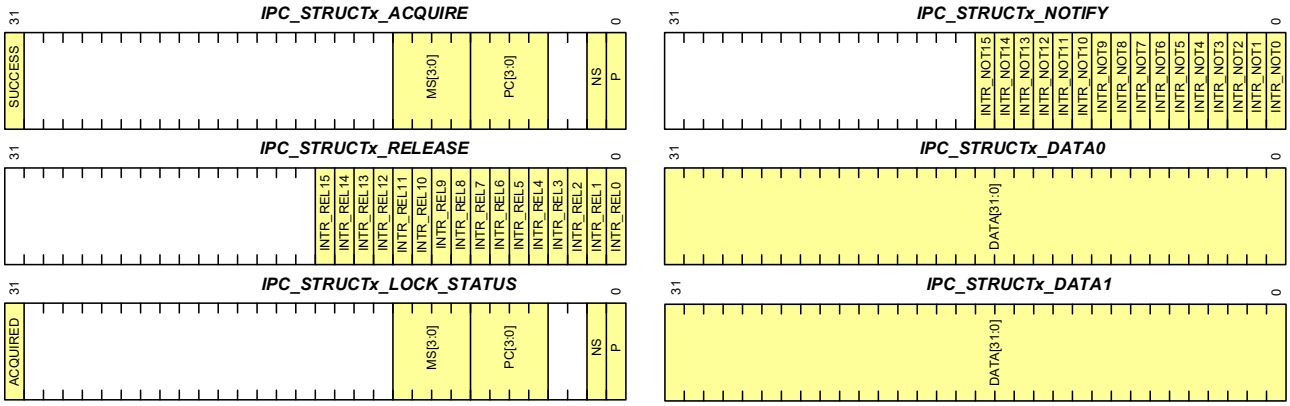
  Note that a single read access performs two functions:

  - The attempt to acquire a lock.
  - Return the result of the acquisition attempt (SUCCESS field).

  The atomicity of these two functions is essential in a CPU with multiple tasks that can preempt each other.

  The register also has bitfields that provide information about the processor that acquired it. When acquired, this register is released by writing any value into the IPC_STRUCTx_RELEASE register. If the register was already in an acquired state another attempt to read the register will not be able to acquire it.
- IPC_STRUCTx_NOTIFY: This register is used to generate an IPC notify event. Each bit in this register corresponds to an IPC interrupt structure. The notify event generated from an IPC channel can trigger any or multiple interrupt structures.
- IPC_STRUCTx_RELEASE: Any write to this register will release the IPC channel. This register also has a bit that corresponds to each IPC interrupt structure. The release event generated from an IPC channel can trigger any or multiple interrupt structures. To only release the IPC channel and not generate an interrupt, you can write a zero into the IPC release register.
- IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1: These are two 32-bit registers with a combined size of 64 bits that are meant to hold data. These registers can be considered as the shared data memory for the channel. Typically, these registers will hold messages that need to be communicated between processors. If the messages are larger than the combined 64-bit size, place pointers in one or both of these registers.
- IPC_STRUCTx_LOCK_STATUS: This register provides the instantaneous lock status for the IPC channel. The register provides details if the channel is acquired. If acquired, it provides the processor's ID, protection context, and other details. The reading of lock status provides only an instantaneous status, which can be changed in the next cycle based on the activity of other processors on the channel.

Figure 6-2. IPC Channel Structure



6.2.2 IPC Interrupt

Each IPC interrupt line in the system has a corresponding IPC interrupt structure. An IPC interrupt can be triggered by a notify or a release event from any of the IPC channels in the system. You can choose to mask any of the sources of these events using the IPC interrupt registers. Figure 6-3 shows the registers in an IPC Interrupt structure.

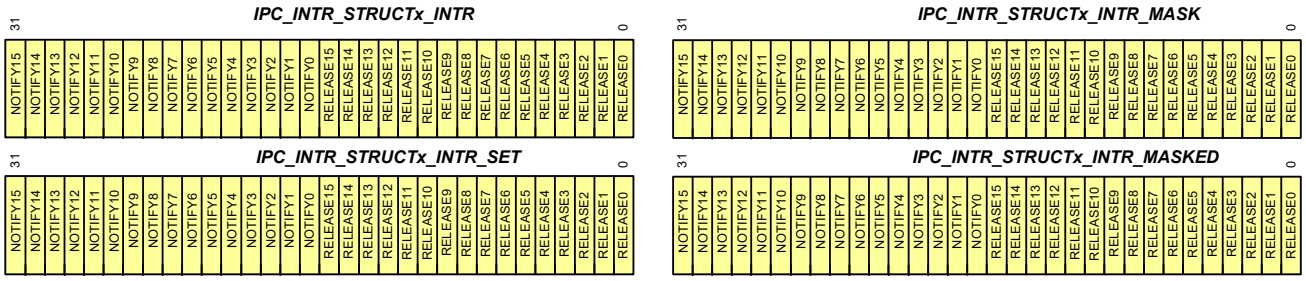
IPC_INTR_STRUCTx_INTR: This register provides the instantaneous status of the interrupt sources. Note that there are 16 notify and 16 release event bits in this register. These are the notify and release events corresponding to the 16 IPC channels. When a notify event is triggered in the IPC channel 0, the corresponding Notify0 bit is activated in the interrupt registers. A write of '1' to a bit will clear the interrupt.

IPC_INTR_STRUCTx_INTR_MASK: The bit in this register masks the interrupt sources. Only the interrupt sources with their masks enabled can trigger the interrupt.

IPC_INTR_STRUCTx_INTR_SET: A write of '1' into this register will set the interrupt.

IPC_INTR_STRUCTx_INTR_MASKED: This register provides the instantaneous value of the interrupts after they are masked. The value in this register is (IPC_INTR_STRUCTx_INTR AND IPC_INTR_STRUCTx_INTR_MASK).

Figure 6-3. IPC Interrupt Structure

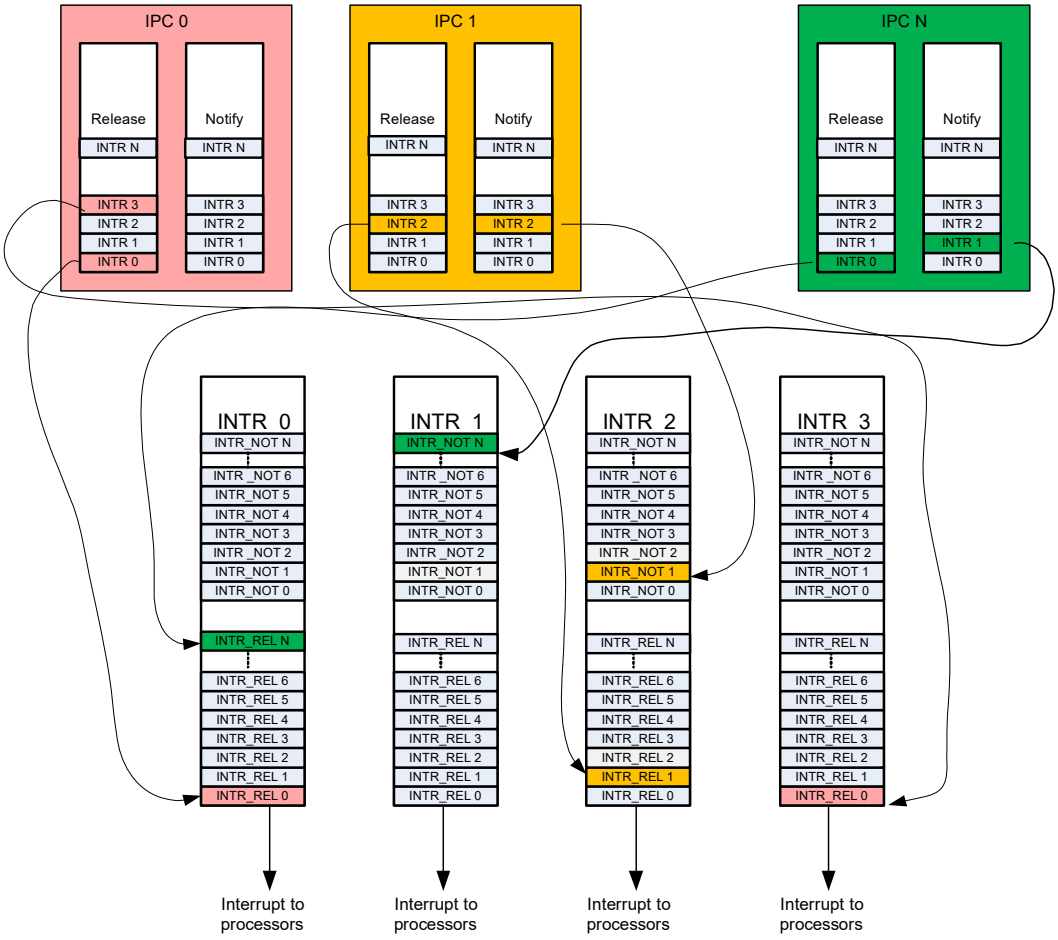


6.2.3 IPC Channels and Interrupts

The IPC block has a set of IPC interrupts associated with it. Each IPC interrupt register structure corresponds to an IPC interrupt line. This interrupt can trigger an interrupt on any of the processors in the system. The interrupt routing for processors are dependent on the device architecture.

Each IPC channel has a release and notify register, which can drive events on any of the IPC interrupts. An illustration of this relation between the IPC channels and the IPC interrupt structure is shown in Figure 6-4.

Figure 6-4. IPC Channels and Interrupts



6.3 Implementing Locks

The IPC channels can be used to implement locks. Locks are typically used in multi-core systems to implement some form of mutually exclusive access to a shared resource. When multiple processors share a resource, the processors are capable of acquiring and releasing the IPC channel. So the processor can assume an IPC channel as a lock. The semantics of this code is that access to the shared resource is gated by the processor's ownership of the channel. So the processors will need to acquire the IPC channel before they access the shared resource.

A failure to acquire the IPC channel signifies a lock on the shared resource because another processor has control of it. Note that the IPC channel will not enforce which processor acquires or releases the channel. All processors can acquire or release the IPC channel and the semantics of the code must make sure that the processor that acquires the channel is the one that releases it.

6.4 Message Passing

IPC channels can be used to communicate messages between processors. In this use case, the channel is used in conjunction with the interrupt structures. The IPC channel is used to lock the access to the data registers. The IPC channel is acquired by the sender and used to populate the message. The receiver reads the message and then releases the channel. Thus, between the sender putting data into the channel and receiver reading it, the channel is locked for all other task access. The sender uses a notify event on the receiver's IPC interrupt to denote a send operation. The receiver acts on this interrupt and reads the data from the data registers. After the reception is complete, the receiver releases the channel and can also generate a release event to the senders IPC interrupt. Note that the action of locking the channel does not, in hardware, restrict access to the data registers. This is a semantic that should be enforced by software.

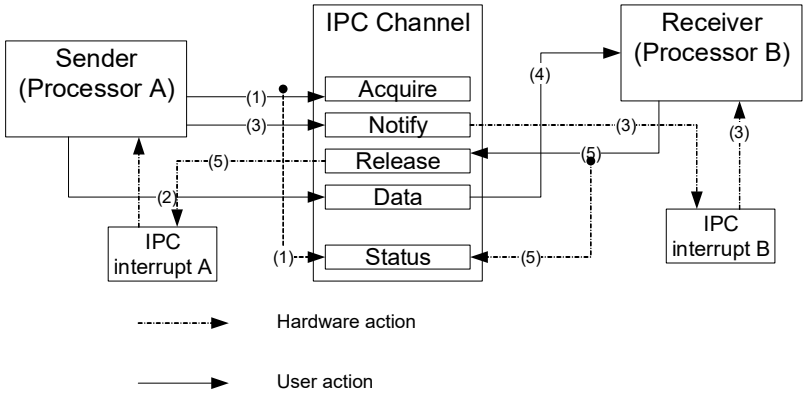
Figure 6-5 portrays an example of a sender (Processor A) sending data to a receiver (Processor B). IPC interrupt A is

configured to interrupt Processor A. IPC interrupt B is configured to interrupt Processor B.

1. The sender will attempt to acquire the IPC channel by reading the IPC_STRUCTx_ACQUIRE register. If the channel was acquired, the sender has ownership of the channel for data transmission. This also changes the status of the channel and its corresponding IPC_STRUCTx_LOCK_STATUS register. If the channel was not acquired, the processor should wait until the channel is free for acquisition. This can be done by polling the IPC channel's IPC_STRUCTx_LOCK_STATUS register.
2. After the IPC channel is acquired, the sender has control of the channel for communication and places the 64-bit message data in the IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1 registers.
3. Now that the message is placed in the IPC channel, the sender generates a notify event on the receiver's interrupt line. It does this by setting the corresponding bit in the IPC channel's IPC_STRUCTx_NOTIFY register. This event creates a notify event at IPC interrupt B (IPC_INTR_STRUCTx_INTR). If the IPC channel's notify event was enabled by setting the mask bit (IPC_INTR_STRUCTx_INTR_MASK [31:23]) in the IPC interrupt B, this will generate an interrupt in the receiver.
4. When it receives IPC interrupt B, the receiver can poll the IPC_INTR_STRUCTx_INTR_MASKED register to understand which IPC channel had triggered the notify event. Based on this, the receiver identifies the channel to read and reads from the IPC channel's IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1 registers. The receiver has now received the data sent by the sender. It needs to release the channel so that other processors/processes can use it.
5. The receiver releases the channel. It also optionally generates a release event on the sender's IPC interrupt A. This will generate a release event interrupt on the sender if the corresponding channel release event was masked.

On receiving the release interrupt, the sender can act on the event based on the application requirement. It can either try to reacquire the channel for further transmission or go on to other tasks because the transmission is complete.

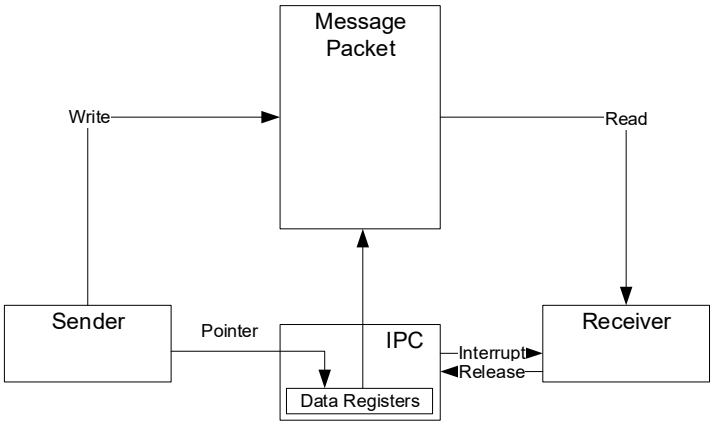
Figure 6-5. Sending Messages using IPC



In the previous example, the size of the data being transmitted was just 64 bits. Larger messages can be sent as pointers. The sender can allocate a larger message structure in memory and pass the pointers in the data registers. Figure 6-6 shows the usage. Note that the user code must implement the synchronization of the message read process.

- The implementation can stall the channel until the receiver has used up all the data in the message packet and the message packet can be rewritten. This is wasteful because it will stall other inter-process communications as the number of IPC channels is limited.
- The receiver can release the channel as soon as it receives the pointer to the message packet. It implements the synchronization logic in the message packet as a flag, which the sender sets on write complete and receiver clears on a read complete.

Figure 6-6. Communicating Larger Messages



6.5 Typical Usage Models

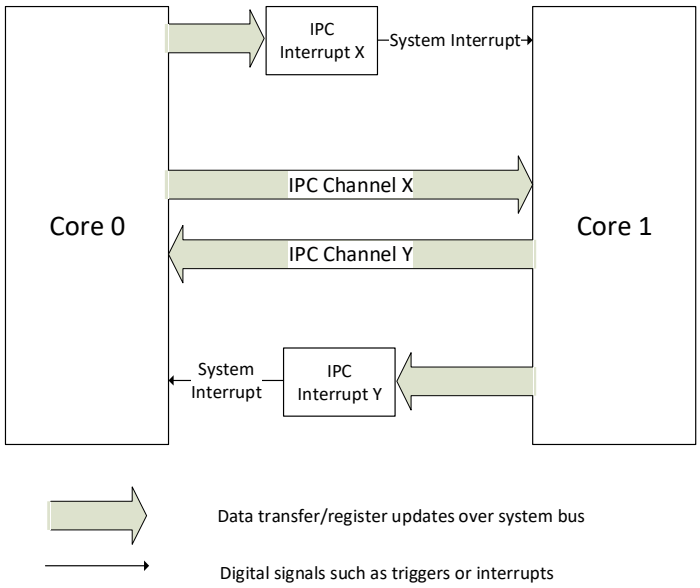
The unique channel and interrupt architecture of the PSoC 6 IPC allows for a range of usage models for multicore communication. Some of these are listed here as an example. Note that the communication models possible based on the IPC architecture are not restricted to the ones listed in this document. Also note that, this document only provides a high-level usage model and does not go into details of data management in the communication. This will need to be determined based on the specific application use case.

6.5.1 Full Duplex Communication

In this usage model, an IPC channel is used according to the direction of communication between cores. For managing events an IPC interrupt is used per core. In a dual core system this will translate to what is shown in Figure 6-7.

In this example, the IPC channel X is dedicated to data communication from Core 0 to Core 1 and IPC channel Y is for data communication from Core 1 to core 0. The IPC interrupt X will signal events on Core 1. Hence its interrupt output is connected to Core 1's system interrupts. The events are triggered by writing into the IPC interrupts register structure over the system bus. Similarly, IPC interrupt Y is dedicated to Core 0.

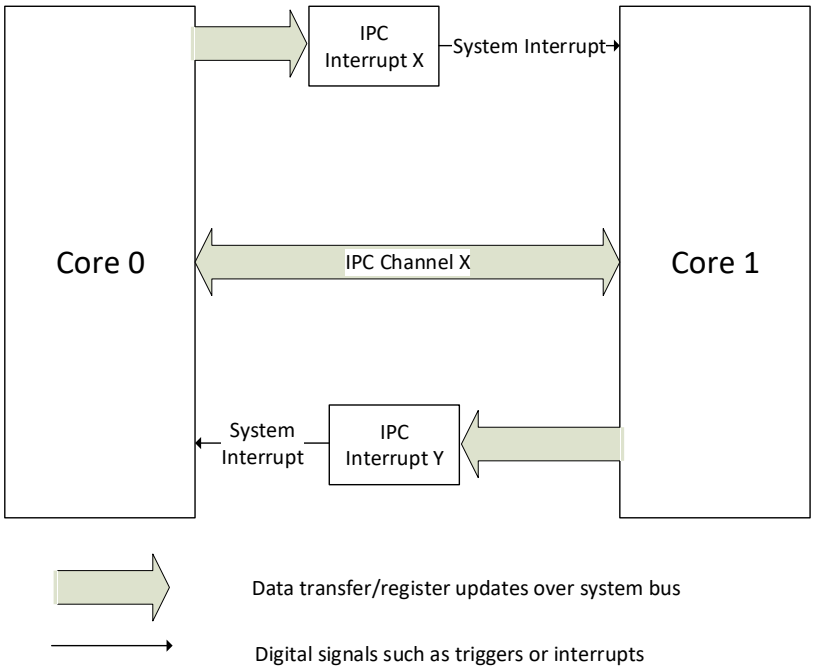
Figure 6-7. Full Duplex IPC for Dual Core Communication



6.5.2 Half Duplex with Independent Event Handling

In this case only one IPC channel is used to set up the transfer between the two cores. This means that only one side controls data transfer at a time. The channels Lock register must be used to avoid contention of the one shared IPC channel. Two independent events are supported due to the two IPC interrupt structures being used. This model is shown in Figure 6-8.

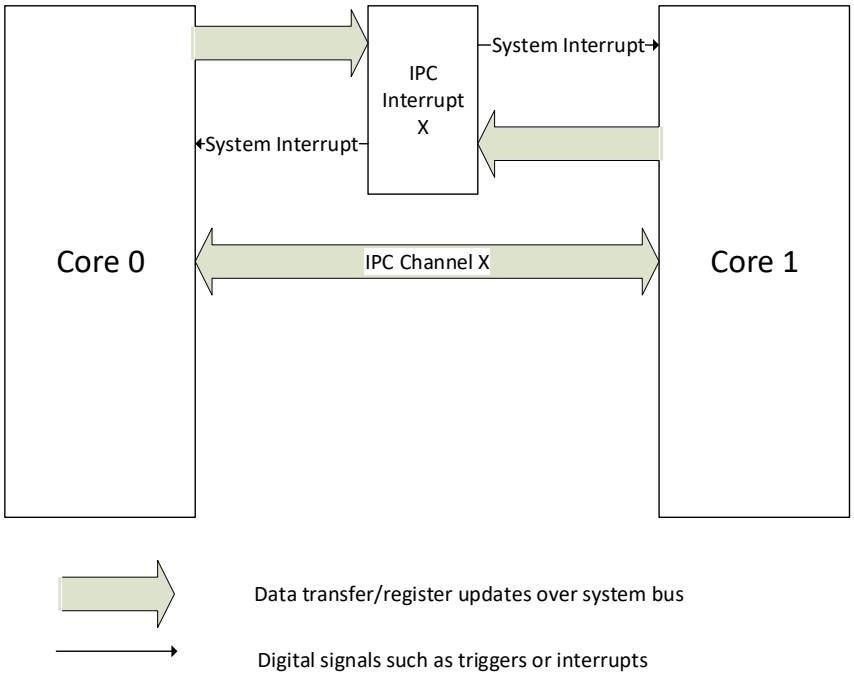
Figure 6-8. Half Duplex with Independent Event Handling



6.5.3 Half Duplex with Shared Event Handling

In this model both the IPC channel and interrupt are shared between the two cores. Since the interrupt is also shared, the access to the interrupt registers must be managed using the IPC lock of the channel. As shown in Figure 6-9, the IPC interrupt will be set up to trigger interrupts in both cores. Hence the individual core interrupts should have logic in its ISR to check which core is in control of the IPC and determine if the message and event was for that core.

Figure 6-9. Half Duplex with Shared Event Handling



Note: Some IPC channel and interrupt structures are reserved as part of the SROM code. Refer to the SROM architecture and API in Flash Memory Programming on page 167 for a list IPC channels and interrupts being used by this API.

7. Fault Monitoring

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

Fault monitoring allows you to monitor various faults generated within the device and take actions based on the fault reported. The fault structures present in the PSoC 6 MCU monitor access violation faults at protection units (MPU, SMPU, or PPU) and flash controller bus error/fault. In addition to reporting faults, the fault structures in PSoC 6 MCUs provide a mechanism to log data from the fault sources and optionally perform soft reset.

The PSoC 6 MCU family supports two centralized fault report/monitoring structures that monitor faults generated within the device. Each fault report structure can monitor and report faults from up to 96 sources.

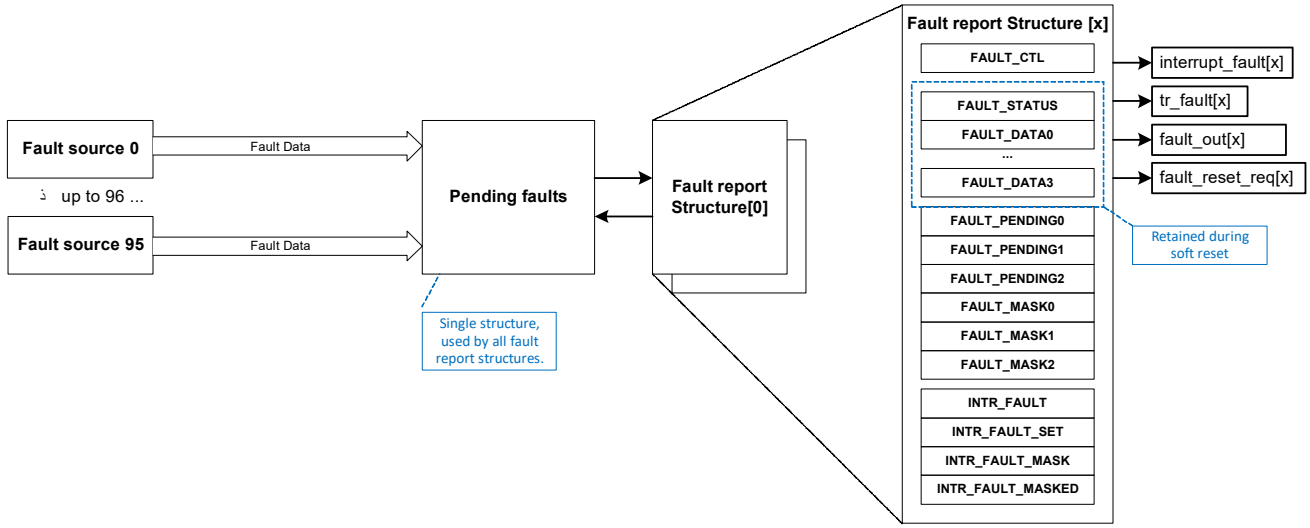
7.1 Features

Each PSoC 6 MCU fault report structure supports:

- Monitoring protection unit access violation faults and flash controller bus errors
- Four 32-bit data registers to record fault information
- Soft reset on fault detection while retaining the fault information
- Interrupt on fault detection
- Trigger output to DMA for fault data transfer
- Fault detected output to a pin for external fault handling

7.2 Architecture

Figure 7-1. Fault Report Structure



The PSoC 6 MCU family uses centralized fault report structures. This centralized nature allows for a system wide handling of faults simplifying firmware development. Only a single fault interrupt handler is required to monitor multiple faults. The fault report structure provides the fault source and additional fault specific information through a single set of registers; no iterative search for the fault source and fault information is required.

The fault structure can be configured to capture one or more faults as listed in Table 7-2. When a fault structure is configured to capture a specific fault, an occurrence of that fault will be recorded as a pending fault. If the fault structure has finished processing all other faults or if there are no other pending faults, the fault data will be captured into the fault structure registers. In addition, a successful capture can trigger an interrupt and be processed by either Cortex-M4 or Cortex-M0+ depending on the application requirement.

It should be noted that each fault structure is capable of capturing only one fault at a time and as long as that fault is not serviced, subsequent faults will not be captured by the fault structure. In addition to capturing faults, the fault structure can optionally perform a soft reset while retaining the fault information. This reset results in RESET_ACT_FAULT reset cause in the SRSS_RES_CAUSE register.

7.2.1 Fault Report

The PSoC 6 MCU family supports two fault report structures. Each fault report structure has a dedicated set of control and status registers. Each fault report structure captures a single fault. The captured fault information includes:

- Fault validity bit that indicates a fault is captured (VALID bit [31] of the FAULT_STRUCTx_STATUS register). This bit is set whenever a fault is captured. The bit should be cleared after processing the fault information. New faults are captured only when this bit is ‘0’.
- Fault index, as shown in Table 7-2, identifies the fault source (IDX bits [6:0] of FAULT_STRUCTx_STATUS)
- Additional fault information describing fault specifics (FAULT_STRUCTx_DATA0 through FAULT_STRUCTx_DATA3 registers). This additional information is fault source specific. For example, an MPU protection violation provides

information on the violating bus address, the bus master identifier, and bus access control information in only two FAULT_DATA registers. The details of the fault information for various faults is explained in Table 7-1.

Table 7-1. Fault Information

Fault Source	Fault Information
MPU/SMPU violation	DATA0[31:0]: Violating address. DATA1[0]: User read. DATA1[1]: User write. DATA1[2]: User execute. DATA1[3]: Privileged read. DATA1[4]: Privileged write. DATA1[5]: Privileged execute. DATA1[6]: Non-secure. DATA1[11:8]: Master identifier. DATA1[15:12]: Protection context identifier. DATA1[31]: '0' MPU violation; '1': SMPU violation.
Master interface PPU violation	DATA0[31:0]: Violating address. DATA1[0]: User read. DATA1[1]: User write. DATA1[2]: User execute. DATA1[3]: Privileged read. DATA1[4]: Privileged write. DATA1[5]: Privileged execute. DATA1[6]: Non-secure. DATA1[11:8]: Master identifier. DATA1[15:12]: Protection context identifier. DATA1[31]: '0': PPU violation, '1': peripheral bus error.
Peripheral group PPU violation	DATA0[31:0]: Violating address. DATA1[0]: User read. DATA1[1]: User write. DATA1[2]: User execute. DATA1[3]: Privileged read. DATA1[4]: Privileged write. DATA1[5]: Privileged execute. DATA1[6]: Non-secure. DATA1[11:8]: Master identifier. DATA1[15:12]: Protection context identifier. DATA1[31:30]: ‘0’: PPU violation, ‘1’: timeout detected, ‘2’: peripheral bus error.
Flash controller bus error	FAULT_DATA0[31:0]: Violating address. FAULT_DATA1[31]: '0': FLASH macro interface bus error; '1': memory hole. FAULT_DATA1[15:12]: Protection context identifier. FAULT_DATA1[11:8]: Master identifier.

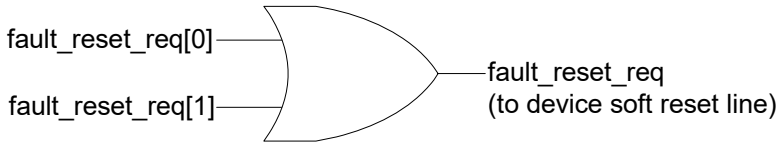
7.2.2 Signaling Interface

In addition to captured fault information, each fault report structure supports a signaling interface to notify the system about the captured fault. The interface of fault report structure 'x' supports the following:

- A fault interrupt (interrupt_fault[x]). Use the FAULT_STRUCTx_INTR, FAULT_STRUCTx_INTR_SET, FAULT_STRUCTx_INTR_MASK and FAULT_STRUCTx_INTR_MASKED registers to monitor, set, and mask the FAULT_STRUCTURE[x]'s interrupt. Only a single interrupt cause is available, which indicates that a fault is detected. The fault report registers can be read in the interrupt handler to deduce the fault. The FAULT bit [0] of the FAULT_STRUCTx_INTR_MASK register provides a mask/enable for the interrupt. The FAULT bit [0] of the FAULT_STRUCTx_INTR register is set to '1' when a fault is captured. Setting this bit in firmware clears the interrupt.
- A DMA trigger (tr_fault[x]). The fault structure generates a DMA trigger when VALID bit [31] of the FAULT_STRUCTx_STATUS register is set. To enable the trigger, set the TR_EN bit [0] of the FAULT_STRUCTx_CTL register. The trigger can be connected to a DMA controller, which can transfer captured fault information from the fault report structure to memory and can clear the VALID bit [31] of the FAULT_STRUCTx_STATUS register. See the Trigger Multiplexer Block chapter on page 294 for more details.
- A chip output signal (fault_out[x]). The fault structure generates an output signal, which is set when VALID bit [31] of the FAULT_STRUCTx_STATUS register is set. This signal can be routed out of the device through the HSIOM (refer to the PSoC 61 datasheet/PSoC 62 datasheet). The output signal is enabled by setting the OUT_EN bit [1] of the FAULT_STRUCTx_CTL register. The output signal can be used to communicate non-recoverable faults to off-chip components (possibly resulting in a reset of the chip).
- A fault reset request signal (fault_reset_req[x]). The fault structure generates a soft reset when VALID bit [31] of the FAULT_STRUCTx_STATUS register is set. The reset capability is enabled by setting RESET_REQ_EN bit [2] of FAULT_STRUCTx_CTL. The reset request performs a soft reset. This reset is captured as RESET_ACT_FAULT in the SRSS_RES_CAUSE register. The fault information in FAULT_STRUCTx_STATUS and FAULT_STRUCTx_DATA registers is retained through this reset.

Because the device has a single fault_reset_req signal, the individual fault_reset_req[x] signals from the fault structures are combined into a single fault_reset_req signal as shown in Figure 7-2.

Figure 7-2. Fault Reset Request



7.2.3 Monitoring

A central structure, which is shared by all fault report structures, keeps track of all pending faults in the system. The FAULT_STRUCTx_PENDINGx registers reflect what fault sources are pending and provide a single pending bit for up to 96 fault sources. The registers are mirrored in all the fault report structures; that is, they read the same value in all fault structures. The bit indexing in the registers follow the fault index captured in Table 7-2. For instance, bit [0] of FAULT_STRUCTx_PENDING0 captures a CM0+ MPU/SMPU violation and bit [1] of FAULT_STRUCTx_PENDING1 captures a peripheral group#1 PPU violation.

The pending faults are faults that are not yet captured by a fault structure. When a pending fault is captured by a fault structure, the associated pending bit is cleared to '0'.

Each fault report structure is selective in the faults it captures. The FAULT_STRUCTx_MASK0, FAULT_STRUCTx_MASK1, FAULT_STRUCTx_MASK2 registers of a fault structure decide the pending faults that it captures. These faults are referred to as "enabled" faults. The FAULT_STRUCTx_MASK registers are unique to each fault structure. This allows for the following:

- One fault report structure is used to capture recoverable faults and one fault report structure is used to capture non-recoverable faults. The former can be used to generate a fault interrupt and the latter can be used to activate a chip output signal and/or activate a reset request.
- Two fault report structures are used to capture the same faults. The first fault is captured by the structure with the lower index (for example, fault structure 0) and the second fault is captured by the structure with the higher index (for example, fault structure 1). Note that both structures cannot capture the same fault at the same time. As soon as a fault is captured, the pending bit is cleared and the other structure will not be aware of the

fault. Fault structure 0 has precedence over fault structure 1.

The fault structure captures “enabled” faults only when VALID bit [31] of FAULT_STRUCTx_STATUS register is ‘0’. When a fault is captured, hardware sets the VALID bit [31] of the FAULT_STRUCTx_STATUS register. In addition, hardware clears the associated pending bit to ‘0’. When a fault structure is processed, firmware or a DMA transfer should clear the VALID bit [31] of the FAULT_STRUCTx_STATUS register. Note that fault capturing does not consider FAULT bit [0] of FAULT_STRUCTx_INTR register and firmware should clear the bit after servicing the interrupt, if the interrupt is enabled.

7.2.4 Low-power Mode Operation

The fault report structure functionality is available in Active and Sleep (and their LP counterparts) power modes only. The interfaces between the fault sources and fault report structures are reset in the Deep Sleep power mode. Because the fault report structure is an active functionality, pending faults (in the FAULT_STRUCTx_PENDING registers) are not retained when transitioning to Deep Sleep power mode. The fault structure’s registers can be partitioned based on the reset domain and their retention capability as follows:

- Active reset domain: FAULT_STRUCTx_PENDING, FAULT_STRUCTx_INTR, FAULT_STRUCTx_INTR_SET, and FAULT_STRUCTx_INTR_MASKED registers. These registers are not retained in Deep Sleep power mode.
- Deep Sleep reset domain: FAULT_STRUCTx_CTL, FAULT_STRUCTx_MASK, and FAULT_STRUCTx_INTR_MASK registers. These registers are retained in Deep Sleep power mode but any system reset will reset these registers to the default state.
- Hard reset domain: FAULT_STRUCTx_STATUS and FAULT_STRUCTx_DATA registers. These registers are retained through soft resets (detectable in SRSS_RES_CAUSE registers). However, hard resets such as XRES/POR/BOD will reset the registers.

7.2.5 Using a Fault Structure

Follow these steps to configure and use a fault structure:

1. Identify the faults from Table 7-2 to be monitored in the system.
2. For firmware fault handling through interrupts
   a. Set the FAULT bit [0] of the FAULT_STRUCTx_INTR_MASK register.
   b. Set the FAULT bit [0] of the FAULT_STRUCTx_INTR register to clear any pending interrupt.
   c. Enable the FAULTx interrupt to the CPU by configuring the appropriate ISER register. Refer to the Interrupts chapter on page 56.
3. For fault handling through DMA
   a. Set the TR_EN bit [0] of the FAULT_STRUCTx_CTL register.
   b. Route the tr_fault[x] signal to the trigger input DMA controller. Refer to the Trigger Multiplexer Block chapter on page 294.
   c. Configure and enable the DMA controller to transfer FAULT_STRUCTx_STATUS and FAULT_STRUCTx_DATA registers to memory and write back ‘0’ to FAULT_STRUCTx_STATUS register after the transfer is complete. Refer to the DMA Controller (DW) chapter on page 91.
4. For fault handling outside the device
   a. Set the OUT_EN bit [1] of FAULT_STRUCTx_CTL register.
   b. Route the fault_out[x] signal to a pin through HSIOM. Refer to the PSoC 61 datasheet/PSoC 62 datasheet.
   c. Use the signal externally for processing the fault – generate external reset, power cycle, or log fault information.
5. Set the RESET_REQ_EN bit [2] of the FAULT_STRUCTx_CTL register, if a soft reset is required on any fault detection in the structure.
6. Clear VALID bit [31] of the FAULT_STRUCTx_STATUS register to clear any fault captured.
7. Set the fault index bits in the FAULT_STRUCTx_MASK registers for faults that need to be captured by the fault structure as explained in 7.2.3 Monitoring.

7.2.6 CPU Exceptions Versus Fault Monitoring

Some faults captured in Table 7-2 also result in bus errors or CPU exceptions (Cortex-M4 Bus/Usage/Memory/Hard faults). The faults can be communicated in two ways:

- As a bus error to the master of the faulting bus transfer. This will result in Bus, Usage, Memory, or Hard fault exceptions in the CPU.
- As a fault in a fault report structure. This fault can be communicated as a fault interrupt to any processor in the system. This allows fault handling on a processor that is not the master of the faulting bus transfer. It is useful for faults that cause the master of the faulting transfer to become unresponsive or unreliable.

7.3 Fault Sources

The fault sources can vary between device families. Table 7-2 provides the list of fault sources available in PSoC 6 MCUs.

Table 7-2. Fault Sources

Fault Index	Source	Description
0	cpuss.mpu_vio[0]	CM0+ MPU/SMPU violation
1	cpuss.mpu_vio[1]	CRYPTO MPU/SMPU violation
2	cpuss.mpu_vio[2]	DW0 MPU/SMPU violation
3	cpuss.mpu_vio[3]	DW1 MPU/SMPU violation
4	cpuss.mpu_vio[4]	DMAC MPU/SMPU violation
5	cpuss.mpu_vio[5]	SDHC0 MPU/SMPU violation
6		
7 to 15	Reserved	
16	cpuss.mpu_vio[16]	CM4 MPU/SMPU violation (System bus)
17	cpuss.mpu_vio[17]	CM4 code error
18	cpuss.mpu_vio[18]	CM4 code flash controller errors
19 to 27	Reserved	
28	peri.ms_vio[0]	CM0+ Peripheral Master Interface PPU violation
29	peri.ms_vio[1]	CM4 Peripheral Master Interface PPU violation
30	peri.ms_vio[2]	DW0 Peripheral Master Interface PPU violation
31	peri.ms_vio[3]	DW1 Peripheral Master Interface PPU violation
32	peri.group_vio[0]	Peripheral group #0 (Peripheral Clock dividers, Trigger Mux etc) PPU violation Register address range: 0x40000000 to 0x40100000
33	peri.group_vio[1]	Peripheral group #1 (Crypto block) PPU violation Register address range: 0x40100000 to 0x40200000
34	peri.group_vio[2]	Peripheral group #2 (CPUSS, SRSS, EFUSE) PPU violation Register address range: 0x40200000 to 0x40300000
35	peri.group_vio[3]	Peripheral group #3 (IOSS, LPCOMP, CSD, TCPWM, LCD, BLE) PPU violation Register address range: 0x40300000 to 0x40400000
36	peri.group_vio[4]	Peripheral group #4 (SMIF) PPU violation Register address range: 0x40400000 to 0x40500000
37		
38	peri.group_vio[6]	Peripheral group #6 (SCB) PPU violation Register address range: 0x40600000 to 0x40700000
39	Reserved	
40	Reserved	
41	peri.group_vio[9]	Peripheral group #9 (PASS) PPU violation Register address range: 0x41000000 to 0x42000000
42		
43 to 47	Reserved	
48	cpuss.flashc_main_bus_err	Flash controller bus error
49 to 95	Reserved	

7.4 Register List

Name	Description
FAULT_STRUCTx_CTL	Fault control register for enabling DMA trigger, fault output, and fault reset signals
FAULT_STRUCTx_STATUS	Fault status register that stores the validity and fault index of the currently captured fault
FAULT_STRUCTx_DATA0	Fault data register 0 that stores fault information associated with the currently captured fault
FAULT_STRUCTx_DATA1	Fault data register 1 that stores fault information associated with the currently captured fault
FAULT_STRUCTx_DATA2	Fault data register 2 that stores fault information associated with the currently captured fault
FAULT_STRUCTx_DATA3	Fault data register 3 that stores fault information associated with the currently captured fault
FAULT_STRUCTx_PENDING0	Fault pending register 0 that stores pending (not captured) faults with fault index from 0 to 31
FAULT_STRUCTx_PENDING1	Fault pending register 1 that stores pending (not captured) faults with fault index from 32 to 63
FAULT_STRUCTx_PENDING2	Fault pending register 2 that stores pending (not captured) faults with fault index from 64 to 95
FAULT_STRUCTx_MASK0	Fault mask register 0 that enables the capture of pending faults with fault index from 0 to 31 by the fault structure
FAULT_STRUCTx_MASK1	Fault mask register 1 that enables the capture of pending faults with fault index from 32 to 63 by the fault structure
FAULT_STRUCTx_MASK2	Fault mask register 2 that enables the capture of pending faults with fault index from 64 to 95 by the fault structure
FAULT_STRUCTx_INTR	Fault interrupt register that stores the unmasked status of the fault structure's interrupt
FAULT_STRUCTx_INTR_SET	Fault interrupt set register used to set the fault structure's interrupt through firmware
FAULT_STRUCTx_INTR_MASK	Fault interrupt mask register that masks fault interrupt
FAULT_STRUCTx_INTR_MASKED	Fault interrupt register that stores the masked status of the fault structure's interrupt

8. Interrupts

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU family supports interrupts and CPU exceptions on both Cortex-M4 and Cortex-M0+ cores. Any condition that halts normal execution of instructions is treated as an exception by the CPU. Thus an interrupt request is treated as an exception. However, in the context of this chapter, interrupts refer to those events generated by peripherals external to the CPU such as timers, serial communication block, and port pin signals; exceptions refer to those events that are generated by the CPU such as memory access faults and internal system timer events. Both interrupts and exceptions result in the current program flow being stopped and the exception handler or interrupt service routine (ISR) being executed by the CPU. Both Cortex-M4 and Cortex-M0+ cores provide their own unified exception vector table for both interrupt handlers/ISR and exception handlers.

8.1 Features

The PSoC 6 MCU supports the following interrupt features:

- Supports 168 system interrupts
  - 168 Arm Cortex-M4 interrupts
  - Eight Arm Cortex-M0+ external interrupts and eight Arm Cortex-M0+ internal (software only) interrupts. The CPU supports up to 32 interrupts, but only 16 interrupts are used by PSoC 6 interrupt infrastructure. The eight external CPU interrupts support deep sleep (WIC) functionality
  - Four system interrupts can be mapped to each of the CPU non-maskable interrupt (NMI)
  - Up to 39 interrupt sources capable of waking the device from Deep Sleep power mode
- Nested vectored interrupt controller (NVIC) integrated with each CPU core, yielding low interrupt latency
- Wakeup interrupt controller (WIC) enabling interrupt detection (CPU wakeup) in Deep Sleep power mode
- Vector table may be placed in either flash or SRAM
- Configurable priority levels (eight levels for Cortex-M4 and four levels for Cortex-M0+) for each interrupt
- Level-triggered and pulse-triggered interrupt signals

8.2 Architecture

Figure 8-1. PSoC 6 MCU Interrupts Block Diagram

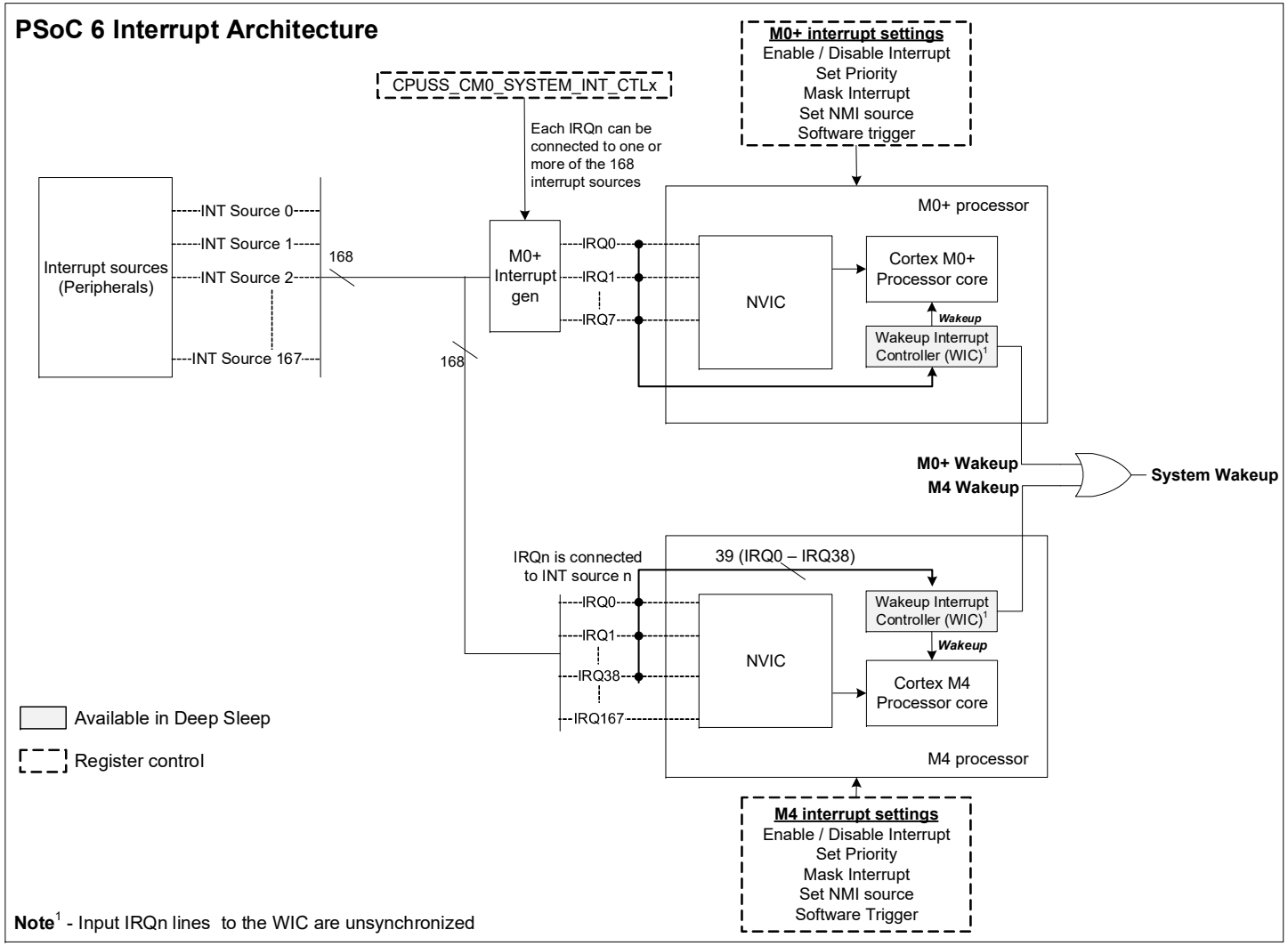


Figure 8-1 shows the PSoC 6 MCU interrupt architecture. The PSoC 6 MCU has 168 system interrupts that are generated by various peripherals. These interrupt signals are processed by the NVIC of the individual core. In the Cortex-M4 core, the system interrupt source 'n' is directly connected to IRQn. The Cortex-M0+ interrupt architecture uses eight CPU interrupts IRQ[7:0] out of the 32 available IRQn lines of the core. In the Cortex-M0+ core, the system interrupt source connection to a particular IRQn of the core is configurable and any of the 168 system interrupts can be mapped to any of the IRQ[7:0]. This ensures that all the system interrupts can be mapped onto any CPU interrupt simultaneously. Refer to Interrupt Sources on page 62 for more details about the system interrupt to CPU interrupt mapping. The NVIC takes care of enabling/disabling individual interrupt IRQs, priority resolution, and communication with the CPU core. The other exceptions such as NMI and hard faults are not shown in Figure 8-1 because they are part of CPU core generated events, unlike interrupts, which are generated by peripherals external to the CPU.

In addition to the NVIC, the PSoC 6 MCU supports wakeup interrupt controllers (WIC) for each CPU and a shared interrupt synchronization block that synchronizes the interrupts to CLK_HF domain (adds two CLK_HF cycles delay for synchronization). The WIC provides detection of Deep Sleep interrupts in the Deep Sleep CPU power mode. Each CPU can individually be in Deep Sleep mode; the device is said to be in Deep Sleep mode only when both the CPUs are in Deep Sleep mode. Refer to the Device Power Modes chapter on page 225 for details. The Cortex-M4 WIC block supports up to 39 interrupts that can wake up the CPU from Deep Sleep power mode. The Cortex-M0+ WIC block supports all eight interrupts. The device exits Deep Sleep mode (System Wakeup signal in Figure 8-1) as soon as one CPU wakes up. The synchronization blocks synchronize the interrupts to the CPU clock frequency as the peripheral interrupts can be asynchronous to the CPU clock frequency.

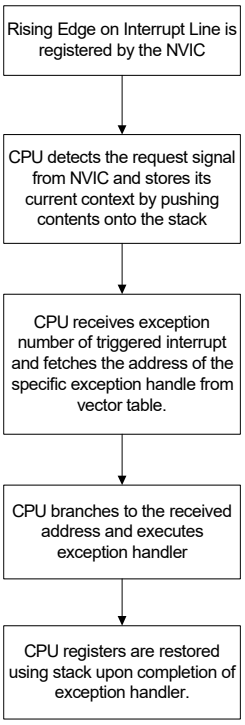
8.3 Interrupts and Exceptions - Operation

8.3.1 Interrupt/Exception Handling

The following sequence of events occurs when an interrupt or exception event is triggered:

1. Assuming that all the interrupt and exception signals are initially low (idle or inactive state) and the processor is executing the main code, a rising edge on any one of the signals is registered by the NVIC, if the interrupt or exception is enabled to be serviced by the CPU. The signal is now in a pending state waiting to be serviced by the CPU.
2. On detecting the signal from the NVIC, the CPU stores its current context by pushing the contents of the CPU registers onto the stack.
3. The CPU also receives the exception number of the triggered interrupt from the NVIC. All interrupts and exceptions have a unique exception number, as given in Table 8-1. By using this exception number, the CPU fetches the address of the specific exception handler from the vector table.
4. The CPU then branches to this address and executes the exception handler that follows.
5. Upon completion of the exception handler, the CPU registers are restored to their original state using stack pop operations; the CPU resumes the main code execution.

Figure 8-2. Interrupt Handling When Triggered



When the NVIC receives an interrupt request while another interrupt is being serviced or receives multiple interrupt requests at the same time, it evaluates the priority of all these interrupts, sending the exception number of the highest priority interrupt to the CPU. Thus, a higher priority interrupt can block the execution of a lower priority ISR at any time.

Exceptions are handled in the same way that interrupts are handled. Each exception event has a unique exception number, which is used by the CPU to execute the appropriate exception handler.

8.3.2 Level and Pulse Interrupts

Both CM0+ and CM4 NVICs support level and pulse signals on the interrupt lines (IRQn). The classification of an interrupt as level or pulse is based on the interrupt source.

Figure 8-3. Level Interrupts

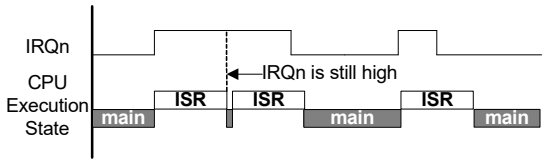


Figure 8-4. Pulse Interrupts

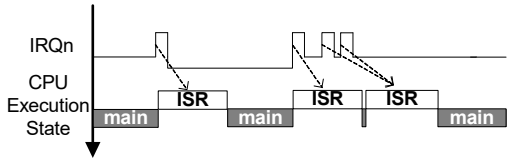


Figure 8-3 and Figure 8-4 show the working of level and pulse interrupts, respectively. Assuming the interrupt signal is initially inactive (logic low), the following sequence of events explains the handling of level and pulse interrupts:

1. On a rising edge event of the interrupt signal, the NVIC registers the interrupt request. The interrupt is now in the pending state, which means the interrupt requests have not yet been serviced by the CPU.
2. The NVIC then sends the exception number along with the interrupt request signal to the CPU. When the CPU starts executing the ISR, the pending state of the interrupt is cleared.
3. For pulse interrupts, when the ISR is being executed by the CPU, one or more rising edges of the interrupt signal are logged as a single pending request. The pending interrupt is serviced again after the current ISR execution is complete (see Figure 8-4 for pulse interrupts).
4. For level interrupts, if the interrupt signal is still high after completing the ISR, it will be pending and the ISR is executed again. Figure 8-3 illustrates this for level triggered interrupts, where the ISR is executed as long as the interrupt signal is high.

8.3.3 Exception Vector Table

The exception vector tables (Table 8-1 and Table 8-2) store the entry point addresses for all exception handlers in Cortex-M0+ and Cortex-M4 cores. The CPU fetches the appropriate address based on the exception number.

Table 8-1. M0+ Exception Vector Table

Exception Number	Exception	Exception Priority	Vector Address
–	Initial Stack Pointer Value	Not applicable (NA)	Start_Address = 0x0000 or CM0P_SCS_VTOR[a]
1	Reset	–3, the highest priority	Start_Address + 0x04
2	Non Maskable Interrupt (NMI)	–2	Start_Address + 0x08
3	HardFault	–1	Start_Address + 0x0C
4-10	Reserved	NA	Start_Address + 0x10 to Start_Address + 0x28
11	Supervisory Call (SVCall)	Configurable (0 – 3)	Start_Address + 0x2C
12-13	Reserved	NA	Start_Address + 0x30 to Start_Address + 0x34
14	PendSupervisory (PendSV)	Configurable (0 – 3)	Start_Address + 0x38
15	System Timer (SysTick)	Configurable (0 – 3)	Start_Address + 0x3C
16	External Interrupt (IRQ0)	Configurable (0 – 3)	Start_Address + 0x40
…	…	Configurable (0 – 3)	…
31	External Interrupt (IRQ15)	Configurable (0 – 3)	Start_Address + 0x7C

a. Start Address = 0x0000 on reset and is later modified by firmware by updating the CM0P_SCS_VTOR register.

Table 8-2. Cortex-M4 Exception Vector Table

Exception Number	Exception	Exception Priority	Vector Address
–	Initial stack pointer value	–	Start_Address = 0x0000 or CM4_SCS_VTOR[a]
1	Reset	–3, highest priority	Start _Address + 0x0004
2	Non Maskable Interrupt (NMI)	–2	Start _Address + 0x0008
3	Hard fault	–1	Start _Address + 0x000C
4	Memory management fault	Configurable (0 – 7)	Start _Address + 0x0010
5	Bus fault	Configurable (0 – 7)	Start _Address + 0x0014

Table 8-2. Cortex-M4 Exception Vector Table *(continued)*

Exception Number	Exception	Exception Priority	Vector Address
6	Usage fault	Configurable (0 – 7)	Start _Address + 0x0018
7–10	Reserved	–	–
11	Supervisory call (SVCall)	Configurable (0 – 7)	Start _Address + 0x002C
12–13	Reserved	–	–
14	Pend Supervisory (PendSV)	Configurable (0 – 7)	Start _Address + 0x0038
15	System Tick timer (SysTick)	Configurable (0 – 7)	Start _Address + 0x003C
16	External interrupt (IRQ0)	Configurable (0 – 7)	Start _Address + 0x0040
….	….	….	….
182	External interrupt (IRQ166)	Configurable (0 – 7)	Start _Address + 0x02D8
183	External interrupt (IRQ167)	Configurable (0 – 7)	Start _Address + 0x02DC

a. Start Address = 0x0000 on reset and is later modified by firmware by updating CM4_SCS_VTOR register.

In Table 8-1 and Table 8-2, the first word (4 bytes) is not marked as exception number zero. This is because the first word in the exception table is used to initialize the main stack pointer (MSP) value on device reset; it is not considered as an exception. In the PSoC 6 MCU, both the vector tables can be configured to be located either in flash memory or SRAM. The vector table offset register (VTOR) present as part of Cortex-M0+ and Cortex-M4 system control space registers configures the vector table offset from the base address (0x0000). The CM0P_SCS_VTOR register sets the vector offset address for the CM0+ core and CM4_SCS_VTOR sets the offset for the M4 core. The VTOR value determines whether the vector table is in flash memory (0x10000000 to 0x10100000) or SRAM (0x08000000 to 0x08048000). Note that the VTOR registers can be updated only in privilege CPU mode. The advantage of moving the vector table to SRAM is that the exception handler addresses can be dynamically changed by modifying the SRAM vector table contents. However, the nonvolatile flash memory vector table must be modified by a flash memory write. Note that the exception table must be 256 byte-aligned for Cortex-M0+ and 1024 byte-aligned for Cortex-M4.

The exception sources (exception numbers 1 to 15) are explained in 8.4 Exception Sources. The exceptions marked as Reserved in Table 8-1 are not used, although they have addresses reserved for them in the vector table. The interrupt sources (exception numbers 16 to 183) are explained in 8.5 Interrupt Sources.

8.4 Exception Sources

This section explains the different exception sources listed in Table 8-1 and Table 8-2 (exception numbers 1 to 15).

8.4.1 Reset Exception

Device reset is treated as an exception in PSoC 6 MCUs. Reset exception is always enabled with a fixed priority of –3, the highest priority exception in both the cores. When the device boots up, only the Cortex-M0+ core is available. The CM0+ core executes the ROM boot code and can enable Cortex-M4 core from the application code. The reset exception of the CM0+ core is tied to the device reset or startup. When the Cortex-M0+ core releases the Cortex-M4 reset, the M4 reset exception is executed. A device reset can occur due to multiple reasons, such as power-on-reset (POR), external reset signal on XRES pin, or watchdog reset. When the device is reset, the initial boot code for configuring the device is executed by the Cortex-M0+ out of supervisory read-only memory (SROM). The boot code and other data in SROM memory are programmed by Cypress, and are not read/write accessible to external users. After completing the SROM boot sequence, the Cortex-M0+ code execution jumps to flash memory. Flash memory address 0x10000004 (Exception#1 in Table 8-1) stores the location of the startup code in flash memory. The CPU starts executing code out of this address. Note that the reset exception address in the SRAM vector table will never be used because the device comes out of reset with the flash vector table selected. The register configuration to select the SRAM vector table can be done only as part of the startup code in flash after the reset is de-asserted. Note that the reset exception flow for Cortex-M4 is the same as Cortex-M0+. However, Cortex-M4 execution begins only after CM0+ core de-asserts the M4 reset.

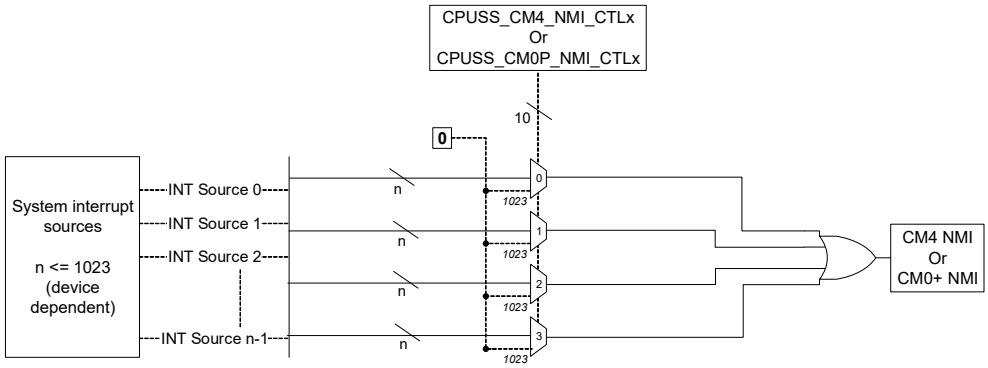
8.4.2 Non-Maskable Interrupt Exception

Non-maskable interrupt (NMI) is the highest priority exception next to reset. It is always enabled with a fixed priority of –2. Both the cores have their own NMI exception. There are three ways to trigger an NMI exception in a CPU core:

- **NMI exception from a system interrupt:** Both Cortex-M0+ and Cortex-M4 provide an option to trigger an NMI exception from up to four of the 168 system interrupts. The NMI exception triggered due to the interrupt will execute the NMI handler pointed to by the active exception vector table. The CPUSS_CM4_NMI_CTLx and CPUSS_CM0P_NMI_CTLx registers select the interrupt source that triggers the NMI from hardware.

The registers have a default value of 1023; that is, if the register is set to 1023, then that particular register does not map any interrupt source to the NMI. There are four such registers and each can map one interrupt vector to the NMI. NMI is triggered when any of the four interrupts are triggered; that is, the interrupts are logically ORed. See Figure 8-5.

Figure 8-5. NMI Trigger



- **NMI exception by setting NMIPENDSET bit (user NMI exception):** An NMI exception can be triggered in software by setting the NMIPENDSET bit in the interrupt control state registers (CM0P_SCS_ICSR and CM4_SCS_ICSR). Setting this bit will execute the NMI handler pointed to by the active vector table in the respective CPU cores.
- **System Call NMI exception:** This exception is used for nonvolatile programming and other system call operations such as flash write operation and flash checksum operation. Inter processor communication (IPC) mechanism is used to implement a system call in PSoC 6 MCUs. A dedicated IPC mailbox is associated with each core (M0+ and M4) and the debug access port (DAP) to trigger a system call. The CPU or DAP acquires this dedicated mailbox, writes the system call opcode and argument to the mailbox, and notifies a dedicated IPC structure. Typically, the argument is a pointer to a structure in SRAM. This results in an NMI interrupt in the CM0+ core. Note that all the system calls are serviced by Cortex-M0+ core. A Cortex-M0+ NMI exception triggered by this method executes the NMI exception handler code that resides in SROM. Note that the NMI exception handler address is automatically initialized to the system call API located in SROM (at 0x0000000D) by the boot code. The value should be retained during vector table relocations; otherwise, no system call will be executed. The NMI handler code in SROM is not read/write accessible because it contains nonvolatile programming routines that cannot be modified by the user. The result of the system call is passed through the same IPC mechanism. For details, refer to the Inter-Processor Communication chapter on page 41.

8.4.3 HardFault Exception

Both CM0+ and CM4 cores support HardFault exception. HardFault is an always-enabled exception that occurs because of an error during normal or exception processing. HardFault has a fixed priority of –1, meaning it has higher priority than any exception with configurable priority. A HardFault exception is a catch-all exception for different types of fault conditions, which include executing an undefined instruction and accessing an invalid memory addresses. The CPU does not provide fault status information to the HardFault exception handler, but it does permit the handler to perform an exception return and continue execution in cases where software has the ability to recover from the fault situation.

8.4.4 Memory Management Fault Exception

A memory management fault is an exception that occurs because of a memory protection-related fault. The fixed memory protection constraints determine this fault, for both instruction and data memory transactions. This fault is always used to abort instruction accesses to Execute Never (XN) memory regions. The memory management fault is only supported by the M4 core. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

8.4.5 Bus Fault Exception

A Bus Fault is an exception that occurs because of a memory-related fault for an instruction or data memory transaction. This might be from an error detected on a bus in the memory system. The bus fault is supported only by the M4 core. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

8.4.6 Usage Fault Exception

A Usage Fault is an exception that occurs because of a fault related to instruction execution. This includes:

- an undefined instruction
- an illegal unaligned access

- invalid state on instruction execution
- an error on exception return

The following can cause a usage fault when the core is configured to report them:

- an unaligned address on word and halfword memory access
- division by zero

The usage fault is supported only by the M4 core. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

8.4.7 Supervisor Call (SVCall) Exception

Both CM0+ and CM4 cores support SVCall exception. Supervisor Call (SVCall) is an always-enabled exception caused when the CPU executes the SVC instruction as part of the application code. Application software uses the SVC instruction to make a call to an underlying operating system and provide a service. This is known as a supervisor call. The SVC instruction enables the application to issue an SVCall that requires privileged access to the system.

The priority of an SVCall exception can be configured to a value between 0 and 3 for CM0+ and 0 to 7 for CM4 core by writing to the bitfields PRI_11 of the System Handler Priority Register 2 (CM0P_SCS_SHPR2 and CM4_SCS_SHPR2). When the SVC instruction is executed, the SVCall exception enters the pending state and waits to be serviced by the CPU. The SVCALLPENDED bit in the System Handler Control and State Register (CM0P_SCS_SHCSR and CM4_SCS_SHCSR) can be used to check or modify the pending status of the SVCall exception.

8.4.8 PendSupervisory (PendSV) Exception

Both CM0+ and CM4 cores support PendSV exception. PendSV is another supervisor call related exception similar to SVCall, normally being software-generated. PendSV is always enabled and its priority is configurable similar to SVCall. The PendSV exception is triggered by setting the PENDSVSET bit in the Interrupt Control State Register (CM0P_SCS_ICSR and CM4_SCS_ICSR). On setting this bit, the PendSV exception enters the pending state, and waits to be serviced by the CPU. The pending state of a PendSV exception can be cleared by setting the PENDSVCLR bit in the Interrupt Control State Register. The priority of a PendSV exception can be configured to a value between 0 and 3 for CM0+ and 0 to 7 for M4 by writing to the bitfields PRI_14 of the System Handler Priority Register 3. See the Armv6-M Architecture Reference Manual for more details.

8.4.9 System Tick (SysTick) Exception

Both CM0+ and CM4 cores in PSoC 6 MCUs support a system timer, referred to as SysTick, as part of their internal architecture. SysTick provides a simple, 24-bit decrementing counter for various timekeeping purposes such as an RTOS tick timer, high-speed alarm timer, or simple counter. The SysTick timer can be configured to generate an interrupt when its count value reaches zero, which is referred to as a SysTick exception. The exception is enabled by setting the TICKINT bit in the SysTick Control and Status Register (CM0P_SCS_SYST_CSR and CM4_SCS_SYST_CSR). The priority of a SysTick exception can be configured to a value between 0 and 3 for CM0+ and 0 to 7 for M4 by writing to the bitfields PRI_15 of the System Handler Priority Register 3 (SHPR3). The SysTick exception can always be generated in software at any instant by writing a one to the PENDSTSET bit in the Interrupt Control State Register. Similarly, the pending state of the SysTick exception can be cleared by writing a one to the PENDSTCLR bit in the Interrupt Control State Register.

8.5 Interrupt Sources

The PSoC 6 MCU supports 168 interrupts from peripherals. The source of each interrupt is listed in Table 8-3. These system interrupts are mapped directly to Cortex-M4 core (IRQ0 to IRQ167 or exception 16 to 183). For Cortex-M0+ core, any of the 168 interrupts can be routed to any of the available eight interrupts (IRQ0 to IRQ7 or exception 16 to 23). The CPUSS_CM0_SYSTEM_INT_CTLx registers are used to make this interrupt selection in CM0+.

There are 168 CPUSS_CM0_SYSTEM_INT_CTLx registers. The CPU_INT_IDX bits [2:0] of the register selects which IRQn line of CM0+, the x^{th} system interrupt is mapped to. The CPU_INT_VALID bit [31] of the register decides whether the x^{th} interrupt is enabled for the core. For Cortex-M0+, multiple system interrupts can be mapped on the same CPU interrupt. Therefore, an active CPU interrupt may indicate one or multiple active system interrupts. For instance to connect Port 0 (IDX = 0) and Port 2 (IDX = 2) GPIO interrupts to IRQ0 of CM0+, CPU_INT_IDX bits should be set to ‘0’ and CPU_INT_VALID bit should be set to ‘1’ in CPUSS_CM0_SYSTEM_INT_CTL0 and CPUSS_CM0_SYSTEM_INT_CTL2 registers. Note that this only connects and enables the interrupts; configuration and masking of the interrupt to the NVIC should be done as part of source peripheral configuration.

As a result of the reduction functionality in the Cortex-M0+ core, multiple system interrupts share a CPU interrupt handler as provided by the CPU’s vector table. Each CPU interrupt has an associated CPUSS_CM0_INTx_STATUS register:

- SYSTEM_INT_VALID bit[31] of the register specifies if any system interrupt is active for the CPU interrupt. This

bit remains set until all the system interrupts connected to the CPU interrupts are cleared at the source.

- SYSTEM_INT_IDX bits[9:0] of the register specifies the index (a number in the range [0, 1022]) of the lowest active system interrupt mapped to the corresponding CPU interrupt.

For instance, say CM0+ IRQ0 is connected to Port 0 and Port 2 GPIO interrupts. When an interrupt in both Port 0 and Port 2 are triggered simultaneously, IRQ0 in CM0+ will be triggered with CPUSS_CM0_INT0_STATUS reading 0 (Port 0 interrupt) in SYSTEM_INT_IDX bit and SYSTEM_INT_VALID bit set. After Port 0 interrupt is serviced and cleared, IRQ0 will be triggered again with SYSTEM_INT_IDX bit set to 2 (Port 2 interrupt) in the CPUSS_CM0_INT0_STATUS register. Only after servicing Port 2 interrupt and clearing, the SYSTEM_INT_VALID bit will be cleared.

The CPU interrupt handler should use the SYSTEM_INT_IDX field to index a system interrupt lookup table and jump to the system interrupt handler. The lookup table is typically located in one of the system memories. The following code illustrates the approach:

```
typedef void (* SystemIntr_Handler)(void);
void CM0_CpuIntr0_Handler (void)
{
uint32_t system_int_idx;
SystemIntr_Handler handler;
if(CPUSS_CM0_INT0_STATUS.SYSTEM_INT_VALID)
{
system_int_idx = CPUSS_CM0_INT0_STATUS.SYSTEM_INT_IDX;
handler = SystemIntr_Table[system_int_idx];
handler(); // jump to system interrupt handler
}
else
{
// Triggered by SW or due to SW clear error (SW cleared a peripheral
// interrupt flag but didn't clear the Pending flag at NVIC)
}
}
…
void CM0_CpuIntr7_Handler (void)
{
uint32_t system_int_idx;
SystemIntr_Handler handler;
if(CPUSS_CM0_INT7_STATUS.SYSTEM_INT_VALID)
{
system_int_idx = CPUSS_CM0_INT7_STATUS.SYSTEM_INT_IDX;
handler = SystemIntr_Table[system_int_idx];
handler(); // jump to system interrupt handler
}
else
{
// Triggered by SW or due to SW clear error (SW cleared a peripheral
// interrupt flag but didn't clear the Pending flag at NVIC)
}
}
void CM0_SystemIntr0_Handler (void)
{
// Clear the peripheral interrupt request flag by register write
// Read back the register, to ensure the completion of register write access
// Handle system interrupt 0.
}
…
void CM0_SystemIntr167_Handler (void)
{
// Clear the peripheral interrupt request flag by register write
// Read back the register, to ensure the completion of register write access
```

```
// Handle system interrupt 167.
}
```

The interrupts include standard interrupts from the on-chip peripherals such as TCPWM, serial communication block, CSD block, watchdog, ADC, and so on. The interrupt generated is usually the logical OR of the different peripheral states. The peripheral interrupt status register should be read in the ISR to detect which condition generated the interrupt. These interrupts are usually level interrupts. The appropriate interrupt registers should be cleared in the ISR to deassert the interrupt. Usually a write '1' is required to clear the registers. If the interrupt register is not cleared in the ISR, the interrupt will remain asserted and the ISR will be executed continuously. See the I/O System chapter on page 261 for details on GPIO interrupts.

As seen from Table 8-3, 39 interrupts (IRQ0 to IRQ38) are capable of waking up the device from Deep Sleep power mode. For Cortex-M4, IRQ0 to IRQ38 directly map to these sources. However, in the Cortex-M0+, all the eight IRQ lines support Deep Sleep wakeup if they are connected to a deep sleep capable interrupt. This means the 39 Deep Sleep wakeup-capable interrupts can be connected to any of the eight IRQ lines of Cortex-M0+, if such a wakeup is desired.

Table 8-3. List of PSoC 6 Interrupts

System Interrupt	Cortex M4 Exception Number	Power Mode	Interrupt Source
NMI	2	Active	Any of the below 168 IRQ source
IRQ0	16	DeepSleep	GPIO Interrupt - Port 0
IRQ1	17	DeepSleep	GPIO Interrupt - Port 1
IRQ2	18	DeepSleep	GPIO Interrupt - Port 2
IRQ3	19	DeepSleep	GPIO Interrupt - Port 3
IRQ4	20	DeepSleep	GPIO Interrupt - Port 4
IRQ5	21	DeepSleep	GPIO Interrupt - Port 5
IRQ6	22	DeepSleep	GPIO Interrupt - Port 6
IRQ7	23	DeepSleep	GPIO Interrupt - Port 7
IRQ8	24	DeepSleep	GPIO Interrupt - Port 8
IRQ9	25	DeepSleep	GPIO Interrupt - Port 9
IRQ10	26	DeepSleep	GPIO Interrupt - Port 10
IRQ11	27	DeepSleep	GPIO Interrupt - Port 11
IRQ12	28	DeepSleep	GPIO Interrupt - Port 12
IRQ13	29	DeepSleep	GPIO Interrupt - Port 13
IRQ14	30	DeepSleep	GPIO Interrupt - Port 14
IRQ15	31	DeepSleep	GPIO Interrupt - All Ports
IRQ16	32	DeepSleep	GPIO Supply Detect Interrupt
IRQ17	33	DeepSleep	Low Power Comparator Interrupt
IRQ18	34	DeepSleep	Serial Communication Block #8 (DeepSleep capable)
IRQ19	35	DeepSleep	Multi Counter Watchdog Timer interrupt
IRQ20	36	DeepSleep	Multi Counter Watchdog Timer interrupt
IRQ21	37	DeepSleep	Backup domain interrupt
IRQ22	38	DeepSleep	Other combined Interrupts for SRSS (LVD, WDT, CLKCAL)
IRQ23	39	DeepSleep	CPUSS Inter Process Communication Interrupt #0
IRQ24	40	DeepSleep	CPUSS Inter Process Communication Interrupt #1
IRQ25	41	DeepSleep	CPUSS Inter Process Communication Interrupt #2
IRQ26	42	DeepSleep	CPUSS Inter Process Communication Interrupt #3
IRQ27	43	DeepSleep	CPUSS Inter Process Communication Interrupt #4
IRQ28	44	DeepSleep	CPUSS Inter Process Communication Interrupt #5
IRQ29	45	DeepSleep	CPUSS Inter Process Communication Interrupt #6

Table 8-3. List of PSoC 6 Interrupts

System Interrupt	Cortex M4 Exception Number	Power Mode	Interrupt Source
IRQ30	46	DeepSleep	CPUSS Inter Process Communication Interrupt #7
IRQ31	47	DeepSleep	CPUSS Inter Process Communication Interrupt #8
IRQ32	48	DeepSleep	CPUSS Inter Process Communication Interrupt #9
IRQ33	49	DeepSleep	CPUSS Inter Process Communication Interrupt #10
IRQ34	50	DeepSleep	CPUSS Inter Process Communication Interrupt #11
IRQ35	51	DeepSleep	CPUSS Inter Process Communication Interrupt #12
IRQ36	52	DeepSleep	CPUSS Inter Process Communication Interrupt #13
IRQ37	53	DeepSleep	CPUSS Inter Process Communication Interrupt #14
IRQ38	54	DeepSleep	CPUSS Inter Process Communication Interrupt #15
IRQ39	55	Active	Serial Communication Block #0
IRQ40	56	Active	Serial Communication Block #1
IRQ41	57	Active	Serial Communication Block #2
IRQ42	58	Active	Serial Communication Block #3
IRQ43	59	Active	Serial Communication Block #4
IRQ44	60	Active	Serial Communication Block #5
IRQ45	61	Active	Serial Communication Block #6
IRQ46	62	Active	Serial Communication Block #7
IRQ47	63	Active	Serial Communication Block #9
IRQ48	64	Active	Serial Communication Block #10
IRQ49	65	Active	Serial Communication Block #11
IRQ50	66	Active	Serial Communication Block #12
IRQ51	67	Active	CSD (Capsense) interrupt
IRQ52	68	Active	CPUSS DMAC, Channel #0
IRQ53	69	Active	CPUSS DMAC, Channel #1
IRQ54	70	Active	CPUSS DMAC, Channel #2
IRQ55	71	Active	CPUSS DMAC, Channel #3
IRQ56	72	Active	CPUSS DataWire #0, Channel #0
IRQ57	73	Active	CPUSS DataWire #0, Channel #1
IRQ58	74	Active	CPUSS DataWire #0, Channel #2
IRQ59	75	Active	CPUSS DataWire #0, Channel #3
IRQ60	76	Active	CPUSS DataWire #0, Channel #4
IRQ61	77	Active	CPUSS DataWire #0, Channel #5
IRQ62	78	Active	CPUSS DataWire #0, Channel #6
IRQ63	79	Active	CPUSS DataWire #0, Channel #7
IRQ64	80	Active	CPUSS DataWire #0, Channel #8
IRQ65	81	Active	CPUSS DataWire #0, Channel #9
IRQ66	82	Active	CPUSS DataWire #0, Channel #10
IRQ67	83	Active	CPUSS DataWire #0, Channel #11
IRQ68	84	Active	CPUSS DataWire #0, Channel #12
IRQ69	85	Active	CPUSS DataWire #0, Channel #13
IRQ70	86	Active	CPUSS DataWire #0, Channel #14
IRQ71	87	Active	CPUSS DataWire #0, Channel #15
IRQ72	88	Active	CPUSS DataWire #0, Channel #16

Table 8-3. List of PSoC 6 Interrupts

System Interrupt	Cortex M4 Exception Number	Power Mode	Interrupt Source
IRQ73	89	Active	CPUSS DataWire #0, Channel #17
IRQ74	90	Active	CPUSS DataWire #0, Channel #18
IRQ75	91	Active	CPUSS DataWire #0, Channel #19
IRQ76	92	Active	CPUSS DataWire #0, Channel #20
IRQ77	93	Active	CPUSS DataWire #0, Channel #21
IRQ78	94	Active	CPUSS DataWire #0, Channel #22
IRQ79	95	Active	CPUSS DataWire #0, Channel #23
IRQ80	96	Active	CPUSS DataWire #0, Channel #24
IRQ81	97	Active	CPUSS DataWire #0, Channel #25
IRQ82	98	Active	CPUSS DataWire #0, Channel #26
IRQ83	99	Active	CPUSS DataWire #0, Channel #27
IRQ84	100	Active	CPUSS DataWire #0, Channel #28
IRQ85	101	Active	CPUSS DataWire #1, Channel #0
IRQ86	102	Active	CPUSS DataWire #1, Channel #1
IRQ87	103	Active	CPUSS DataWire #1, Channel #2
IRQ88	104	Active	CPUSS DataWire #1, Channel #3
IRQ89	105	Active	CPUSS DataWire #1, Channel #4
IRQ90	106	Active	CPUSS DataWire #1, Channel #5
IRQ91	107	Active	CPUSS DataWire #1, Channel #6
IRQ92	108	Active	CPUSS DataWire #1, Channel #7
IRQ93	109	Active	CPUSS DataWire #1, Channel #8
IRQ94	110	Active	CPUSS DataWire #1, Channel #9
IRQ95	111	Active	CPUSS DataWire #1, Channel #10
IRQ96	112	Active	CPUSS DataWire #1, Channel #11
IRQ97	113	Active	CPUSS DataWire #1, Channel #12
IRQ98	114	Active	CPUSS DataWire #1, Channel #13
IRQ99	115	Active	CPUSS DataWire #1, Channel #14
IRQ100	116	Active	CPUSS DataWire #1, Channel #15
IRQ101	117	Active	CPUSS DataWire #1, Channel #16
IRQ102	118	Active	CPUSS DataWire #1, Channel #17
IRQ103	119	Active	CPUSS DataWire #1, Channel #18
IRQ104	120	Active	CPUSS DataWire #1, Channel #19
IRQ105	121	Active	CPUSS DataWire #1, Channel #20
IRQ106	122	Active	CPUSS DataWire #1, Channel #21
IRQ107	123	Active	CPUSS DataWire #1, Channel #22
IRQ108	124	Active	CPUSS DataWire #1, Channel #23
IRQ109	125	Active	CPUSS DataWire #1, Channel #24
IRQ110	126	Active	CPUSS DataWire #1, Channel #25
IRQ111	127	Active	CPUSS DataWire #1, Channel #26
IRQ112	128	Active	CPUSS DataWire #1, Channel #27
IRQ113	129	Active	CPUSS DataWire #1, Channel #28
IRQ114	130	Active	CPUSS Fault Structure Interrupt #0
IRQ115	131	Active	CPUSS Fault Structure Interrupt #1

Table 8-3. List of PSoC 6 Interrupts

System Interrupt	Cortex M4 Exception Number	Power Mode	Interrupt Source
IRQ116	132	Active	CRYPTO Accelerator Interrupt
IRQ117	133	Active	FLASH Macro Interrupt
IRQ118	134	Active	Floating Point operation fault
IRQ119	135	Active	CM0+ CTI #0
IRQ120	136	Active	CM0+ CTI #1
IRQ121	137	Active	CM4 CTI #0
IRQ122	138	Active	CM4 CTI #1
IRQ123	139	Active	TCPWM #0, Counter #0
IRQ124	140	Active	TCPWM #0, Counter #1
IRQ125	141	Active	TCPWM #0, Counter #2
IRQ126	142	Active	TCPWM #0, Counter #3
IRQ127	143	Active	TCPWM #0, Counter #4
IRQ128	144	Active	TCPWM #0, Counter #5
IRQ129	145	Active	TCPWM #0, Counter #6
IRQ130	146	Active	TCPWM #0, Counter #7
IRQ131	147	Active	TCPWM #1, Counter #0
IRQ132	148	Active	TCPWM #1, Counter #1
IRQ133	149	Active	TCPWM #1, Counter #2
IRQ134	150	Active	TCPWM #1, Counter #3
IRQ135	151	Active	TCPWM #1, Counter #4
IRQ136	152	Active	TCPWM #1, Counter #5
IRQ137	153	Active	TCPWM #1, Counter #6
IRQ138	154	Active	TCPWM #1, Counter #7
IRQ139	155	Active	TCPWM #1, Counter #8
IRQ140	156	Active	TCPWM #1, Counter #9
IRQ141	157	Active	TCPWM #1, Counter #10
IRQ142	158	Active	TCPWM #1, Counter #11
IRQ143	159	Active	TCPWM #1, Counter #12
IRQ144	160	Active	TCPWM #1, Counter #13
IRQ145	161	Active	TCPWM #1, Counter #14
IRQ146	162	Active	TCPWM #1, Counter #15
IRQ147	163	Active	TCPWM #1, Counter #16
IRQ148	164	Active	TCPWM #1, Counter #17
IRQ149	165	Active	TCPWM #1, Counter #18
IRQ150	166	Active	TCPWM #1, Counter #19
IRQ151	167	Active	TCPWM #1, Counter #20
IRQ152	168	Active	TCPWM #1, Counter #21
IRQ153	169	Active	TCPWM #1, Counter #22
IRQ154	170	Active	TCPWM #1, Counter #23
IRQ155	171	Active	SAR ADC interrupt
IRQ156	172	Active	I2S0 Audio interrupt
IRQ157	173	Active	PDM0/PCM0 Audio interrupt
IRQ158	174	Active	I2S1 Audio interrupt

Table 8-3. List of PSoC 6 Interrupts

System Interrupt	Cortex M4 Exception Number	Power Mode	Interrupt Source
IRQ159	175	Active	Profiler interrupt
IRQ160	176	Active	Serial Memory Interface interrupt
IRQ161	177	Active	USB Interrupt
IRQ162	178	Active	USB Interrupt
IRQ163	179	Active	USB Interrupt
IRQ164	180	Active	SDIO wakeup interrupt for mxsdhc
IRQ165	181	Active	Consolidated interrupt for mxsdhc for everything else
IRQ166	182	Active	EEMC wakeup interrupt for mxsdhc, not used
IRQ167	183	Active	Consolidated interrupt for mxsdhc for everything else

8.6 Interrupt/Exception Priority

Exception priority is useful for exception arbitration when there are multiple exceptions that need to be serviced by the CPU. Both M4 and M0+ cores in PSoC 6 MCUs provide flexibility in choosing priority values for different exceptions. All exceptions other than Reset, NMI, and HardFault can be assigned a configurable priority level. The Reset, NMI, and HardFault exceptions have a fixed priority of –3, –2, and –1, respectively. In PSoC 6 MCUs, lower priority numbers represent higher priorities. This means that the Reset, NMI, and HardFault exceptions have the highest priorities. The other exceptions can be assigned a configurable priority level between 0 and 3 for Cortex-M0+ and 0 to 7 for Cortex-M4.

Both M0+ and M4 support nested exceptions in which a higher priority exception can obstruct (interrupt) the currently active exception handler. This pre-emption does not happen if the incoming exception priority is the same as or lower than the active exception. The CPU resumes execution of the lower priority exception handler after servicing the higher priority exception. The CM0+ core in the PSoC 6 MCU allows nesting of up to four exceptions; the CM4 core allows up to eight exceptions. When the CPU receives two or more exceptions requests of the same priority, the lowest exception number is serviced first.

The registers to configure the priority of exception numbers 1 to 15 are explained in Exception Sources on page 60.

The priority of the 32 CM0+ and 168 CM4 interrupts can be configured by writing to the respective Interrupt Priority registers (CM0P_SCS_IPR and CM4_SCS_IPR). This is a group of eight (CM0+) and 60 (CM4) 32-bit registers with each register storing the priority values of four interrupts, as given in Table 8-4 and Table 8-5. For CM0+, the first 16 bits of the ISER and ICER registers are valid. Refer to Interrupt Sources on page 62 for details on how to map a CPU interrupt to a system interrupt in CM0+.

Table 8-4. Interrupt Priority Register Bit Definitions for Cortex-M0+ (CM0P_SCS_IPR)

Bits	Name	Description
7:6	PRI_N0	Priority of interrupt number N.
15:14	PRI_N1	Priority of interrupt number N+1.
23:22	PRI_N2	Priority of interrupt number N+2.
31:30	PRI_N3	Priority of interrupt number N+3.

Table 8-5. Interrupt Priority Register Bit definitions for Cortex-M4 (CM4_SCS_IPR)

Bits	Name	Description
7:5	PRI_N0	Priority of interrupt number N
15:13	PRI_N1	Priority of interrupt number N+1
23:21	PRI_N2	Priority of interrupt number N+2
31:29	PRI_N3	Priority of interrupt number N+3

8.7 Enabling and Disabling Interrupts

The NVICs of both CM0+ andCM4 core provide registers to individually enable and disable the interrupts in software. If an interrupt is not enabled, the NVIC will not process the interrupt requests on that interrupt line. The Interrupt Set-Enable Register (CM0P_SCS_ISER and CM4_SCS_ISER) and the Interrupt Clear-Enable Register (CM0P_SCS_ICER and CM4_SCS_ICER) are used to enable and disable the interrupts respectively. These registers are 32-bit wide and each bit corresponds to the same numbered interrupt in CM0+. For CM4 core, there are eight ISER/ICER registers. These registers can also be read in software to get the enable status of the interrupts. Table 8-6 shows the register access properties for these two registers. Note that writing zero to these registers has no effect.

For CM0+, the first 16 bits of the ISER and ICER registers are valid. Refer to Interrupt Sources on page 62 for details on how to map a CPU interrupt to a system interrupt in CM0+.

Table 8-6. Interrupt Enable/Disable Registers

Register	Operation	Bit Value	Comment
Interrupt Set Enable Register	Write	1	To enable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled
Interrupt Clear Enable Register	Write	1	To disable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled

The ISER and ICER registers are applicable only for the interrupts. These registers cannot be used to enable or disable the exception numbers 1 to 15. The 15 exceptions have their own support for enabling and disabling, as explained in Exception Sources on page 60.

The PRIMASK register in the CPUs (both CM0+ and CM4) can be used as a global exception enable register to mask all the configurable priority exceptions irrespective of whether they are enabled. Configurable priority exceptions include all the exceptions except Reset, NMI, and HardFault listed in Table 8-1. When the PM bit (bit 0) in the PRIMASK register is set, none of the configurable priority exceptions can be serviced by the CPU, though they can be in the pending state waiting to be serviced by the CPU after the PM bit is cleared.

8.8 Interrupt/Exception States

Each exception can be in one of the following states.

Table 8-7. Exception States

Exception State	Meaning
Inactive	The exception is not active and not pending. Either the exception is disabled or the enabled exception has not been triggered.
Pending	The exception request has been received by the CPU/NVIC and the exception is waiting to be serviced by the CPU.
Active	An exception that is being serviced by the CPU but whose exception handler execution is not yet complete. A high-priority exception can interrupt the execution of lower priority exception. In this case, both the exceptions are in the active state.
Active and Pending	The exception is being serviced by the processor and there is a pending request from the same source during its exception handler execution.

The Interrupt Control State Register (CM0P_SCS_ICSR and CM4_SCS_ICSR) contains status bits describing the various exceptions states.

- The VECTACTIVE bits ([8:0]) in the ICSR store the exception number for the current executing exception. This value is zero if the CPU does not execute any exception handler (CPU is in thread mode). Note that the value in VECTACTIVE bitfields is the same as the value in bits [8:0] of the Interrupt Program Status Register (IPSR), which is also used to store the active exception number.
- The VECTPENDING bits ([20:12]) in the ICSR store the exception number of the highest priority pending exception. This value is zero if there are no pending exceptions.
- The ISRPENDING bit (bit 22) in the ICSR indicates if a NVIC generated interrupt is in a pending state.

8.8.1 Pending Interrupts/Exceptions

When a peripheral generates an interrupt request signal to the NVIC or an exception event occurs, the corresponding exception enters the pending state. When the CPU starts executing the corresponding exception handler routine, the exception is changed from the pending state to the active state. The NVIC allows software pending of the 32 (CM0+) or 168 (CM4) interrupt lines by providing separate register bits for setting and clearing the pending states of the interrupts. The Interrupt Set-Pending register (CM0P_SCS_ISPR and CM4_SCS_ISPR) and the Interrupt Clear-Pending register (CM0P_SCS_ICPR and CM4_SCS_ICPR) are used to set and clear the pending status of the interrupt lines. These registers are 32 bits wide, and each bit corresponds to the same numbered interrupt. In the case of CM4, there are eight sets of such registers to accommodate all 168 interrupts. Table 8-8 shows the register access properties for these two registers. Note that writing zero to these registers has no effect.

Table 8-8. Interrupt Set Pending/Clear Pending Registers

Register	Operation	Bit Value	Comment
Interrupt Set-Pending Register (ISPR)	Write	1	To put an interrupt to pending state
		0	No effect
	Read	1	Interrupt is pending
		0	Interrupt is not pending
Interrupt Clear-Pending Register (ICPR)	Write	1	To clear a pending interrupt
		0	No effect
	Read	1	Interrupt is pending
		0	Interrupt is not pending

Setting the pending bit when the same bit is already set results in only one execution of the ISR. The pending bit can be updated regardless of whether the corresponding interrupt is enabled. If the interrupt is not enabled, the interrupt line will not move to the pending state until it is enabled by writing to the ISER register.

Note that the ISPR and ICPR registers are used only for the peripheral interrupts. These registers cannot be used for pending the exception numbers 1 to 15. These 15 exceptions have their own support for pending, as explained in Exception Sources on page 60.

8.9 Stack Usage for Interrupts/ Exceptions

When the CPU executes the main code (in thread mode) and an exception request occurs, the CPU stores the state of its general-purpose registers in the stack. It then starts executing the corresponding exception handler (in handler mode). The CPU pushes the contents of the eight 32-bit internal registers into the stack. These registers are the Program and Status Register (PSR), ReturnAddress, Link Register (LR or R14), R12, R3, R2, R1, and R0. Both Cortex-M4 and Cortex-M0+ have two stack pointers - MSP and PSP. Only one of the stack pointers can be active at a time. When in thread mode, the Active Stack Pointer bit in the Control register is used to define the current active stack pointer. When in handler mode, the MSP is always used as the stack pointer. The stack pointer always grows downwards and points to the address that has the last pushed data.

When the CPU is in thread mode and an exception request comes, the CPU uses the stack pointer defined in the control register to store the general-purpose register contents. After the stack push operations, the CPU enters handler mode to execute the exception handler. When another higher priority exception occurs while executing the current exception, the MSP is used for stack push/pop operations, because the CPU is already in handler mode. See the CPU Subsystem (CPUSS) chapter on page 32 for details.

8.10 Interrupts and Low-Power Modes

The PSoC 6 MCU family allows device (CPU) wakeup from low-power modes when certain peripheral interrupt requests are generated. The Wakeup Interrupt Controller (WIC) block generates a wakeup signal that causes the CPU to enter Active mode when one or more wakeup sources generate an interrupt signal. After entering Active mode, the ISR of the peripheral interrupt is executed.

The Wait For Interrupt (WFI) or Wait For Event (WFE) instruction, executed by the CPU, triggers the transition into Sleep, and Deep Sleep modes. Both the WFI and WFE instructions are capable of waking up on interrupts. However, the WFE requires the interrupts to be unmasked in the CPU's Priority Mask register. Refer to the PRIMASK register definition on the Arm website. In addition, the WFE instruction puts the CPU to sleep based on the status of an event bit and wakes up from an event signal, typically sent by the other CPU. WFI does not require PRIMASK unmasking and can wake up the CPU from any pending interrupt masked to the NVIC or WIC. However, WFI cannot wake up the CPU from event signals from other CPUs. The sequence of entering the different low-power modes is detailed in the Device Power Modes chapter on page 225.

Chip low-power modes have two categories of interrupt sources:

- Interrupt sources that are available in the Active, Sleep, and Deep Sleep modes (watchdog timer interrupt, RTC, GPIO interrupts, and Low-Power comparators)
- Interrupt sources that are available only in the Active and Sleep modes

When using the WFE instruction in CM4, make sure to call the WFE instruction twice to properly enter and exit Sleep/ Deep Sleep modes. This behavior comes from the event register implementation in Arm v7 architecture used in Cortex-M4. According to the ARM V7 architecture reference manual (Section B1.5.18 Wait For Event and Send Event):

- A reset clears the event register.
- Any WFE wakeup event, or the execution of an exception return instruction, sets the event register.
- A WFE instruction clears the event register.
- Software cannot read or write the value of the event register directly.

Therefore, the first WFE instruction puts CM4 to sleep and second WFE clears the event register after a WFE wakeup, which sets the event register. So the next WFE will put the core to sleep.

Note that this behavior is not present in Arm v6 architecture used in Cortex-M0+. Therefore, in CM0+ only one WFE instruction is sufficient to successfully enter or exit Sleep and Deep Sleep modes.

8.11 Interrupt/Exception – Initialization/ Configuration

This section covers the different steps involved in initializing and configuring exceptions in the PSoC 6 MCU.

1. Configuring the Exception Vector Table Location: The first step in using exceptions is to configure the vector table location as required - either in flash memory or SRAM. This configuration is done as described in Exception Vector Table on page 59.

   The vector table should be available in SRAM if the application must change the vector addresses dynamically. If the table is located in flash, then a flash write operation is required to modify the vector table contents. The ModusToolbox IDE uses the vector table in SRAM by default.
2. Configuring Individual Exceptions: The next step is to configure individual exceptions required in an application, as explained in earlier sections.
   a. Configure the exception or interrupt source; this includes setting up the interrupt generation conditions. The register configuration depends on the specific exception required. Refer to the respective peripheral chapter to know more about the interrupt configuration supported by them.

b. Define the exception handler function and write the address of the function to the exception vector table. Table 8-1 gives the exception vector table format; the exception handler address should be written to the appropriate exception number entry in the table.
c. For Cortex-M0+, define and enable the additional system interrupt handler table and functions as explained in Interrupt Sources on page 62.
d. Set up the exception priority, as explained in Interrupt/Exception Priority on page 69.
e. Enable the exception, as explained in Enabling and Disabling Interrupts on page 69.

8.12 Register List

Table 8-9. Register List

Register Name	Description
CPUSS_CM0_NMI_CTLx	Cortex-M0+ NMI control registers (4 registers)
CPUSS_CM0_SYSTEM_INT_CTLx	Cortex-M0+ interrupt control registers for x^{th} system interrupt (168 registers)
CPUSS_CM0_SYSTEM_INTx_STATUS	Cortex-M0+ interrupt status registers for x^{th} CPU interrupt (8 registers)
CPUSS_CM4_NMI_CTLx	Cortex-M4 NMI control registers (4 registers)
SYSTEM_CM0P_SCS_ISER	Cortex-M0+ interrupt set-enable register
SYSTEM_CM0P_SCS_ICER	Cortex-M0+ interrupt clear enable register
SYSTEM_CM0P_SCS_ISPR	Cortex-M0+ interrupt set-pending register
SYSTEM_CM0P_SCS_ICPR	Cortex-M0+ interrupt clear-pending register
SYSTEM_CM0P_SCS_IPR	Cortex-M0+ interrupt priority register
SYSTEM_CM0P_SCS_ICSR	Cortex-M0+ interrupt control state register
SYSTEM_CM0P_SCS_VTOR	Cortex-M0+ vector table offset register
SYSTEM_CM0P_SCS_AIRCR	Cortex-M0+ application interrupt and reset control register
SYSTEM_CM0P_SCS_SHPR2	Cortex-M0+ system handler priority register 2
SYSTEM_CM0P_SCS_SHPR3	Cortex-M0+ system handler priority register 3
SYSTEM_CM0P_SCS_SHCSR	Cortex-M0+ system handler control and state register
SYSTEM_CM4_SCS_ISER	Cortex-M4 interrupt set-enable register
SYSTEM_CM4_SCS_ICER	Cortex-M4 interrupt clear enable register
SYSTEM_CM4_SCS_ISPR	Cortex-M4 interrupt set-pending register
SYSTEM_CM4_SCS_ICPR	Cortex-M4 interrupt clear-pending register
SYSTEM_CM4_SCS_IPR	Cortex-M4 interrupt priority registers
SYSTEM_CM4_SCS_ICSR	Cortex-M4 interrupt control state register
SYSTEM_CM4_SCS_VTOR	Cortex-M4 vector table offset register
SYSTEM_CM4_SCS_AIRCR	Cortex-M4 application interrupt and reset control register
SYSTEM_CM4_SCS_SHPR2	Cortex-M4 system handler priority register 2
SYSTEM_CM4_SCS_SHPR3	Cortex-M4 system handler priority register 3
SYSTEM_CM4_SCS_SHCSR	Cortex-M4 system handler control and state register

9. Protection Units

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

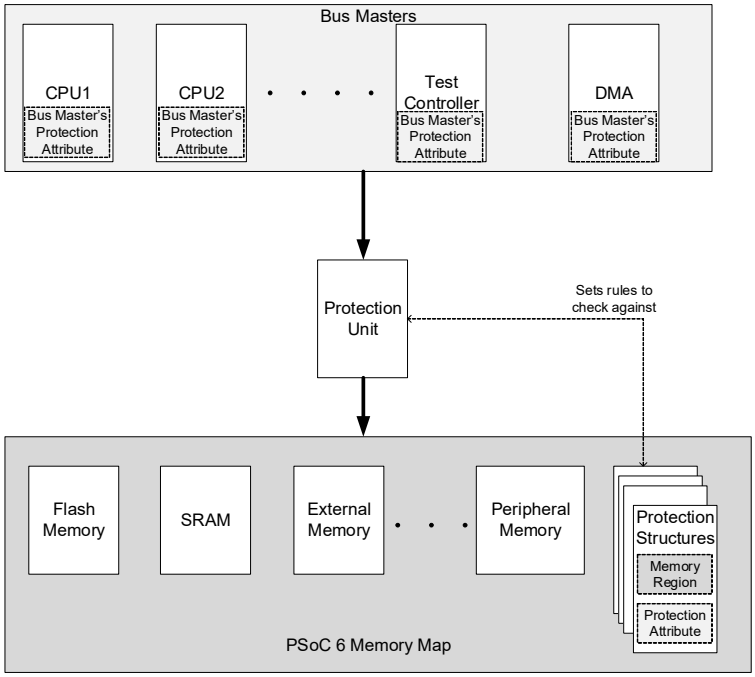
- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

Protection units are implemented in the PSoC 6 MCU to enforce security based on different operations. A protection unit allows or restricts bus transfers. The rules are enforced based on specific properties of a transfer. The rules that determine protection are implemented in protection structures (a register structure). A protection structure defines the protected address space and the protection attributes. The hardware that evaluates these protection structures, to restrict or permit access, is the protection unit. The PSoC device has different types of protection units such as MPU, SMPU, and PPU. Each have a distinct set of protection structures, which helps define different protection regions and their attributes.

9.1 Architecture

Figure 9-1 shows a conceptual view of implementation of the PSoC protection system.

Figure 9-1. Conceptual View of PSoC Protection System



The functioning of a secure system is based on the following:

- Bus masters: This term refers to the bus masters in the architecture. In a PSoC 6 device, an example of a bus master is a Cortex-M core, DMA, or a test controller.
- Protection units: Protection units are the hardware engines that enforce the protection defined by protection structures. There are three types of protection units, acting at different levels of memory access with different precedence and priority of protection – MPU, SMPU, and PPU.
- Protection structure: A protection structure is a register structure in memory that sets up the rules based on which each protection unit will evaluate a transfer. Each protection unit associates itself to multiple protection structures. The protection structure associated with a protection unit are evaluated in the order starting with the protection structure with the largest index. For example, if there are 16 protection structures associated with a protection unit, then the evaluation of a transfer starts from protection structure 15 and counts down. Physically a protection structure is a register structure in the memory map that defines a protection rule. Each protection structure constitutes the following:
  - Defines a memory region on which the rule is applied. It designates what the bus transfer needs to be evaluated against this protection structure.
    - Base address
    - Size of memory block
  - A set of protection attributes
    - R/W/X
    - User/privilege
    - Secure/non Secure
    - Protection context
- Protection attributes: These are properties based on which a transfer is evaluated. There are multiple protection attributes. The set of protection attributes available for a protection structure depends on the protection unit it is associated with. Protection attributes appear in two places:
  - Protection structures: Protection attributes associated with a protection structure set the rules for access based on these attributes.
  - Bus master's protection attribute: Each bus master has its own access attributes, which define the bus master's access privileges. Some of these attributes, such as secure/non-secure, are set for a master. Other attributes such as protection context and user/privilege attribute are dynamic attributes, which change based on bus master's context and state.

In summary, a PSoC 6 device has protection units that act as a gate for any access to the PSoC memory map. The rules for protection are set by the protection structures. Each bus master is qualified by its own protection attribute. For every bus transfer, the protection unit compares the bus master's protection attribute and accessed address against the rules set in the protection structures and decides on providing or denying access.

9.2 PSoC 6 Protection Architecture

When there is a memory (SRAM/flash/peripheral) access by a bus master, the access is evaluated by a protection unit against the protection attributes set in protection structures for the memory location being accessed. If the bus master’s protection attributes satisfy the protection attributes set in the protection structures, then access is allowed by the protection unit. If there is an access restriction, a fault condition is triggered and a bus error occurs. Thus protection units secure bus transfer address range either in memory locations (SRAM/flash) or peripheral registers. From an architectural perspective, there is no difference between memory protection and peripheral protection. However, from an implementation perspective, separate memory and peripheral protection is provided.

Two types of protection units, memory protection units (MPU) and shared memory protection units (SMPU), are provided in the CPU subsystem (CPUSS) to protect memory locations. A separate protection unit type is provided for peripheral protection (PPU) in the PERI:

- A bus master may have a dedicated MPU. In a CPU bus master, the MPU is typically implemented as part of the CPU and is under control of the OS/kernel. In a non-CPU bus master, the MPU is typically implemented as part of the bus infrastructure and under control of the OS/kernel of the CPU that “owns or uses” the bus master. If a CPU switches tasks or if a non-CPU switches ownership, the MPU settings are typically updated by OS/kernel software. The different MPU types are:
  - An MPU that is implemented as part of the CPU. This type is found in the Arm CM0+ and CM4 CPUs.
  - An MPU that is implemented as part of the bus infrastructure. This type is found in bus masters such as crypto and test controller. The definition of this MPU type follows the Arm MPU definition (in terms of memory region and access attribute definition) to ensure a consistent software interface.
- SMPUs are intended for implementing protection in a situation with multiple bus masters. These protection units implement a concept called Protection Context. A protection context is a pseudo state of a bus master, which can be used to determine access attributes across multiple masters. The protection context is a protection attribute not specific to a bus master. The SMPUs can distinguish between different protection contexts; they can also distinguish secure from non-secure accesses.

This allows for an effective protection in a multi-core scenario.

- PPUs are protection units provided in the PERI register space for peripheral protection. The PPU attributes are similar to the SMPU, except that they are intended for protecting the peripheral space. Refer to the registers TRM for details. The PPUs are intended to distinguish between different protection contexts and to distinguish secure from non-secure accesses and user mode accesses from privileged mode accesses. There are two types of PPU structures.
  - Fixed PPUs implement protection for fixed address regions that typically correspond to a specific peripheral
  - Programmable PPUs allows the user to program the address region to be protected

The platform's DMA controller does not have an MPU. Instead, a DMA controller channel inherits the access control attributes of the bus transfer that programmed the channel.

The definition of SMPU and PPU follows the MPU definition and adds the capability to distinguish accesses from different protection contexts (the MPU does not include support for a protection context). If security is required, the SMPU and possibly PPUs MMIO registers must be controlled by a secure CPU that enforces system-wide protection.

Figure 9-2 gives an overview of the location of MPUs, SMPUs, and PPUs in the system. Note that a peripheral group PPU needs to provide access control only to the peripherals within a peripheral group (group of peripherals with a shared bus infrastructure).

As mentioned, the MPU, SMPU, and PPU protection functionality follows the Arm MPU definition:

- Multiple protection structures are supported.
- Each structure specifies an address range in the unified memory architecture and access attributes. An address range can be as small as 32 bytes.

A protection violation is caused by a mismatch between a bus master's access attributes and the protection structure and access attributes for the memory region configured in the protection structure.

A bus transfer that violates a protection structure results in a bus error.

For AXI transfers, the complete address range is matched. If a transfer references multiple 32-byte regions (the smallest protection structure address range is 32 bytes), multiple cycles are required for matching – one cycle per 32-byte region.

Protection violations are captured in the fault report structure to allow for failure analysis. The fault report structures can generate an interrupt to indicate the occurrence of a fault. This is useful if the violating bus master cannot resolve the bus error by itself, but requires another CPU bus master to resolve the bus error on its behalf.

For a buffered mode of transfer (CPUSS_BUFF_CTL[WRITE_BUFF]), the behavior during protection violation is different. When CPUSS_BUFF_CTL[WRITE_BUFF] is set to '1', the write transfers on the bus are buffered. So the transfer is first acknowledged when the buffer receives the transfer. A protection violation will be only evaluated when the actual write happens at the destination register. This leads to the write transfer not generating a bus error for buffered mode. However, a fault will be registered as soon as the transfer tries to write the destination location. Therefore, for buffered writes, the user must verify the fault structure to make sure no violations have occurred.

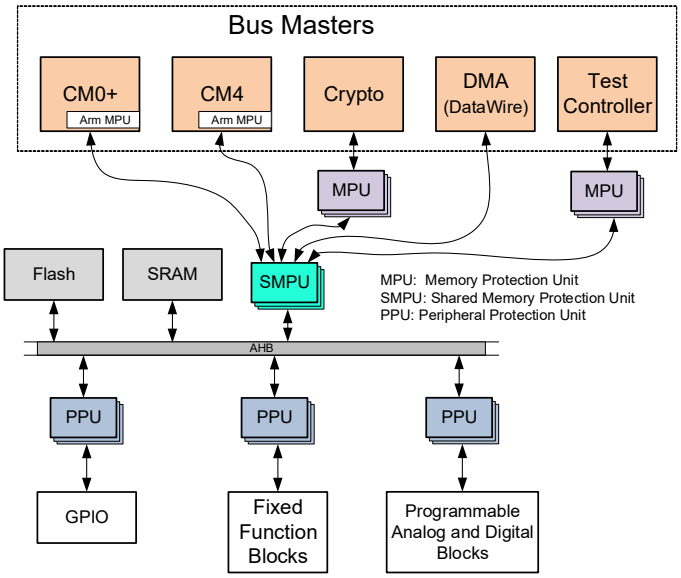
A protection violation results in a bus error and the bus transfer will not reach its target. An MPU or SMPU violation that targets a peripheral will not reach the associated protection evaluation (PPU). In other words, MPU and SMPU have a higher priority over PPU.

Protection unit addresses the following:

- Security requirements. This includes prevention of malicious attacks to access secure memory or peripherals. For example, a non-secure master should not be able to access key information in a secure memory region.
- Safety requirements. This includes detection of accidental (non-malicious) software errors and random hardware errors. Enabling failure analysis is important so the root cause of a safety violation can be investigated. For example, analyzing a flash memory failure on a device that is returned from the field should be possible.

To address security requirements, the Cortex M0+ is used as a 'secure CPU'. This CPU is considered a trusted entity. Any access by the CPU tagged as "secure" will be called "secure access".

Figure 9-2. PSoC 6 Protection Architecture



The different types of protection units cater to different use cases for protection.

Table 9-1. Protection Use Cases

Protection Unit Type	Use
Arm MPU	Used to protect memory between tasks within in a single Arm core. A task in one of the Arm cores can protect its memory from access by another task in the same core.
MPU	Same as the Arm MPU, but for other bus masters such as the test controller or crypto, which do not have a built-in MPU in their block IP.
SMPU	Used to protect memory addresses that are shared between multiple bus masters.
Fixed PPU protection structures	These protect specific peripheral memory space. The protection structures have a preprogrammed memory region and can be used only to protect the peripheral it was intended for.
Programmable PPU protection structures	These protect the peripheral space but the memory region is not fixed. So users can easily program it to protect any space in the peripheral memory region.

9.3 Register Architecture

The protection architecture has different conceptual pieces and different sets of registers correspond to each of these concepts.

9.3.1 Protection Structure and Attributes

The MPU, SMPU, and PPU protection structure definition follows the Arm definition. Each protection structure is defined by:

- An address region
- Access control attributes

A protection structure is always aligned on a 32-byte boundary in the memory space. Two registers define a protection structure: ADDR (address register) and ATT (attribute register). This structure alignment and organization allow straightforward protection of the protection structures by the protection scheme. This is discussed later in this chapter.

Address region: The address region is defined by:

- The base address of a region as specified by ADDR.ADDR.
- The size of a region as specified by ATT.REGION_SIZE.
- Individual disables for eight subregions within the region, as specified by ADDR.SUBREGION_DISABLE.

The REGION_SIZE field specifies the size of a region. The region size is a power of 2 in the range of [256 B, 4 GB]. The base address ADDR specifies the start of the region, which must be aligned to the region size. A region is partitioned into eight equally sized sub-regions. The SUBREGION_DISABLE field specifies individual enables for the sub-regions within a region. For example, a REGION_SIZE of “0x08” specifies a region size of 512 bytes. If the start address is 0x1000:5400 (512-byte aligned), the region ranges from 0x1000:5400 to 0x1000:55ff. This region is partitioned into the following eight 64-byte subregions:

subregion 0 from 0x1000:5400 to 0x1000:543f

subregion 1 from 0x1000:5440 to 0x1000:547f

…

subregion 7 from 0x1000:55c0 to 0x1000:55ff.

If the SUBREGION_DISABLE is 0x82 (bitfields 1 and 7 are ‘1’), subregions 1 and 7 are disabled; subregions 0, 2, 3, 4, 5, and 6 are enabled.

In addition, an ATT.ENABLED field specifies whether the region is enabled. Only enabled regions participate in the protection “matching” process. Matching identifies if a bus transfer address is contained within an enabled subregion (SUBREGION_DISABLE) of an enabled region (ENABLED).

Protection attributes: The protection attributes specify access control to the region (shared by all subregions within the region). Access control is performed by comparing against a bus master's protection attributes of the bus master performing the transfer. The following access control fields are supported:

- Control for read accesses in user mode (ATT.UR field).
- Control for write accesses in user mode (ATT.UW field).
- Control for execute accesses in user mode (ATT.UX field).
- Control for read accesses in privileged mode (ATT.PR field).
- Control for write accesses in privileged mode (ATT.PW field).
- Control for execute accesses in privileged mode (ATT.PX field).
- Control for secure access (ATT.NS field).
- Control for individual protection contexts (ATT.PC_MASK[15:0], with MASK[0] always constant at 1). This protection context control field is present only for the SMPU and PPU.

The execute and read access control attributes are orthogonal. Execute transfers are typically read transfers. To allow execute and read transfers in user mode, both ATT.UR and ATT.UX must be set to ‘1’. To allow data and read transfers in user mode, only ATT.UR must be set to ‘1’. In addition, the ATT.PC_MATCH control field is supported, which controls “matching” and “access evaluation” processes. This control field is present only for the SMPU and PPU protection structures.

For example, only protection context 2 can access a specific address range. These accesses are restricted to read and write secure accesses in privileged mode. The access control fields are programmed as follows:

- ATT.UR is 0: read accesses in user mode not allowed.
- ATT.UW is 0: write accesses in user mode not allowed.
- ATT.UX is 0: execute accesses in user mode not allowed.
- ATT.PR is 1: read accesses in privileged mode allowed.
- ATT.PW is 1: write accesses in privileged mode allowed.
- ATT.PX is 0: execute accesses in privileged mode not allowed.
- ATT.NS is 0: secure access required.
- ATT.PC_MASK is 0x0005: protection context 1 and 3 accesses enabled (all other protection contexts are disabled).
- ATT.PC_MATCH is 0: the PC_MASK field is used for access evaluation. Three separate access evaluation subprocesses are distinguished:
  - A subprocess that evaluates the access based on read/write, execute, and user/privileged access attributes.
  - A subprocess that evaluates the access based on the secure/non-secure attribute.
  - A subprocess that evaluates the access based on the protection context index (used only by the SMPU and PPU when ATT.PC_MATCH is 0).

If all access evaluations are successful, access is allowed. If any process evaluation is unsuccessful, access is not allowed. Matching the bus transfer address and access evaluation of the bus transfer (based on access attributes) are two independent processes:

- Matching process. For each protection structure, the process identifies whether a transfer address is contained within the address range. This identifies the “matching” regions.
- Access evaluation process. For each protection structure, the process evaluates the bus transfer access attributes against the access control attributes.

A protection unit typically has multiple protection structures and evaluates the protection structures in decreasing order. The first matching structure provides the access control attributes for the evaluation of the transfer's access attributes. In other words, higher-indexed structures take precedence over lower-indexed structures.

The following pseudo code illustrates the process.

```
match = 0;
for (i = n-1; i >= 0; i--)// n: number of protection regions
   if (Match (“transfer address”, “protection context”
              “MMIO registers ADDR and ATT of protection structure i”)) {
      match = 1; break;
   }
|

if (match)
   AccessEvaluate (“transfer access attributes”, “protection context”
                   “MMIO register ATT of protection structure i”);
else
   “access allowed”
```

Notes:

- If no protection structure provides a match, the access is allowed.
- If multiple protection structures provide a match, the access control attributes for the access evaluation are provided by the protection structure with the highest index.

An example of using the PC_MATCH feature is as follows. Two SMPU structures are configured to protect the same address range:

- Case 1: SMPU#2: PC = 3, PC_MATCH = 0 SMPU#1: PC = 2, PC_MATCH = 0

  To access the master of protection context 2, SMPU#2 has the highest index and address match, but attributes do not match; therefore, access is restricted. The SMPU#1 is not evaluated because the PC_MATCH is 0.
- Case 2: SMPU#2: PC = 3, PC_MATCH = 1 SMPU#1: PC = 2, PC_MATCH = 0

  The SMPU#2 address matches but PC does not match and is skipped because PC_MATCH is 1. SMPU#1 is evaluated and the address and attributes match; therefore, access is allowed.

As mentioned, the protection unit evaluates the protection structures in decreasing order. From a security requirements perspective, this is of importance: a non-secure protection context must not be able to add protection structures that have a higher index than the protection structures that provide secure access. The protection structure with a higher index can be programmed to allow non-secure accesses. Therefore, in a secure system, the higher programmable protection structures are protected to only allow restricted accesses

9.4 Bus Master Protection Attributes

The protection structures set up the rules for different memory regions and their access attributes. The bus master's own protection attributes are used by the protection units to regulate access, based on rules set by the protection structures. Not all bus masters provide all protection attributes that are associated with a bus transfer. Some examples are:

- None of the bus masters has a native protection context attribute. This must be set dynamically based on the task being executed by the bus master.
- The Arm Cortex M4 and Arm Cortex M0+ CPUs provide a user/privilege attribute, but do not provide a secure/non-secure attribute natively. This must be set at a system level.

To ensure system-wide restricted access, missing attributes are provided by register fields. These fields may be set during the boot process or by the secure CPU.

- The SMPU MS_CTL.PC_MASK[] and MPU MS_CTL.PC[] register fields provide protection context functionality.
- The SMPU MS_CTL.P register field provides the user/privileged attribute for those masters that do not provide their own attribute.
- The SMPU MS_CTL.NS register provides the secure/non-secure attribute for those masters that do not provide their own attribute.
- Masters that do not provide an execute attribute have the execute attribute set to '0'.

The DMA controller channels inherit the access control attributes of the bus transfers that configured the DMA channel.

- All the bus masters in the system have SMPU and MPU MS_CTL registers associated with them.
- The MPU MS_CTL.PC_SAVED field (and associated protection context 0 functionality, which is discussed later in the chapter) is only present for the CM0+ master.
- The SMPU MS_CTL.P, MS_CTL.NS, and MS_CTL.PC_MASK fields are not present for the DMA. The bus transfer attributes are provided through "inheritance": the bus transfer attributes are from the master that owns the DMA channel that initiated the bus transfer.
- The MPU MS_CTL register is not present for the DMA masters. The protection context (PC) bus transfer attribute is provided through inheritance.

9.5 Protection Context

Each bus master has a MPU MS_CTL.PC[3:0] protection context field. This protection context is used as the protection context attribute for all bus transfers that are initiated by the master. The SMPUs and PPUs allow or restrict bus transfers based on the protection context attribute.

Multiple masters can share a protection context. For example, a CPU and a crypto controlled by the CPU may share a protection context (the CPU and crypto PC[] fields are the same). Therefore, the CPU and crypto share the SMPU and PPU access restrictions.

A bus master protection context is changed by reprogramming the master's PC[] field. Changing a protection context is required for CPU bus masters that may transition between multiple tasks, each catering to different protection contexts. As the protection context allows or restricts bus transfers, changes to the protection context should be controlled and should not compromise security. Furthermore, changes to the protection context should incur limited CPU overhead to allow for frequent protection context changes. Consider a case in which a CPU executes two software stacks with different protection contexts. To this end, each bus master has an SMPU MS_CTL.PC_MASK[15:0] protection context mask field that identifies what protection contexts can be programmed for the bus master:

- The protection context field MS_CTL.PC[3:0]. This register is controlled by the associated bus master and has the same access restrictions as the bus master's MPU registers.
- The protection context mask field MS_CTL.PC_MASK[15:0]. This register is controlled by the secure CPU and has the same access restrictions as the SMPU registers.

The PC_MASK[] field is a "hot-one" field that specifies whether the PC[] field can be programmed with a specific protection context. Consider an attempt to program PC[] to '3':

- If PC_MASK[3] is '1', PC[] is set to "3".
- If PC_MASK[3] is '0', PC[] is not changed.

9.6 Protection Contexts 0, 1, 2, 3

The PSoC 6 MCU supports protection contexts to isolate software execution for security and safety purposes. Protection contexts are used to restrict access to memory and peripheral resources.

Out of a maximum of 16 protections contexts (PCs), four PCs are treated special: the entry to special PCs 0, 1, 2, and 3 is hardware-controlled. For each PC i, a programmable exception handler address is provided (CM0_PCi_HANDLER.ADDR [31:0]). A CPU exception handler fetch, which returns a handler address that matches the programmed CM0_PCi_HANDLER.ADDR[31:0] address value, causes the CM0+ PC to be changed to PC i by hardware. However, if the current PC is already 0, 1, 2, or 3, the current PC is not changed (an attempt to change the PC results in an AHB-Lite bus error). This ensures that CPU execution in PC 0, 1, 2, or 3 cannot be interrupted/preempted by CPU execution in another PC 0, 1, 2, or 3. In other words, CPU execution in PC 0, 1, 2, or 3 requires cooperative multi-tasking between the different PCs. This means that hand-overs between different PCs are software-scheduled/controlled. A security implementation requires PC software to clean/zeroize information that it wants to keep confidential from other PC software.

The following pseudo code gives a description of the hardware control over the CM0+ current PC “pc”:

```
if      (CM0_PC_CTL.VALID[0] & (addr == CM0_PC0_HANDLER. ADDR)) {match_new, pc_new} = {1, 0};
else if (CM0_PC_CTL.VALID[1] & (addr == CM0_PC1_HANDLER.ADDR)) {match_new, pc_new} = {1, 1};
else if (CM0_PC_CTL.VALID[2] & (addr == CM0_PC2_HANDLER.ADDR)) {match_new, pc_new} = {1, 2};
else if (CM0_PC_CTL.VALID[3] & (addr == CM0_PC3_HANDLER.ADDR)) {match_new, pc_new} = {1, 3};
else                                                            match_new          =  0;

if ("exception handler fetch")
begin
   if (match_new)
   begin
      if      (pc == pc_new)         ;
      else if (CM0_PC_CTL.VALID[pc]) "AHB-Lite bus error";
      else                           {pc, pc_saved} = {pc_new,   pc};     // "push"
   end
   else
   begin
                                     {pc, pc_saved} = {pc_saved, pc_saved};   // "pop"
   end
end
```

Note that each of the protection special PCs 0, 1, 2, and 3 have a dedicated CM0_PC_CTL.VALID[i] field to specify that the PC's exception handler address is provided through CM0_PCi_HANDLER.ADDR[31:0]. If a PC's exception handler address is not provided, the PC is treated as an ordinary PC (PCs 4, 5, ..., 15). Note that the current PC “pc” and a saved PC “pc_saved” implement a two-entry stack. Hardware pushes the current PC to the stack upon entry of a special exception handler. Hardware then pushes the saved PC from the stack upon entry of an ordinary exception handler. An attempt to enter a special exception handler from a special exception handler with a different PC results in an AHB-Lite bus error (which causes the CPU to enter the bus fault exception handler). This scenario should not occur in a carefully designed cooperative multi-tasking software implementation.

Of the four special PCs, PC 0 is treated differently:

- It is the default PC value after a Deep Sleep reset.
- It has unrestricted access; that is, it is not affected by the CPUSS and PERI protection schemes. Therefore, the Cypress boot code software always starts execution in PC 0. The boot code software initializes the protection structures and initializes the CM0_PCi_HANDLER registers. The boot code is considered “trusted” software and its unrestricted access in PC 0 is not considered a protection concern. After initialization of the protection information, the access to the protection information itself is typically restricted for all other PCs (the boot code software deploys the restrictions) and the protection information provides specific restricted access to the other special PCs and ordinary PCs (1, 2, ..., 15). As part of the boot process configuration, the following things happen:
  - The boot process sets MPUn.MS_CTL.PC_SAVED to a context that is not a special. context. This is required to prevent accidental/malicious incorrect entry into a special context.
  - If the system configuration does not use PC=0 after boot is complete, the boot process disables all future use of PC=0 by making ineffective CPUSS.CM0_PC0_HANDLER by clearing CPUSS.CM0_PC_CTL.VALID[0]. Furthermore, the

protection region CPUSS.BOOT is assigned to be accessible from PC=0 only, which ensures these settings can no longer be changed after boot.

- The boot process then changes the current PC by changing MPUn.MS_CTL.PC to a protection context that is not a special context. This is done for all currently active masters (which typically is only the CM0+).
- When the above is complete, the system executes a regular (not special) context and can enter into special contexts only through the use of interrupts using the special handler addresses.

9.7 Protection Structure

9.7.1 Protection Violation

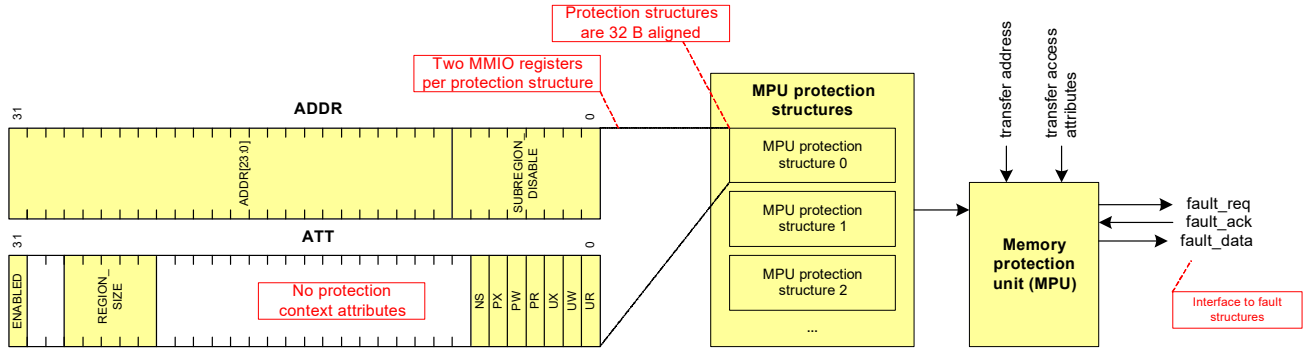
If an MPU, SMPU, or PPU detects a not-allowed transfer, the bus transfer results in a bus error. The bus transfer does not reach its target memory location or peripheral register. In addition, information on the violating bus transfer is communicated to the fault report structure.

9.7.2 MPU

The MPUs are situated in the CPUSS and are associated to a single master. An MPU distinguishes user and privileged accesses from a single bus master. However, the capability exists to perform access control on the secure/non-secure attribute.

As an MPU is associated to a single master, the MPU protection structures do not provide protection context control attributes.

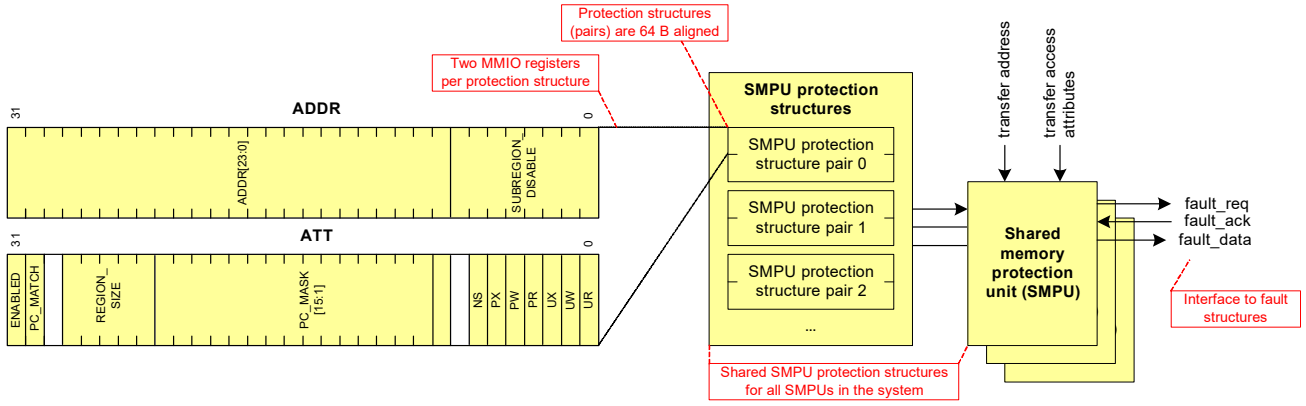
Figure 9-3. MPU Functionality



9.7.3 SMPU

The SMPU is situated in the CPUSS and is shared by all bus masters. The SMPU distinguishes between different protection contexts and distinguishes secure from non-secure accesses. However, the capability exists to perform access control on the user/privileged mode attribute.

Figure 9-4. SMPU Functionality

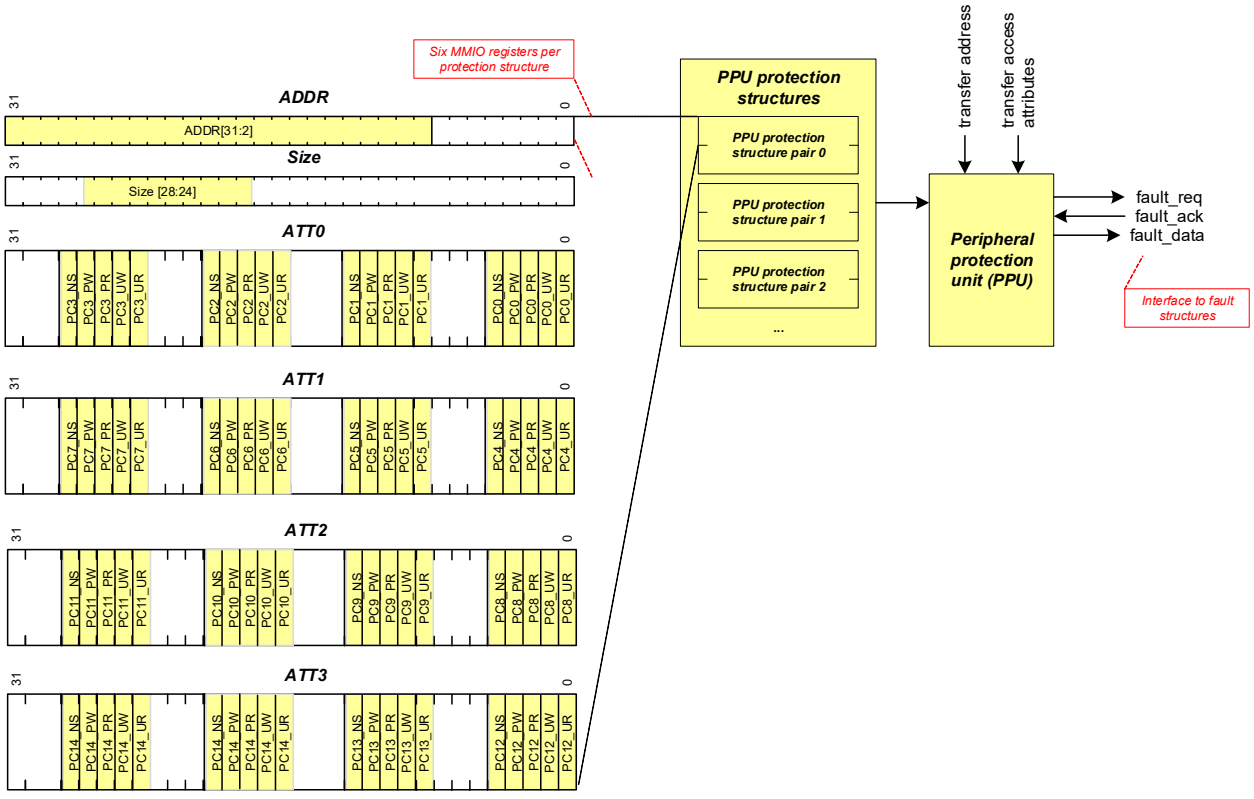


Note that a single set of SMPU region structures provides the same protection information to all SMPUs in the systems.

9.7.4 PPU

- The PPUs are situated in the PERI block and are associated with a peripheral group (a group of peripherals with a shared AHB-Lite bus infrastructure). A PPU is shared by all bus masters. The PPU distinguishes between different protection contexts; it also distinguishes secure from non-secure accesses and user mode from privileged mode accesses.

Figure 9-5. PPU Functionality



There are two types of PPU structures: fixed and programmable.

- The fixed PPU structures protect fixed areas of memory and hence a specific predetermined peripheral region. In other words, the ADDR, SUBREGION_DISABLE, and REGION_SIZE fields are fixed for a specific device. Refer to the registers TRM for a definition of fixed PPUs and the address regions they protect. Their protection attributes can be

configured for each protection context. The following table summarizes the different fixed PPU structures in CY8C62xx devices and details the regions they protect.

Table 9-2. Fixed PPU Structures

Protection Structure	Base Address of Protected Region	Size of Protected Region	Protected Peripheral/Block(s)
PERI_MS_PPU_FX0	0x40000000	8 KB	PERI
PERI_MS_PPU_FX1	0x40004010	4 B	PERI_GR0
PERI_MS_PPU_FX2	0x40004030	4 B	PERI_GR1
PERI_MS_PPU_FX3	0x40004050	4 B	PERI_GR2
PERI_MS_PPU_FX4	0x40004060	32 B	PERI_GR3
PERI_MS_PPU_FX5	0x40004080	32 B	PERI_GR4
PERI_MS_PPU_FX6	0x400040C0	32 B	PERI_GR6
PERI_MS_PPU_FX7	0x40004120	32 B	PERI_GR9
PERI_MS_PPU_FX8	0x40004140	32 B	PERI_GR10
PERI_MS_PPU_FX9	0x40008000	32 KB	PERI_TR
PERI_MS_PPU_FX10	0x40100000	1 KB	CRYPTO
PERI_MS_PPU_FX11	0x40101000	2 KB	CRYPTO
PERI_MS_PPU_FX12	0x40102000	256 B	CRYPTO
PERI_MS_PPU_FX13	0x40102100	4 B	CRYPTO
PERI_MS_PPU_FX14	0x40102120	4 B	CRYPTO
PERI_MS_PPU_FX15	0x40108000	4 KB	CRYPTO
PERI_MS_PPU_FX16	0x40200000	1 KB	CPUSS
PERI_MS_PPU_FX17	0x40201000	4 KB	CPUSS
PERI_MS_PPU_FX18	0x40202000	512 B	CPUSS
PERI_MS_PPU_FX19	0x40208000	1 KB	CPUSS_CM0_SYSTEM_INT
PERI_MS_PPU_FX20	0x4020A000	1 KB	CPUSS_CM4_SYSTEM_INT
PERI_MS_PPU_FX21	0x40210000	256 B	FAULT_STRUCT0
PERI_MS_PPU_FX22	0x40210100	256 B	FAULT_STRUCT1
PERI_MS_PPU_FX23	0x40220000	32 B	IPC_STRUCT0
PERI_MS_PPU_FX24	0x40220020	32 B	IPC_STRUCT1
PERI_MS_PPU_FX25	0x40220040	32 B	IPC_STRUCT2
PERI_MS_PPU_FX26	0x40220060	32 B	IPC_STRUCT3
PERI_MS_PPU_FX27	0x40220080	32 B	IPC_STRUCT4
PERI_MS_PPU_FX28	0x402200A0	32 B	IPC_STRUCT5
PERI_MS_PPU_FX29	0x402200C0	32 B	IPC_STRUCT6
PERI_MS_PPU_FX30	0x402200E0	32 B	IPC_STRUCT7
PERI_MS_PPU_FX31	0x40220100	32 B	IPC_STRUCT8
PERI_MS_PPU_FX32	0x40220120	32 B	IPC_STRUCT9
PERI_MS_PPU_FX33	0x40220140	32 B	IPC_STRUCT10
PERI_MS_PPU_FX34	0x40220160	32 B	IPC_STRUCT11
PERI_MS_PPU_FX35	0x40220180	32 B	IPC_STRUCT12
PERI_MS_PPU_FX36	0x402201A0	32 B	IPC_STRUCT13
PERI_MS_PPU_FX37	0x402201C0	32 B	IPC_STRUCT14
PERI_MS_PPU_FX38	0x402201E0	32 B	IPC_STRUCT15
PERI_MS_PPU_FX39	0x40221000	16 B	IPC_INTR_STRUCT0
PERI_MS_PPU_FX40	0x40221020	16 B	IPC_INTR_STRUCT1

Table 9-2. Fixed PPU Structures

Protection Structure	Base Address of Protected Region	Size of Protected Region	Protected Peripheral/Block(s)
PERI_MS_PPU_FX41	0x40221040	16 B	IPC_INTR_STRUCT2
PERI_MS_PPU_FX42	0x40221060	16 B	IPC_INTR_STRUCT3
PERI_MS_PPU_FX43	0x40221080	16 B	IPC_INTR_STRUCT4
PERI_MS_PPU_FX44	0x402210A0	16 B	IPC_INTR_STRUCT5
PERI_MS_PPU_FX45	0x402210C0	16 B	IPC_INTR_STRUCT6
PERI_MS_PPU_FX46	0x402210E0	16 B	IPC_INTR_STRUCT7
PERI_MS_PPU_FX47	0x40221100	16 B	IPC_INTR_STRUCT8
PERI_MS_PPU_FX48	0x40221120	16 B	IPC_INTR_STRUCT9
PERI_MS_PPU_FX49	0x40221140	16 B	IPC_INTR_STRUCT10
PERI_MS_PPU_FX50	0x40221160	16 B	IPC_INTR_STRUCT11
PERI_MS_PPU_FX51	0x40221180	16 B	IPC_INTR_STRUCT12
PERI_MS_PPU_FX52	0x402211A0	16 B	IPC_INTR_STRUCT13
PERI_MS_PPU_FX53	0x402211C0	16 B	IPC_INTR_STRUCT14
PERI_MS_PPU_FX54	0x402211E0	16 B	IPC_INTR_STRUCT15
PERI_MS_PPU_FX55	0x40230000	64 B	PROT_SMPU
PERI_MS_PPU_FX56	0x40234000	4 B	PROT_MPU0
PERI_MS_PPU_FX57	0x40235400	1 KB	PROT_MPU5
PERI_MS_PPU_FX58	0x40235800	1 KB	PROT_MPU6
PERI_MS_PPU_FX59	0x40237800	4 B	PROT_MPU14
PERI_MS_PPU_FX60	0x40237C00	1 KB	PROT_MPU15
PERI_MS_PPU_FX61	0x40240000	8 B	FLASHC_FLASH
PERI_MS_PPU_FX62	0x40240008	4 B	FLASHC_FLASH
PERI_MS_PPU_FX63	0x40240200	256 B	FLASHC
PERI_MS_PPU_FX64	0x40240400	128 B	FLASHC_CM0
PERI_MS_PPU_FX65	0x40240480	128 B	FLASHC_CM4
PERI_MS_PPU_FX66	0x40240500	4 B	FLASHC_CRYPTO
PERI_MS_PPU_FX67	0x40240580	4 B	FLASHC_DW0
PERI_MS_PPU_FX68	0x40240600	4 B	FLASHC_DW1
PERI_MS_PPU_FX69	0x40240680	4 B	FLASHC_DMAC
PERI_MS_PPU_FX70	0x40240700	4 B	FLASHC_EXT_MS0
PERI_MS_PPU_FX71	0x40240780	4 B	FLASHC_EXT_MS1
PERI_MS_PPU_FX72	0x4024F000	4 KB	FLASHC_FM
PERI_MS_PPU_FX73	0x40260000	256 B	PWR
PERI_MS_PPU_FX74	0x40260100	16 B	
PERI_MS_PPU_FX75	0x40260180	16 B	WDT
PERI_MS_PPU_FX76	0x40260200	128 B	MCWDT
PERI_MS_PPU_FX77	0x40260300	256 B	CLK
PERI_MS_PPU_FX78	0x40260400	1 KB	CLK
PERI_MS_PPU_FX79	0x40260800	8 B	RES
PERI_MS_PPU_FX80	0x40267000	4 KB	PWR_TRIM, CLK_TRIM
PERI_MS_PPU_FX81	0x4026FF00	128 B	PWR_TRIM, CLK_TRIM
PERI_MS_PPU_FX82	0x40270000	64 KB	BACKUP
PERI_MS_PPU_FX83	0x40280000	128 B	DW00

Table 9-2. Fixed PPU Structures

Protection Structure	Base Address of Protected Region	Size of Protected Region	Protected Peripheral/Block(s)
PERI_MS_PPU_FX84	0x40290000	128 B	DW10
PERI_MS_PPU_FX85	0x40280100	128 B	DW0_CRC
PERI_MS_PPU_FX86	0x40290100	128 B	DW1_CRC
PERI_MS_PPU_FX87	0x40288000	64 B	DW0_CH_STRUCT0
PERI_MS_PPU_FX88	0x40288040	64 B	DW0_CH_STRUCT1
PERI_MS_PPU_FX89	0x40288080	64 B	DW0_CH_STRUCT2
PERI_MS_PPU_FX90	0x402880C0	64 B	DW0_CH_STRUCT3
PERI_MS_PPU_FX91	0x40288100	64 B	DW0_CH_STRUCT4
PERI_MS_PPU_FX92	0x40288140	64 B	DW0_CH_STRUCT5
PERI_MS_PPU_FX93	0x40288180	64 B	DW0_CH_STRUCT6
PERI_MS_PPU_FX94	0x402881C0	64 B	DW0_CH_STRUCT7
PERI_MS_PPU_FX95	0x40288200	64 B	DW0_CH_STRUCT8
PERI_MS_PPU_FX96	0x40288240	64 B	DW0_CH_STRUCT9
PERI_MS_PPU_FX97	0x40288280	64 B	DW0_CH_STRUCT10
PERI_MS_PPU_FX98	0x402882C0	64 B	DW0_CH_STRUCT11
PERI_MS_PPU_FX99	0x40288300	64 B	DW0_CH_STRUCT12
PERI_MS_PPU_FX100	0x40288340	64 B	DW0_CH_STRUCT13
PERI_MS_PPU_FX101	0x40288380	64 B	DW0_CH_STRUCT14
PERI_MS_PPU_FX102	0x402883C0	64 B	DW0_CH_STRUCT15
PERI_MS_PPU_FX103	0x40288400	64 B	DW0_CH_STRUCT16
PERI_MS_PPU_FX104	0x40288440	64 B	DW0_CH_STRUCT17
PERI_MS_PPU_FX105	0x40288480	64 B	DW0_CH_STRUCT18
PERI_MS_PPU_FX106	0x402884C0	64 B	DW0_CH_STRUCT19
PERI_MS_PPU_FX107	0x40288500	64 B	DW0_CH_STRUCT20
PERI_MS_PPU_FX108	0x40288540	64 B	DW0_CH_STRUCT21
PERI_MS_PPU_FX109	0x40288580	64 B	DW0_CH_STRUCT22
PERI_MS_PPU_FX110	0x402885C0	64 B	DW0_CH_STRUCT23
PERI_MS_PPU_FX111	0x40288600	64 B	DW0_CH_STRUCT24
PERI_MS_PPU_FX112	0x40288640	64 B	DW0_CH_STRUCT25
PERI_MS_PPU_FX113	0x40288680	64 B	DW0_CH_STRUCT26
PERI_MS_PPU_FX114	0x402886C0	64 B	DW0_CH_STRUCT27
PERI_MS_PPU_FX115	0x40288700	64 B	DW0_CH_STRUCT28
PERI_MS_PPU_FX116	0x40298000	64 B	DW1_CH_STRUCT0
PERI_MS_PPU_FX117	0x40298040	64 B	DW1_CH_STRUCT1
PERI_MS_PPU_FX118	0x40298080	64 B	DW1_CH_STRUCT2
PERI_MS_PPU_FX119	0x402980C0	64 B	DW1_CH_STRUCT3
PERI_MS_PPU_FX120	0x40298100	64 B	DW1_CH_STRUCT4
PERI_MS_PPU_FX121	0x40298140	64 B	DW1_CH_STRUCT5
PERI_MS_PPU_FX122	0x40298180	64 B	DW1_CH_STRUCT6
PERI_MS_PPU_FX123	0x402981C0	64 B	DW1_CH_STRUCT7
PERI_MS_PPU_FX124	0x40298200	64 B	DW1_CH_STRUCT8
PERI_MS_PPU_FX125	0x40298240	64 B	DW1_CH_STRUCT9
PERI_MS_PPU_FX126	0x40298280	64 B	DW1_CH_STRUCT10

Table 9-2. Fixed PPU Structures

Protection Structure	Base Address of Protected Region	Size of Protected Region	Protected Peripheral/Block(s)
PERI_MS_PPU_FX127	0x402982C0	64 B	DW1_CH_STRUCT11
PERI_MS_PPU_FX128	0x40298300	64 B	DW1_CH_STRUCT12
PERI_MS_PPU_FX129	0x40298340	64 B	DW1_CH_STRUCT13
PERI_MS_PPU_FX130	0x40298380	64 B	DW1_CH_STRUCT14
PERI_MS_PPU_FX131	0x402983C0	64 B	DW1_CH_STRUCT15
PERI_MS_PPU_FX132	0x40298400	64 B	DW1_CH_STRUCT16
PERI_MS_PPU_FX133	0x40298440	64 B	DW1_CH_STRUCT17
PERI_MS_PPU_FX134	0x40298480	64 B	DW1_CH_STRUCT18
PERI_MS_PPU_FX135	0x402984C0	64 B	DW1_CH_STRUCT19
PERI_MS_PPU_FX136	0x40298500	64 B	DW1_CH_STRUCT20
PERI_MS_PPU_FX137	0x40298540	64 B	DW1_CH_STRUCT21
PERI_MS_PPU_FX138	0x40298580	64 B	DW1_CH_STRUCT22
PERI_MS_PPU_FX139	0x402985C0	64 B	DW1_CH_STRUCT23
PERI_MS_PPU_FX140	0x40298600	64 B	DW1_CH_STRUCT24
PERI_MS_PPU_FX141	0x40298640	64 B	DW1_CH_STRUCT25
PERI_MS_PPU_FX142	0x40298680	64 B	DW1_CH_STRUCT26
PERI_MS_PPU_FX143	0x402986C0	64 B	DW1_CH_STRUCT27
PERI_MS_PPU_FX144	0x40298700	64 B	DW1_CH_STRUCT28
PERI_MS_PPU_FX145	0x402A0000	16 B	DMAC
PERI_MS_PPU_FX146	0x402A1000	256 B	DMAC_CH0
PERI_MS_PPU_FX147	0x402A1100	256 B	DMAC_CH1
PERI_MS_PPU_FX148	0x402A1200	256 B	DMAC_CH2
PERI_MS_PPU_FX149	0x402A1300	256 B	DMAC_CH3
PERI_MS_PPU_FX150	0x402C0000	128 B	EFUSE
PERI_MS_PPU_FX151	0x402C0800	512 B	EFUSE
PERI_MS_PPU_FX152	0x402D0000	64 KB	PROFILE
PERI_MS_PPU_FX153	0x40300000	8 B	HSIOM_PRT0_PORT
PERI_MS_PPU_FX154	0x40300010	8 B	HSIOM_PRT1_PORT
PERI_MS_PPU_FX155	0x40300020	8 B	HSIOM_PRT2_PORT
PERI_MS_PPU_FX156	0x40300030	8 B	HSIOM_PRT3_PORT
PERI_MS_PPU_FX157	0x40300040	8 B	HSIOM_PRT4_PORT
PERI_MS_PPU_FX158	0x40300050	8 B	HSIOM_PRT5_PORT
PERI_MS_PPU_FX159	0x40300060	8 B	HSIOM_PRT6_PORT
PERI_MS_PPU_FX160	0x40300070	8 B	HSIOM_PRT7_PORT
PERI_MS_PPU_FX161	0x40300080	8 B	HSIOM_PRT8_PORT
PERI_MS_PPU_FX162	0x40300090	8 B	HSIOM_PRT9_PORT
PERI_MS_PPU_FX163	0x403000A0	8 B	HSIOM_PRT10_PORT
PERI_MS_PPU_FX164	0x403000B0	8 B	HSIOM_PRT11_PORT
PERI_MS_PPU_FX165	0x403000C0	8 B	HSIOM_PRT12_PORT
PERI_MS_PPU_FX166	0x403000D0	8 B	HSIOM_PRT13_PORT
PERI_MS_PPU_FX167	0x403000E0	8 B	HSIOM_PRT14_PORT
PERI_MS_PPU_FX168	0x40302000	32 B	HSIOM_AMUX_SPLIT
PERI_MS_PPU_FX169	0x40302200	16 B	

Table 9-2. Fixed PPU Structures

Protection Structure	Base Address of Protected Region	Size of Protected Region	Protected Peripheral/Block(s)
PERI_MS_PPU_FX170	0x40310000	64 B	GPIO_PRT0
PERI_MS_PPU_FX171	0x40310080	64 B	GPIO_PRT1
PERI_MS_PPU_FX172	0x40310100	64 B	GPIO_PRT2
PERI_MS_PPU_FX173	0x40310180	64 B	GPIO_PRT3
PERI_MS_PPU_FX174	0x40310200	64 B	GPIO_PRT4
PERI_MS_PPU_FX175	0x40310280	64 B	GPIO_PRT5
PERI_MS_PPU_FX176	0x40310300	64 B	GPIO_PRT6
PERI_MS_PPU_FX177	0x40310380	64 B	GPIO_PRT7
PERI_MS_PPU_FX178	0x40310400	64 B	GPIO_PRT8
PERI_MS_PPU_FX179	0x40310480	64 B	GPIO_PRT9
PERI_MS_PPU_FX180	0x40310500	64 B	GPIO_PRT10
PERI_MS_PPU_FX181	0x40310580	64 B	GPIO_PRT11
PERI_MS_PPU_FX182	0x40310600	64 B	GPIO_PRT12
PERI_MS_PPU_FX183	0x40310680	64 B	GPIO_PRT13
PERI_MS_PPU_FX184	0x40310700	64 B	GPIO_PRT14
PERI_MS_PPU_FX185	0x40310040	16 B	GPIO_PRT0
PERI_MS_PPU_FX186	0x403100C0	16 B	GPIO_PRT1
PERI_MS_PPU_FX187	0x40310140	16 B	GPIO_PRT2
PERI_MS_PPU_FX188	0x403101C0	16 B	GPIO_PRT3
PERI_MS_PPU_FX189	0x40310240	16 B	GPIO_PRT4
PERI_MS_PPU_FX190	0x403102C0	16 B	GPIO_PRT5
PERI_MS_PPU_FX191	0x40310340	16 B	GPIO_PRT6
PERI_MS_PPU_FX192	0x403103C0	16 B	GPIO_PRT7
PERI_MS_PPU_FX193	0x40310440	16 B	GPIO_PRT8
PERI_MS_PPU_FX194	0x403104C0	16 B	GPIO_PRT9
PERI_MS_PPU_FX195	0x40310540	16 B	GPIO_PRT10
PERI_MS_PPU_FX196	0x403105C0	16 B	GPIO_PRT11
PERI_MS_PPU_FX197	0x40310640	16 B	GPIO_PRT12
PERI_MS_PPU_FX198	0x403106C0	16 B	GPIO_PRT13
PERI_MS_PPU_FX199	0x40310740	8 B	GPIO_PRT14
PERI_MS_PPU_FX200	0x40314000	64 B	GPIO_INTR
PERI_MS_PPU_FX201	0x40315000	8 B	
PERI_MS_PPU_FX202	0x40320800	256 B	SMARTIO_PRT8
PERI_MS_PPU_FX203	0x40320900	256 B	SMARTIO_PRT9
PERI_MS_PPU_FX204	0x40350000	64 KB	LPCOMP
PERI_MS_PPU_FX205	0x40360000	4 KB	CSD
PERI_MS_PPU_FX206	0x40380000	64 KB	TCPWM0
PERI_MS_PPU_FX207	0x40390000	64 KB	TCPWM1
PERI_MS_PPU_FX208	0x403B0000	64 KB	LCD0
PERI_MS_PPU_FX209	0x403F0000	64 KB	USBFS0
PERI_MS_PPU_FX210	0x40420000	64 KB	SMIF0
PERI_MS_PPU_FX211	0x40460000	64 KB	SDHC0
PERI_MS_PPU_FX212	0x40470000	64 KB	SDHC1

Table 9-2. Fixed PPU Structures

Protection Structure	Base Address of Protected Region	Size of Protected Region	Protected Peripheral/Block(s)
PERI_MS_PPU_FX213	0x40600000	64 KB	SCB0
PERI_MS_PPU_FX214	0x40610000	64 KB	SCB1
PERI_MS_PPU_FX215	0x40620000	64 KB	SCB2
PERI_MS_PPU_FX216	0x40630000	64 KB	SCB3
PERI_MS_PPU_FX217	0x40640000	64 KB	SCB4
PERI_MS_PPU_FX218	0x40650000	64 KB	SCB5
PERI_MS_PPU_FX219	0x40660000	64 KB	SCB6
PERI_MS_PPU_FX220	0x40670000	64 KB	SCB7
PERI_MS_PPU_FX221	0x40680000	64 KB	SCB8
PERI_MS_PPU_FX222	0x40690000	64 KB	SCB9
PERI_MS_PPU_FX223	0x406A0000	64 KB	SCB10
PERI_MS_PPU_FX224	0x406B0000	64 KB	SCB11
PERI_MS_PPU_FX225	0x406C0000	64 KB	SCB12
PERI_MS_PPU_FX226	0x40A00000	4 KB	PDM0
PERI_MS_PPU_FX227	0x40A10000	4 KB	I2S0
PERI_MS_PPU_FX228	0x40A11000	4 KB	I2S1

- The programmable PPU structures can have configurable address regions. Similar to fixed-PPU structures, the protection attributes of programmable PPU structures can be configured for each protection context. Programmable PPU structures are similar to SMPU structures but are intended to be used with the peripheral register space. These protection structures are typically used to protect registers in a specific block, which are not covered by the resolution of fixed PPU structures.

  Note that the memory regions of the fixed master structures, fixed slave structures, and programmable master structures are fixed by hardware and are mutually exclusive; that is, they do not overlap. The memory regions of the programmable slave structure are software-programmable and can potentially overlap. Therefore, it is important to assign priority to the protection structure matching process. The order in which these are evaluated are as follows:

  - The fixed master structures are evaluated in decreasing order.
  - The fixed slave structures are evaluated in decreasing order.
  - The programmable master structures are evaluated in decreasing order.
  - The programmable slave structures are evaluated in decreasing order.

  The programmable slave structures are evaluated last. These structures are software-programmable and can potentially overlap (overlapping should not allow software to circumvent the protection as provided by the fixed protection structure pairs).

  Each peripheral group has a dedicated PPU. The protection information is provided by peripheral group MMIO registers. A peripheral group PPU uses fixed protection structure pairs for two purposes.

  - Fixed protection structure pairs protect peripherals (one pair for each peripheral). The master structure protects the MMIO registers of the pair (the memory region encompasses the MMIO registers of the pair's master structure and slave structure). The slave structure protects the peripheral (the memory region encompasses the peripheral address region).
  - Fixed protection structure pairs protect specific peripheral subregions (one pair for each subregion). The master structure protects the MMIO registers of the pair. The slave structure protects the peripheral subregion. These pairs can be used to protect, for example, individual DW channels in the DW peripheral or individual IPC structures in the IPC peripheral.

  Note that the memory regions of the fixed peripheral master structures, fixed peripheral slave structures, and fixed peripheral subregion master structures are fixed by hardware and are mutually exclusive; that is, they do not overlap. The memory regions of the fixed peripheral subregion slave structures are fixed by hardware and typically are a subset of a peripheral address region, and therefore overlap with a fixed peripheral slave structure. Therefore, it is important to assign priority to how the protection structure matching process:

  - The fixed peripheral subregion master structures are evaluated in decreasing order.
  - The fixed peripheral subregion slave structures are evaluated in decreasing order.

- The fixed peripheral master structures are evaluated in decreasing order.
- The fixed peripheral slave structures are evaluated in decreasing order.

It is important to evaluate the fixed peripheral subregion master structures first. This allows software to assign different protection for a subregion of a peripheral.

9.7.5 Protection of Protection Structures

The MPU, SMPU, and PPU-based protection architecture is consistent and provides the flexibility to implement different system-wide protection schemes. Protection structures can be set once at boot time or can be changed dynamically during device execution. For example, a CPU RTOS can change the CPU's MPU settings; a secure CPU can change the SMPU and PPUs settings. But such a system will be left insecure if there is no way to protect the protection structures themselves. There must be a way to restrict access to the protection structures.

The protection of protection structures is achieved using another protection structure. For this reason, protection structures are defined in pairs of master and slave. We refer to the slave and master protection structures as a protection pair. Note that the address range of the master protection structure is known, that is, it is constant.

The first (slave) protection structure protects the resource and the second (master) protection structure protects the protection (address range of the second protection structure includes both the master and slave protection structures).

The protection architecture is flexible enough to allow for variations:

- Exclusive peripheral ownership can be shared by more than two protection contexts.
- The ability to change ownership is under control of a single protection context, and exclusive or non-exclusive peripheral ownership is shared by multiple protection contexts.

Note that in secure systems, typically a single secure CPU is used. In these systems, the ability to change ownership is assigned to the secure CPU at boot time and not dynamically changed. Therefore, you must assign the secure CPU its own, dedicated protection context.

Both PPU and SMPU is intended to distinguish between different protection contexts and to distinguish secure from non-secure accesses. Therefore, both PPU and SMPU protection use protection structure pairs. In the SMPU, the slave protection structure provides SMPU protection information and the master protection structure provides PPU protection information (the master and slave protection structures are registers).

9.7.6 Protection Structure Types

Different protection structure types are used because some resources, such as peripheral registers, have a fixed address range. Protection of protection structures requires pairs of neighboring protection structures.

Three types of protection structures with a consistent register interface are described here:

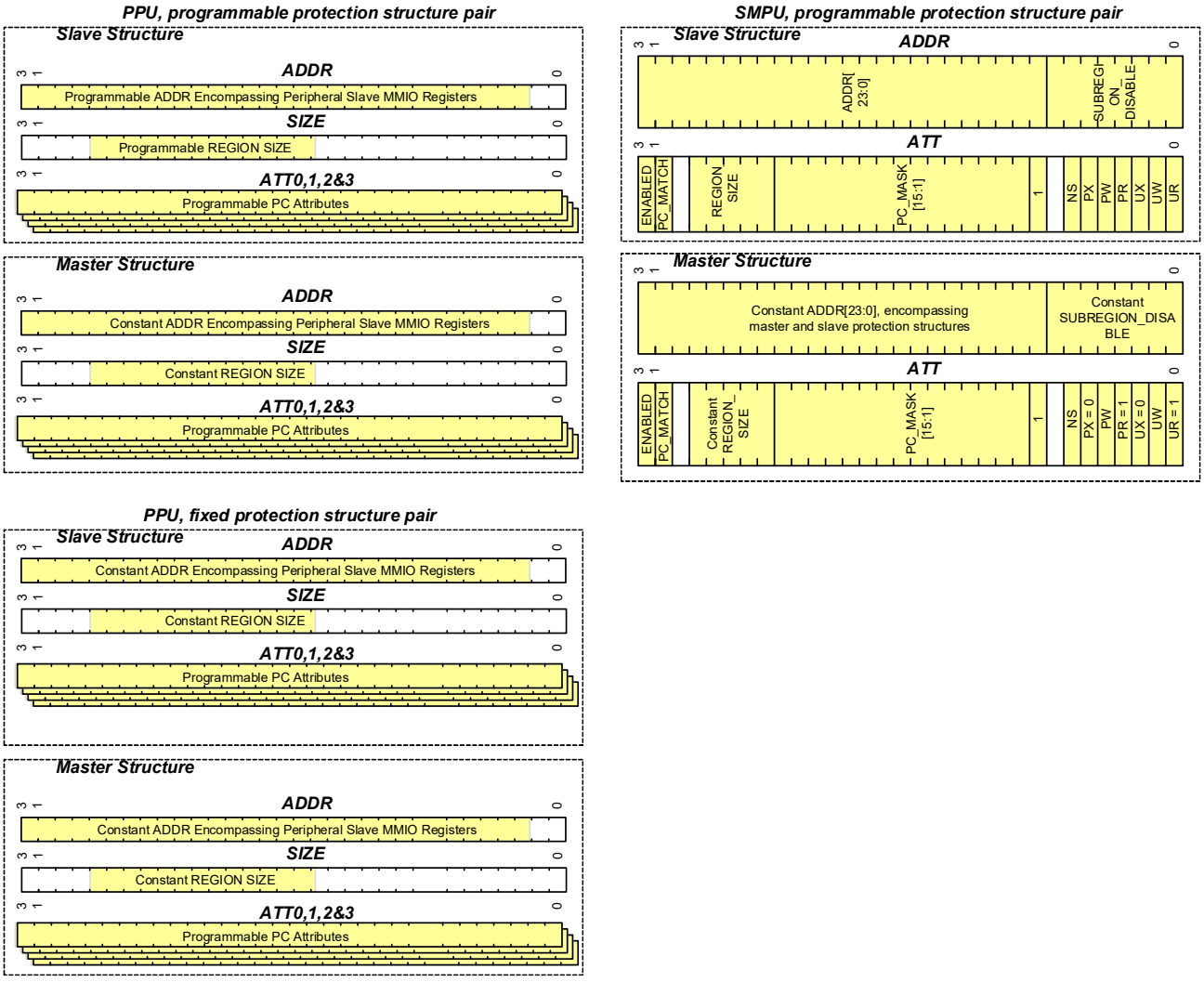
- Programmable protection structures. These are 32-byte protection structures with a programmable address range. These structures are used by the MPUs.
- Fixed protection structure pairs. These are 64-byte master/slave protection structure pairs, consisting of two 32-byte protection structures. These structures are used by the PPUs. Both structures have a fixed, constant address region. The master structure has the UX and PX attributes as constant '0' (execution is never allowed) and the UR and PR attributes as constant '1' (reading is always allowed). The slave structure has the UX and PX attributes as constant '1'.
- Programmable protection structure pairs. These are 64-byte master/slave protection structure pairs, consisting of two 32-byte protection structures. These structures are used by the PPU and SMPU. The master structure has a fixed, constant address region. The slave structure has a programmable address region. The master structure has the UX and PX attributes as constant '0' (execution is never allowed) and the UR and PR attributes as constant '1' (reading is always allowed). The PPU slave structure has the UX and PX attributes as constant '1'. The SMPU slave structure has programmable UX and PX attributes.

Note that the master protection structure in a protection structure pair is required only to address security requirements. The distinction between the three protection structure types is an implementation optimization. From an architectural perspective, all PPU protection structures are the same, with the exception that for some protection structures the address range is fixed and not programmable.

As mentioned earlier, a protection unit evaluates the protection regions in decreasing protection structure index order. The protection structures are evaluated in the following order:

- Fixed protection structures for specific peripherals or peripheral register address ranges.
- Programmable protection structures.

Figure 9-6. PPU and SMPU Pairs



Note: By default, both CPUs (CM0+ and CM4) are in protection context 0 when they come out of reset. In protection context 0, the master is able to access all memory regardless of its protection settings. The master's protection context will need to be changed from protection context 0 to make any protection structure configuration effective. Multiple protection structures may be preconfigured as part of the boot code, which sets up a secure environment at boot time. See the Boot Code chapter on page 193 for details of these configurations.

10. DMA Controller (DW)

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The DMA transfers data to and from memory, peripherals, and registers. These transfers occur independent from the CPU. The DMA can be configured to perform multiple independent data transfers. All data transfers are managed by a channel. There can be up to 32 channels in the DMA. The number of channels in the DMA controller can vary with devices. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for the number of channels supported in the device. A channel has an associated priority; channels are arbitrated according to their priority.

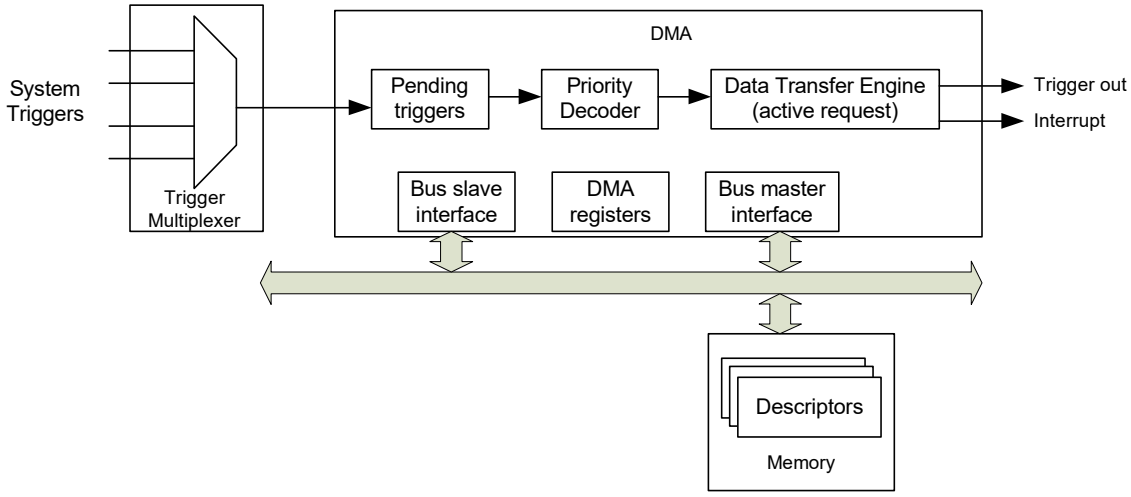
10.1 Features

The DMA controller has the following features:

- Supports up to 29 channels per DMA controller; see the PSoC 61 datasheet/PSoC 62 datasheet for details
- Supports multiple DMA controller instances in a device
- Four levels of priority for each channel
- Descriptors are defined in memory and referenced to the respective channels
- Supports single, 1D, or 2D transfer modes for a descriptor
- Supports transfer of up to 65536 data elements per descriptor
- Configurable source and destination address increments
- Supports 8-bit, 16-bit, and 32-bit data widths at both source and destination
- Configurable input trigger behavior for each descriptor
- Configurable interrupt generation in each descriptor
- Configurable output trigger generation for each descriptor
- Descriptors can be chained to other descriptors in memory

10.2 Architecture

Figure 10-1. DMA Controller



A data transfer is initiated by an input trigger. This trigger may originate from the source peripheral of the transfer, the destination peripheral of the transfer, CPU software, or from another peripheral. Triggers provide Active/Sleep functionality and are not available in Deep Sleep and Hibernate power modes.

The data transfer details are specified by a descriptor. Among other things, this descriptor specifies:

- The source and destination address locations and the size of the transfer.
- The actions of a channel; for example, generation of output triggers and interrupts. See the Interrupts chapter on page 56 for more details.
- Data transfer types can be single, 1D, or 2D as defined in the descriptor structure. These types define the address sequences generated for source and destination. 1D and 2D transfers are used for “scatter gather” and other useful transfer operations.

10.3 Channels

The DMA controller supports multiple independent data transfers that are managed by a channel. Each channel connects to a specific system trigger through a trigger multiplexer that is outside the DMA controller.

Channel priority: A channel is assigned a priority (CHi_CTL.PRIO) between 0 and 3, with 0 being the highest priority and 3 being the lowest priority. Channels with the same priority constitute a priority group. Priority decoding determines the highest priority pending channel, which is determined as follows.

- The highest priority group with pending channels is identified first.
- Within this priority group, round-robin arbitration is applied.

Channel state: At any given time, one channel actively performs a data transfer. This channel is called the active channel. A channel can be in one of four channel states. The active channel in a DW controller can be determined by reading the DWx_STATUS[ACTIVE] and DWx.STATUS[CH_IDX].

Pending state of a channel is determined by reading the DW_CH_STRUCT_CH_STATUS[PENDING] associated with that channel. If a channel is enabled and is not in the Pending or Active state, then it is considered blocked.

Table 10-1. Channel States

Channel State	Description
Disabled	The channel is disabled by setting CHi_CTL.ENABLED to ‘0’. The channel trigger is ignored in this state.
Blocked	The channel is enabled and is waiting for a trigger to initiate a data transfer.
Pending	The channel is enabled and has received an active trigger. In this state, the channel is ready to initiate a data transfer but waiting for it to be scheduled.
Active	The channel is enabled, has received an active trigger and has been scheduled. It is actively performing data transfers. If there are multiple channels pending, the highest priority pending channel is scheduled.

The data transfer associated with a trigger is made up of one or more ‘atomic transfers’ or ‘single transfers’; see Table 10-2 for a better understanding. A single trigger could be configured to transfer multiple “single transfers”.

A channel can be marked preemptable (CHi_CTL.PREEMPTABLE). If preemptable, and there is a higher priority pending channel, then that higher priority channel can preempt the current channel between single transfers. If a channel is preempted, the existing single transfer is completed; the current channel goes to pending state and the higher priority channel is serviced. On completion of the higher priority channel's transfer, the pending channel is resumed. Note that preemption has an impact on the data transfer rates of the channel being preempted. Refer to “DMA Performance” on page 100 for these performance implications.

A channel has two access control attributes that are used by the shared memory protection units (SMPUs) and peripheral protection units (PPUs) for access control. These fields are typically inherited from the master that modified the channel’s control register.

- The Privileged Mode (CHi_CTL.P) attribute can be set to privileged or user.
- The Non-secure (CHi_CTL.NS) attribute can be set to secure or non-secure.

A descriptor associated with each channel describes the data transfer. The descriptor is stored in memory and CHi_CURR_PTR provides the descriptor address associated with channel “i” and Chi_IDX provides the current X and Y indices into the descriptor.

A channel’s descriptor state is encoded as part of the channel’s register state. The following registers provide a channel’s descriptor state:

- CH_CTL. This register provides generic channel control information.
- CH_CURR_PTR. This register provides the address of the memory location where the current descriptor is located. The user firmware must initialize this register. If the descriptors are chained, the DMA hardware automatically sets this register to the next descriptor pointer.
- CH_IDX. This register provides the current X and Y indices of the channel into the current descriptor. User firmware must initialize this register. DMA hardware sets the X and Y indices to 0, when advancing from the current descriptor to the next descriptor in a descriptor list.

Note that channel state is retained in Deep Sleep power mode.

10.3.1 Channel Interrupts

Every DMA channel has an interrupt line associated with it. The INTR_TYPE parameter in the descriptor determines the event that will trigger the interrupt for the channel. In addition each DMA channel has INTR, INTR_SET, INTR_MASK, and INTR_MASKED registers to control their respective interrupt lines. INTR_MASK can be used to mask the interrupt from the DMA channel. The INTR and INTR_SET can be used to clear and set the interrupt, respectively, for debug purposes.

The DW_CH_STRUCT_CH_STATUS[INTR_CAUSE] field provides the user a means to determine the cause of the interrupt being generated. The following are different values for this register:

- 0: No interrupt generated
- 1: Interrupt based on transfer completion configured based on INTR_TYPE field in the descriptor
- 2: Source bus error
- 3: Destination bus error
- 4: Misaligned source address
- 5: Misaligned destination address
- 6: Current descriptor pointer is null
- 7: Active channel is in disabled state
- 8: Descriptor bus error
- 9-15: Not used.

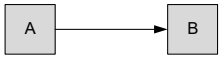
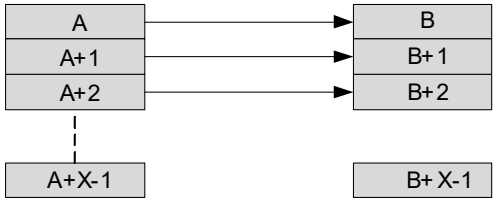
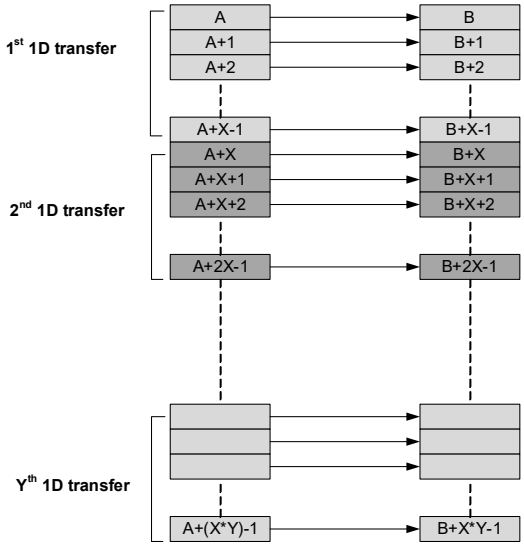
For error related interrupt causes (INTR_CAUSE is 2, 3,..., 8), the channel is disabled (hardware sets CH_CTL.ENABLED to ‘0’).

The bus errors are typically caused by incompatible accesses to the addresses in question. This may be due to those addresses being protected or having read or write restrictions. Source and destination bus errors can also occur due to mismatch in data sizes (see “Transfer Size” on page 97).

10.4 Descriptors

The data transfer between a source and destination in a channel is configured using a descriptor. Descriptors are stored in memory. The descriptor pointer is specified in the DMA channel registers. The DMA controller does not modify the descriptor and treats it as read only. A descriptor is a set of up to six 32-bit registers that contain the configuration for the transfer in the associated channel. There are three types of descriptors.

Table 10-2. Descriptor Types

Descriptor Type	Description
Single transfer	Transfers a single data element 
1D transfer	Performs a one-dimensional “for loop”. This transfer is made up of X number of single transfers 
2D transfer	Performs a two-dimensional “for loop”. This transfer is made up of Y number of 1D transfers 
CRC transfer	This performs a one-dimensional “for loop” similar to the 1D transfer. However, the source data is not transferred to a destination. Instead, a CRC is calculated over the source data. The CRC configuration is provided through a set of registers that is shared by all DMA channels. The assumption is that the DMA channels use the CRC functionality mutually exclusively in time. These registers are: CRC_CTL, CRC_DATA_CTL, CRC_POL_CTL, CRC_LFSR_CTL, CRC_REM_CTL, and CRC_REM_RESULT. Note that the CRC configuration is the same as the Crypto CRC configuration.

Single Transfer:

The following pseudo code illustrates a single transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
DST_ADDR[0] = (t_DATA_SIZE) SRC_ADDR[0];
```

1D Transfer:

The following pseudo code illustrates a 1D transfer. Note that the 1D transfer is represented by a loop with each iteration executing a single transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
DST_ADDR[X_IDX * DST_X_INCR] =
          (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR];
}
```

2D Transfer:

The following pseudo code illustrates a 2D transfer. Note that the 2D transfer is represented by a loop with each iteration executing an inner loop, which is the 1D transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
for (Y_IDX = 0; Y_IDX <= Y_COUNT; Y_IDX++) {
    for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
DST_ADDR[X_IDX * DST_X_INCR + Y_IDX * DST_Y_INCR ] =
          (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR + Y_IDX * SRC_Y_INCR];
    }
}
```

CRC Transfer:

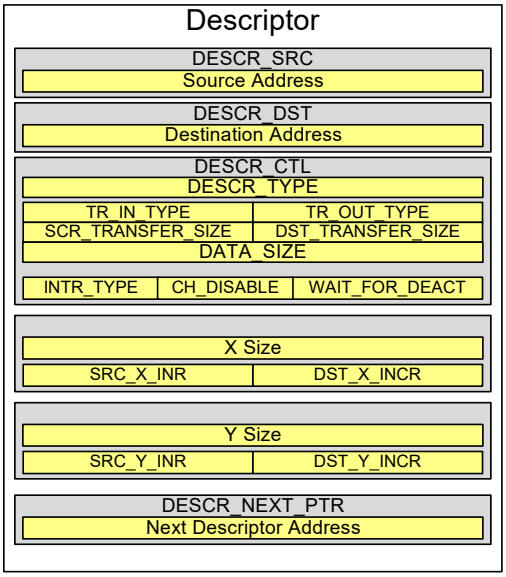
The following psuedo code illustrates CRC transfer.

```
// DST_ADDR is a pointer to an address location where the calculated CRC is stored.
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
CRC_STATE = CRC_LFSR_CTL;
for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
   Update_CRC (CRC_STATE, (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR];
}
DST_ADDR = CRC_STATE;
```

The parameters in the descriptor help configure the different aspects of the transfers explained.

Figure 10-2 shows the structure of a descriptor.

Figure 10-2. Descriptor Structure



10.4.1 Address Configuration

Source and Destination Address: The source and destination addresses are set in the respective registers in the descriptor. These set the base addresses for the source and destination location for the transfer. In case the descriptor is configured to transfer a single element, this field holds the source/destination address of the data element. If the descriptor is configured to transfer multiple elements with source address or destination address or both in an incremental mode, this field will hold the address of the first element that is transferred.

DESCR_TYPE: This field configures whether the descriptor has a single, 1D, or 2D type.

Trigger input type, TR_IN_TYPE: This field determines how the DMA engine responds to input trigger signal. This field can be configured for one of the following modes:

- Type 0: A trigger results in execution of a single transfer. Regardless of the DESCR_TYPE setting, a trigger input will trigger only a single element transfer. For example, in a 1D transfer, the DMA will transfer only one data element in every trigger.
- Type 1: A trigger results in the execution of a single 1D transfer. If the DESCR_TYPE was set to single transfer, the trigger signal will trigger the single transfer specified by the descriptor. For a DESCR_TYPE set to 1D transfer, the trigger signal will trigger the entire 1D transfer configured in the descriptor. For a 2D transfer, the trigger signal will trigger only a single iteration of the Y loop transfer.
- Type 2: A trigger results in execution of the current descriptor. Regardless of DESCR_TYPE, the trigger will execute the entire descriptor. If there was a next descriptor configured for the current descriptor, this trigger setting will not automatically trigger the next descriptor.
- Type 3: A trigger results in execution of the current descriptor and also triggering the next descriptor. The execution of the next descriptor from this point will be determined by the TR_IN_TYPE setting of the next descriptor.

Trigger out type, TR_OUT_TYPE: This field determines what completion event will generate the output trigger signal. This field can be configured to one of the following modes:

- Type 0: Generates a trigger output for completion of every single element transfer.
- Type 1: Generates a trigger output for completion of a 1D transfer
- Type 2: Generates a trigger output for completion of the current descriptor. This trigger output is generated independent of the state of the DESCR_NEXT_PTR.
- Type 3: Generates a trigger output on completion of the current descriptor, when the current descriptor is the last descriptor in the descriptor chain. This means a trigger is generated when the descriptor execution is complete and the DESCR_NEXT_PTR is ‘0’.

Interrupt Type, INTR_TYPE: This field determines which completion event will generate the output interrupt signal. This field can be configured to one of the following modes:

- Type 0: Generates an interrupt output for completion of every single element transfer.
- Type 1: Generates an interrupt output for completion of a 1-D transfer.

- Type 2: Generates an interrupt output for completion of the current descriptor. This interrupt output is generated independent of the state of the DESCR_NEXT_PTR.
- Type 3: Generates an interrupt output on completion of the current descriptor, when the current descriptor is the last descriptor in the descriptor chain. This means an interrupt is generated when the descriptor execution is complete and the DESCR_NEXT_PTR is '0'.

WAIT_FOR_DEACT: When the DMA transfer based on the TR_IN_TYPE is completed, the data transfer engine checks the state of trigger deactivation. The data transfer on the second trigger is initiated only after deactivation of the first. The WAIT_FOR_DEACT parameter will determine when the trigger signal is considered deactivated. The first DMA transfer is activated when the trigger is activated, but the transfer is not considered complete until the trigger is deactivated. This field is used to synchronize the controller's data transfers with the agent that generated the trigger. This field has four settings:

- 0 – Pulse Trigger: Do not wait for deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active then it is considered as another trigger and the subsequent transfer is initiated immediately.
- 1 – Level-sensitive waits four slow clock cycles after the transfer to consider as a deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active then it is considered as another trigger after waiting for four cycles. Then, a subsequent transfer is initiated. The transfer corresponding to the trigger is considered complete only at the end of the four additional cycles. Even trigger output events will be affected based on this delay. This parameter adds a four-cycle delay in each trigger transaction and hence affects throughput.
- 2 – Level-sensitive waits 16 slow clock cycles after the transfer to consider as a deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active then it is considered as another trigger after waiting for 16 cycles. Then, a subsequent transfer is initiated. The transfer corresponding to the trigger is considered complete only at the end of the 16 additional cycles. Even trigger output events will be affected based on this delay. This parameter adds a 16-cycle delay in each trigger transaction and hence affects throughput.
- 3 – Pulse trigger waits indefinitely for deactivation. The DMA transfer is initiated after the trigger signal deactivates. The next transfer is initiated only if the trigger goes low and then high again. A trigger signal that remains active or does not transition to zero between two transaction will simply stall the DMA channel.

  The WAIT_FOR_DEACT field is used in a system to cater to delayed response of other parts of the system to actions of the DMA. Consider an example of a TX FIFO that has a trigger going to the DMA when its not full. Free space in FIFO will trigger a DMA transfer to the FIFO, which in turn will deactivate the trigger. However, there can be a delay in this deactivation by the agent, which may cause the DMA to have initiated another transfer that can cause a FIFO overflow. This can be avoided by using the four or 16 clock cycle delays.

X Count: This field determines the number of single element transfers present in the X loop (inner loop). This field is valid when the DESCR_TYPE is set to 1D or 2D transfer.

Source Address Increment (X loop) (SCR_X_INCR): This field configures the index by which the source address is to be incremented for every iteration in an X loop. The field is expressed in multiples of SRC_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the source address does not need to be incremented, you can set this parameter to zero.

Destination Address Increment (X loop) (DST_X_INCR): This field configures the index by which the destination address is to be incremented, for every iteration in an X loop. The field is expressed in multiples of DST_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the destination address does not need to be incremented, you can set this parameter to zero.

Y Count: This field determines the number of 1-D transfers present in the Y loop (outer loop). This field is valid when the DESCR_TYPE is set to 2-D transfer.

Source Address Increment (Y loop) (SCR_Y_INCR): This field configures the index by which the source address is to be incremented, for every iteration in a Y loop. The field is expressed in multiples of SRC_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the source address does not need to be incremented, you can set this parameter to zero.

Destination Address Increment (X loop) (DST_Y_INCR): This field configures the index by which the destination address is to be incremented, for every iteration in a Y loop. The field is expressed in multiples of DST_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the destination address does not need to be incremented, you can set this parameter to zero.

Channel Disable (CH_DISABLE): This field specifies whether the channel is disabled or not after completion of the current descriptor (independent of the value of the DESCR_NEXT_PTR). A disabled channel will ignore its input triggers.

10.4.2 Transfer Size

The word width for a transfer can be configured using the transfer/data size parameter in the descriptor. The settings

are diversified into source transfer size, destination transfer size, and data size. The data size parameter (DATA_SIZE) sets the width of the bus for the transfer. The source and destination transfer sizes set by SCR_TRANSFER_SIZE and DST_TRANSFER_SIZE can have a value either the DATA_SIZE or 32 bit. DATA_SIZE can have a 32-bit, 16-bit, or 8-bit setting.

The source and destination transfer size for the DMA must match the addressable width of the source and destination, regardless of the width of data that must be moved. The DATA_SIZE parameter will correspond to the width of the actual data. For example, if a 16-bit PWM is used as a destination for DMA data, the DST_TRANSFER_SIZE must be set to 32 bit to match the width of the PWM register, because the peripheral register width for the TCPWM block (and most PSoC 6 MCU peripherals) is always 32-bit wide. However, in this example the DATA_SIZE for the destination may still be set to 16 bit because the 16-bit PWM only uses two bytes of data. SRAM and Flash are 8-bit, 16-bit, or 32-bit addressable and can use any source and destination transfer sizes to match the needs of the application.

Table 10-3 summarizes the possible combinations of the transfer size settings and its description.

Table 10-3. Transfer Size Settings

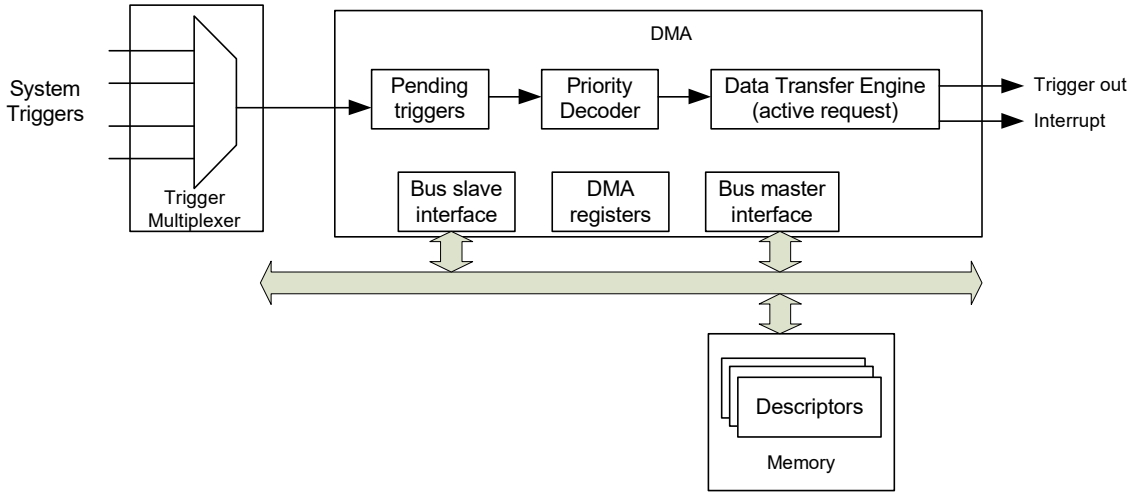
DATA_SIZE	SCR_TRANSFER_SIZE	DST_TRANSFER_SIZE	Typical Usage	Description
8-bit	8-bit	8-bit	Memory to Memory	No data manipulation
8-bit	32-bit	8-bit	Peripheral to Memory	Higher 24 bits from the source dropped
8-bit	8-bit	32-bit	Memory to Peripheral	Higher 24 bits zero padded at destination
8-bit	32-bit	32-bit	Peripheral to Peripheral	Higher 24 bits from the source dropped and higher 24 bits zero padded at destination
16-bit	16-bit	16-bit	Memory to Memory	No data manipulation
16-bit	32-bit	16-bit	Peripheral to Memory	Higher 16 bits from the source dropped
16-bit	16-bit	32-bit	Memory to Peripheral	Higher 16 bits zero padded at destination
16-bit	32-bit	32-bit	Peripheral to Peripheral	Higher 16 bits from the source dropped and higher 16-bit zero padded at destination
32-bit	32-bit	32-bit	Peripheral to Peripheral	No data manipulation

10.4.3 Descriptor Chaining

Descriptors can be chained together. The DESCR_NEXT_PTR field contains a pointer to the next descriptor in the chain. A channel executes the next descriptor in the chain when it completes executing the current descriptor. The last descriptor in the chain has DESCR_NEXT_PTR set to ‘0’ (NULL pointer). A descriptor chain is also referred to as a descriptor list. It is possible to have a circular list; in a circular list, the execution continues indefinitely until there is an error or the channel or the controller is disabled by user code.

10.5 DMA Controller

Figure 10-3. DMA Controller Overview



10.5.1 Trigger Selection

Trigger signals can be generated from different sections of the chips. A trigger multiplexer block helps route these trigger signals to the destination. The DMA is one such destination of triggers. The trigger multiplexer block is outside the DMA block and is discussed in the Trigger Multiplexer Block chapter on page 294.

10.5.2 Pending Triggers

Pending triggers keep track of activated triggers by locally storing them in pending bits. This is essential because multiple channel triggers may be activated simultaneously, whereas only one channel can be served by the data transfer engine at a time. This component enables the use of both level-sensitive (high/‘1’) and pulse-sensitive (two high/‘1’ clk_slow cycles) triggers.

- Level-sensitive triggers are associated with a certain state, for example, a FIFO being full. These triggers remain active as long as the state is maintained. It is not required to track pending level-sensitive triggers in the DMA controller because the triggers are maintained outside the controller.
- Pulse-sensitive triggers are associated with a certain event, for example, sample has become available. It is essential to track these triggers in the DMA controller because the trigger pulse may disappear before it is served by the data transfer engine. Pulse triggers should be high/‘1’ for two clk_slow cycles.

The **priority decoder** determines the highest priority pending channel.

The **data transfer engine** is responsible for the data transfer from a source location to a destination location. When idle, the data transfer engine is ready to accept the highest priority activated channel. It is also responsible for reading the channel descriptor from memory.

Master I/F is an AHB-Lite bus master that allows the DMA controller to initiate AHB-Lite data transfers to the source and destination locations as well as to read the descriptor from memory.

Slave I/F is an AHB-Lite bus slave that allows the main CPU to access DMA controller control/status registers.

10.5.3 Output Triggers

Each channel has an output trigger. This trigger is high for two slow clock cycles. The trigger is generated on the completion of a data transfer. At the system level, these output triggers can be connected to the trigger multiplexer component. This connection allows a DMA controller output trigger to be connected to a DMA controller input trigger. In other words, the completion of a transfer in one channel can activate another channel or even reactivate the same channel.

DMA output triggers routing to other DMA channels or other peripheral trigger inputs is achieved using the trigger multiplexer. Refer to the Trigger Multiplexer Block chapter on page 294.

10.5.4 Status registers

The controller status register (DWx_STATUS) contains the following information.

- ACTIVE – Active channel present, yes/no.
- P – Active channel's access control user/privileged
- NS – Active channel's access control secure/non-secure
- CH_IDX – Active channel index if there is an active channel

- PRIO – Active channel priority
- PREEMPTABLE – Active channel pre-emptable or not
- STATE – State of the DW controller state machine. The following states are specified:
  - Default/inactive state
  - Loading descriptor: This state is when the controller has recognized the channel that was triggered and become active and is now loading its respective descriptor.
  - Loading data element: Reading data from source address
  - Storing data element: Writing data into the destination address
  - Update of active channel control information
  - Waiting for input trigger deactivation

10.5.5 DMA Performance

The DMA block works on the clk_slow domain and hence all clocks described in this section are in clk_slow units.

Every time a DMA channel is triggered the DMA hardware goes through the following steps:

- Trigger synchronization
- Detection, priority decoding, and making channel pending
- Start state machine and load channel configuration
- Load DMA descriptor
- Load next DMA descriptor pointer
- Moving first element of data from source to destination.

Each of these steps involve multiple cycles for completion. Table 10-4 shows the number of cycles needed for each step.

Table 10-4. DMA Steps and Performance

Operation	Cycles
Trigger Synchronization	2
Detection, priority decoding and making channel pending	1
Start state machine and load channel config	3
Load descriptors	4 for single transfer 5 for 1-D transfer 6 for 2-D transfer
Load next pointer	1
Moving data from source to destination	3
Total	14 for single transfer

For subsequent transfers on a preloaded descriptor, cycles are needed only to move the data from source to destination. Therefore, transfers such as 1-D and 2-D, which are not preempted, incurs all the cycles only for the first transfer; subsequent transfers will cost three cycles.

Based on the configuration of TRIG_IN_TYPE, the trigger synchronization cycles may be incurred for each single element transfer or for each 1-D transfer.

The descriptor is four words long for a single transfer type, five words for 1-D transfer, and six words for a 2-D transfer. Hence, the number of cycles needed to fetch a descriptor will vary based on its type.

Another factor to note is the latency in data or descriptor fetch due to wait states or bus latency.

The DMA performance for different types of transfers can be summarized as follows.

- Single transfer
  - 14 cycles per transfer + latency due to wait states or bus latency
- 1D transfer
  - To transfer n data elements

  Number of cycles = 12 + n * 3 + m

  m is the total number of wait states seen by DMA while loading or storing descriptors or data. An additional cycle is required for the first transfer, to load the X-Loop configuration register.
- 2D transfer
  - If the 2 D transfer is transferring n elements then

  Number of cycles = 13 + n * 3 + m

m is total number of wait states seen by DMA while loading or storing descriptors or data. Two additional cycles are required for the first transfer, to load the X-loop and Y-Loop configuration register.

Note: Descriptors in memory and memory wait states will also affect the descriptor load delay.

- Wait states: Memory accesses can have a wait state associated with them. These wait states need to be accounted into the calculation of throughput.
- Channel arbitration: Some time channels are not immediately made active after reception of trigger. This is due to other active channels in the system. This can lead to multiple cycles being lost before the channel is even made active.
- Preemption: The choice of making a DMA channel preemptable impacts its performance. This is because every time a channel is prempted:
  - The channel is in a pending state for as long as the higher priority channel is running
  - On resumption, the channel descriptor needs to be fetched again. This is additional cycles for every resume. So if there are a large number of high-priority channels, making a low-priority channel preemptible can have adverse effects on its throughput. On the other hand, if there is a low-priority channel that is transferring a large amount of data, then not making it preemptable can starve other high-priority channels for too long.

  Sometimes, users can also distribute channels across multiple DW blocks to avoid conditions of preemption and deal with the contention at the bus arbitration level.
- Bus arbitration: Several bus masters access the bus, including the CPU cores and multiple DMA (DW) and DMAC. This makes any access to data movement over the bus subject to arbitration with other masters. Actions such as fetching the descriptor or data can be stalled by arbitration. The arbitration of the bus is based on the arbitration scheme configured in PROT_SMPU_MSx_CTL[PRIO]
- Transfer width: The width of the transfer configured by the Data_size parameter in the descriptor is important in the transfer throughput calculation. 32-bit transfers are four times faster than 8-bit transfers.

11. DMAC Controller (DMAC)

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The DMAC transfers data to and from memory, peripherals, and registers. These transfers occur independent from the CPU. The DMAC can be configured to perform multiple independent data transfers. All data transfers are managed by a channel. There can be up to 32 channels in the DMAC. The number of channels in the DMAC controller can vary with devices. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for the number of channels supported in the device. A channel has an associated priority; channels are arbitrated according to their priority. The main difference between the DMA (DW) and DMAC relate to their usage. The DMA (DW) is meant as a small data size, transactional DMA, which would typically be used to transfer bytes between peripherals such as, from ADC to RAM. Using the DMA (DW) for large transaction is expensive on a system due to its relatively low performance. The DMAC is a transitional DMA. It is more efficient than the DMA (DW) and should be used to transfer large amounts of data. The DMAC has dedicated channel logic for all channels. Furthermore, the DMAC also includes a 12-byte FIFO for temporary data storage. This results in increased memory bandwidth for the DMAC. The DMAC also supports an additional transfer mode called memory copy and scatter transfer.

11.1 Features

The DMAC controller has the following features:

- Supports up to 32 channels per DMAC controller; see the PSoC 61 datasheet/PSoC 62 datasheet for details
- Four levels of priority for each channel
- Descriptors are defined in memory and referenced to the respective channels
- Supports single, 1D, 2D, Memory-copy, or Scatter-transfer modes for a descriptor
- Supports transfer of up to 2^32 data elements per descriptor
- Configurable source and destination address increments
- Supports 8-bit, 16-bit, and 32-bit data widths at both source and destination
- Configurable input trigger behavior for each descriptor
- Configurable interrupt generation in each descriptor
- Configurable output trigger generation for each descriptor
- Descriptors can be chained to other descriptors in memory

11.2 Architecture

A data transfer is initiated by an input trigger. This trigger may originate from the source peripheral of the transfer, the destination peripheral of the transfer, CPU software, or from another peripheral. Triggers provide Active/Sleep functionality and are not available in Deep Sleep and Hibernate power modes.

The data transfer details are specified by a descriptor. Among other things, this descriptor specifies:

- The source and destination address locations and the size of the transfer.
- The actions of a channel; for example, generation of output triggers and interrupts. See the Interrupts chapter on page 56 for more details.
- Data transfer types can be single, 1D, or 2D as defined in the descriptor structure. These types define the address sequences generated for source and destination. 1D and 2D transfers are used for “scatter gather” and other useful transfer operations.

11.3 Channels

The DMAC controller supports multiple independent data transfers that are managed by a channel. Each channel connects to a specific system trigger through a trigger multiplexer that is outside the DMAC controller.

Channel priority: A channel is assigned a priority (CHi_CTL.PRIO) between 0 and 3, with 0 being the highest priority and 3 being the lowest priority. Channels with the same priority constitute a priority group. Priority decoding determines the highest priority pending channel, which is determined as follows.

- The highest priority group with pending channels is identified first.
- Within this priority group, round-robin arbitration is applied. Round-robin arbitration within a priority group gives the highest priority to the lower channel indices in the priority group.

The data transfer associated with a trigger is made up of one or more ‘atomic transfers’ or ‘single transfers’; see Table 11-1 for a better understanding. A single trigger could be configured to transfer multiple “single transfers”.

Channel Registers:

- CH_CTL. This register provides generic channel control information.
- CH_CURR_PTR. This register provides the memory location address where the current descriptor is located. Software needs to initialize this register. Hardware sets this register to the current descriptor’s next descriptor pointer, when advancing from the current descriptor to the next descriptor in a chained descriptor list. When this field is “0”, there is no valid descriptor.
- CH_IDX. This register provides the current X and Y indices of the channel into the current descriptor. Software needs to initialize this register. Hardware sets the X and Y indices to 0, when advancing from the current descriptor to the next descriptor in a descriptor list.
- CH_SRC. This register provides the current address of the source location.
- CH_DST. This register provides the current address of the destination location.
- CH_DESCR_STATUS. This register provides the validity of other CH_DESCR registers.
- CH_DESCR_CTL. This register contains a copy of DESCR_CTL of the currently active descriptor.
- CH_DESCR_SRC. This register contains a copy of DESCR_SRC of the currently active descriptor.
- CH_DESCR_DST. This register contains a copy of DESCR_DST of the currently active descriptor.
- CH_DESCR_X_INCR. This register contains a copy of DESCR_X_INCR of the currently active descriptor.
- CH_DESCR_X_SIZE. This register contains a copy of DESCR_X_SIZE of the currently active descriptor.
- CH_DESCR_Y_INCR. This register contains a copy of DESCR_Y_INCR of the currently active descriptor.
- CH_DESCR_Y_SIZE. This register contains a copy of DESCR_Y_SIZE of the currently active descriptor.
- CH_DESCR_NEXT_PTR. This register contains a copy of DESCR_NEXT_PTR of the currently active descriptor.
- INTR. This register contains the interrupts that are currently activated for this channel
- INTR_SET. Writing ‘1’ to the appropriate bit in this register sets the corresponding INTR field to 1.
- INTR_MASK. Mask for corresponding field in the INTR register.
- INTR_MASKED. Logical AND of corresponding INTR and INTR_MASK fields.

Note that channel state is retained in Deep Sleep power mode.

A channel has three access control attributes that are used by the shared memory protection units (SMPUs) and peripheral protection units (PPUs) for access control. These fields are typically inherited from the master that modified the channel’s control register.

- The Privileged Mode (CHi_CTL.P) attribute can be set to privileged or user.
- The Non-secure (CHi_CTL.NS) attribute can be set to secure or non-secure.
- The Protection context (CHi.CTL.PC) attribute can be set to one of the protection contexts.

11.3.1 Channel Interrupts

Every DMAC channel has an interrupt line associated with it. The INTR_TYPE parameter in the descriptor determines the event that will trigger the interrupt for the channel. In addition, each DMAC channel has INTR, INTR_SET, INTR_MASK, and INTR_MASKED registers to control their respective interrupt lines. INTR_MASK can be used to mask the interrupt from the DMA channel. The INTR and INTR_SET can be used to clear and set the interrupt, respectively, for debug purposes.

The DMAC_CHx_INTR registers allow the user to set different interrupt causes. The following causes are available to configure:

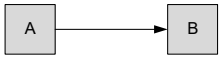
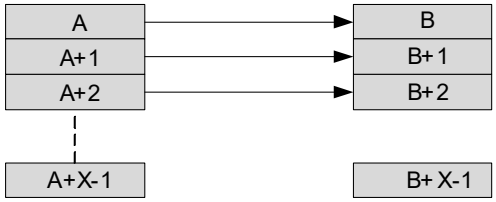
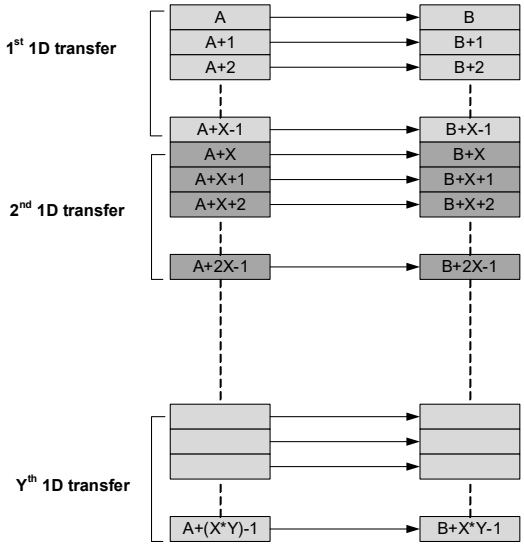
- bit 0: Interrupt based on transfer completion configured based on INTR_TYPE field in the descriptor
- bit 1: Source bus error
- bit 2: Destination bus error
- bit 3: Misaligned source address
- bit 4: Misaligned destination address
- bit 5: Current descriptor pointer is null
- bit 6: Active channel is in disabled state
- bit 7: Descriptor bus error

For error-related interrupt causes (every state other than Completion), the channel is disabled (hardware sets CH_CTL.ENABLED to ‘0’).

11.4 Descriptors

The data transfer between a source and destination in a channel is configured using a descriptor. Descriptors are stored in memory. The descriptor pointer is specified in the DMAC channel registers. The DMAC controller does not modify the descriptor and treats it as read only. A descriptor is a set of up to six 32-bit registers that contain the configuration for the transfer in the associated channel. There are three types of descriptors.

Table 11-1. Descriptor Types

Descriptor Type	Description
Single transfer	Transfers a single data element 
1D transfer	Performs a one-dimensional “for loop”. This transfer is made up of X number of single transfers 
2D transfer	Performs a two-dimensional “for loop”. This transfer is made up of Y number of 1D transfers 
Memory copy	This is a special case of 1D transfer; DESCR_X_INCR.SRC_X_INCR and DESCR_X_INCR.DST_X_INCR are implicitly set to 1 and not as part of the descriptor (source and destinations data are consecutive). The descriptor type assumes that both source and destination locations are memory locations. Therefore, 8-bit, 16-bit, and 32-bit transfers can be used (as long as alignment requirements are met), rather than using transfer of a specific type (as specified by SRC_TRANSFER_SIZE and DST_TRANSFER_SIZE). The size of the descriptor is five 32-bit words. DESCR_CTL, DESCR_SRC, DESCR_DST, DESCR_X_SIZE, and DESCR_NEXT_PTR (note that the descriptor size is one word less than the 1D transfer descriptor size). Scatter: This descriptor type is intended to write a set of 32-bit data elements, whose addresses are “scattered” around the address space. The size of the descriptor is four 32-bit words. DESCR_CTL, DESCR_SRC, DESCR_X_SIZE, and DESCR_NEXT_PTR.
Scatter transfer	This descriptor type is intended to write a set of 32-bit data elements, whose addresses are “scattered” around the address space. The size of the descriptor is four 32-bit words. DESCR_CTL, DESCR_SRC, DESCR_X_SIZE, and DESCR_NEXT_PTR.

Single Transfer:

The following pseudo code illustrates a single transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
DST_ADDR[0] = (t_DATA_SIZE) SRC_ADDR[0];
```

1D Transfer:

The following pseudo code illustrates a 1D transfer. Note that the 1D transfer is represented by a loop with each iteration executing a single transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
DST_ADDR[X_IDX * DST_X_INCR] =
          (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR];
}
```

2D Transfer:

The following pseudo code illustrates a 2D transfer. Note that the 2D transfer is represented by a loop with each iteration executing an inner loop, which is the 1D transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
for (Y_IDX = 0; Y_IDX <= Y_COUNT; Y_IDX++) {
    for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
DST_ADDR[X_IDX * DST_X_INCR + Y_IDX * DST_Y_INCR ] =
          (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR + Y_IDX * SRC_Y_INCR];
    }
}
```

Memory copy:

The following pseudo code illustrates a memory copy.

```
// DST_ADDR is a pointer to an object of type uint8_t
// SRC_ADDR is a pointer to an object of type uint8_t
// This transfer type uses 8-bit, 16-bit an 32-bit transfers. The HW ensures that
// alignment requirements are met.
for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
DST_ADDR[X_IDX] = SRC_ADDR[X_IDX];
}
```

Scatter:

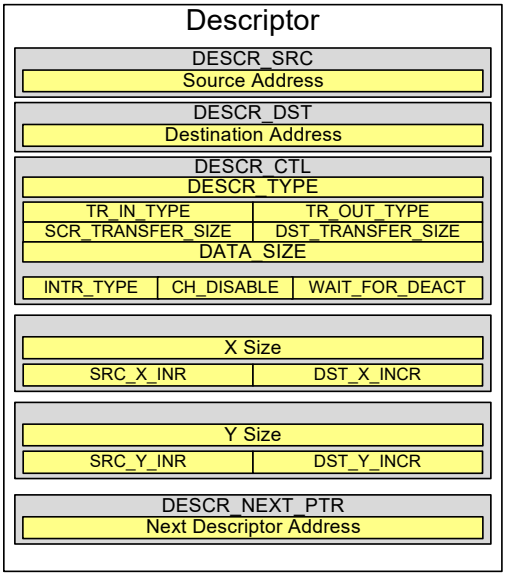
The following pseudo code illustrates the scatter transfer

```
// SRC_ADDR is a pointer to an object of type uint32_t
for (X_IDX = 0; X_IDX < X_COUNT; X_IDX += 2) {
           address = SRC_ADDR[X_IDX];
           data      = SRC_ADDR[X_IDX + 1];
           *address = data;
}
```

The parameters in the descriptor help configure the different aspects of the transfers explained.

Figure 11-2 shows the structure of a descriptor.

Figure 11-1. Descriptor Structure



11.4.1 Address Configuration

Source and Destination Address: The source and destination addresses are set in the respective registers in the descriptor. These set the base addresses for the source and destination location for the transfer. In case the descriptor is configured to transfer a single element, this field holds the source/destination address of the data element. If the descriptor is configured to transfer multiple elements with source address or destination address or both in an incremental mode, this field will hold the address of the first element that is transferred.

DESCR_TYPE: This field configures whether the descriptor has a single, 1D, 2D type, memory copy, or scatter transfer.

Source data prefetch, DATA_PREFETCH: When enabled, source data transfers are initiated as soon as the channel is enabled; the current descriptor pointer is not “0” and there is space available in the channel's data FIFO. When the input trigger is activated, the trigger can initiate destination data transfers with data that is already in the channel's data FIFO. This effectively shortens the initial delay of the data transfer.

Note: Data prefetch should be used with care to ensure that data coherency is guaranteed and that prefetches do not cause undesired side effects.

Trigger input type, TR_IN_TYPE: This field determines how the DMAC engine responds to input trigger signal. This field can be configured for one of the following modes:

- Type 0: A trigger results in execution of a single transfer. Regardless of the DESCR_TYPE setting, a trigger input will trigger only a single element transfer. For example, in a 1D transfer, the DMAC will transfer only one data element in every trigger.
- Type 1: A trigger results in the execution of a single 1D transfer. If the DESCR_TYPE was set to single transfer, the trigger signal will trigger the single transfer specified by the descriptor. For a DESCR_TYPE set to 1D transfer, the trigger signal will trigger the entire 1D transfer configured in the descriptor. For a 2D transfer, the trigger signal will trigger only a single iteration of the Y loop transfer. If the descriptor type is “memory copy”, the trigger results in the execution of a memory copy transfer.
- Type 2: A trigger results in execution of the current descriptor. Regardless of DESCR_TYPE, the trigger will execute the entire descriptor. If there was a next descriptor configured for the current descriptor, this trigger setting will not automatically trigger the next descriptor.
- Type 3: A trigger results in execution of the current descriptor and also triggering the next descriptor. The execution of the next descriptor from this point will be determined by the TR_IN_TYPE setting of the next descriptor.

Trigger out type, TR_OUT_TYPE: This field determines what completion event will generate the output trigger signal. This field can be configured to one of the following modes:

- Type 0: Generates a trigger output for completion of every single element transfer.
- Type 1: Generates a trigger output for completion of a 1D transfer. If the descriptor type is “memory copy”, the output trigger is generated after the execution of a memory copy transfer. If the descriptor type is “scatter”, the output trigger is generated after the execution of a scatter transfer.
- Type 2: Generates a trigger output for completion of the current descriptor. This trigger output is generated independent of the state of the DESCR_NEXT_PTR.
- Type 3: Generates a trigger output on completion of the current descriptor, when the current descriptor is the last descriptor in the descriptor chain. This means a trigger is generated when the descriptor execution is complete and the DESCR_NEXT_PTR is ‘0’.

Interrupt Type, INTR_TYPE: This field determines which completion event will generate the output interrupt signal. This field can be configured to one of the following modes:

- Type 0: Generates an interrupt output for completion of every single element transfer.
- Type 1: Generates an interrupt output for completion of a 1-D transfer. If the descriptor type is “memory copy”, the interrupt is generated after the execution of a memory copy transfer. If the descriptor type is “scatter” the interrupt is generated after the execution of a scatter transfer.
- Type 2: Generates an interrupt output for completion of the current descriptor. This interrupt output is generated independent of the state of the DESCR_NEXT_PTR.
- Type 3: Generates an interrupt output on completion of the current descriptor, when the current descriptor is the last descriptor in the descriptor chain. This means an interrupt is generated when the descriptor execution is complete and the DESCR_NEXT_PTR is ‘0’.

WAIT_FOR_DEACT: When the DMAC transfer based on the TR_IN_TYPE is completed, the data transfer engine checks the state of trigger deactivation. The data transfer on the second trigger is initiated only after deactivation of the first. The WAIT_FOR_DEACT parameter will determine when the trigger signal is considered deactivated. The first DMAC transfer is activated when the trigger is activated, but the transfer is not considered complete until the trigger is deactivated. This field is used to synchronize the controller’s data transfers with the agent that generated the trigger. This field has four settings:

- 0 – Pulse Trigger: Do not wait for deactivation.
- 1 – Level-sensitive waits four SYSCLK cycles: The DMAC trigger is deactivated after the level trigger signal is detected for four cycles.
- 2 – Level-sensitive waits 16 SYSCLK cycles: The DMAC transfer is initiated if the input trigger is seen to be activated after 16 clock cycles.
- 3 – Pulse trigger waits indefinitely for deactivation. The DMAC transfer is initiated after the trigger signal deactivates. The next transfer is initiated only if the trigger goes low and then high again.

X Size: This field determines the number of single element transfers present in the X loop (inner loop). This field is valid when the DESCR_TYPE is set to 1D or 2D transfer. For the “memory copy” descriptor type, (X_COUNT + 1) is the number of transferred Bytes. For the “scatter” descriptor type, ceiling (X_COUNT/2) is the number of (address, write data) initialization pairs processed.

Source Address Increment (X loop) (SCR_X_INCR): This field configures the index by which the source address is to be incremented for every iteration in an X loop. The field is expressed in multiples of SRC_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the source address does not need to be incremented, you can set this parameter to zero.

Destination Address Increment (X loop) (DST_X_INCR): This field configures the index by which the destination address is to be incremented, for every iteration in an X loop. The field is expressed in multiples of DST_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the destination address does not need to be incremented, you can set this parameter to zero.

Y Size: This field determines the number of 1-D transfers present in the Y loop (outer loop). This field is valid when the DESCR_TYPE is set to 2-D transfer.

Source Address Increment (Y loop) (SCR_Y_INCR): This field configures the index by which the source address is to be incremented, for every iteration in a Y loop. The field is expressed in multiples of SRC_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the source address does not need to be incremented, you can set this parameter to zero.

Destination Address Increment (X loop) (DST_Y_INCR): This field configures the index by which the destination address is to be incremented, for every iteration in a Y loop. The field is expressed in multiples of DST_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the destination address does not need to be incremented, you can set this parameter to zero.

Channel Disable (CH_DISABLE): This field specifies whether the channel is disabled or not after completion of the current descriptor (independent of the value of the DESCR_NEXT_PTR). A disabled channel will ignore its input triggers.

11.4.2 Transfer Size

The word width for a transfer can be configured using the transfer/data size parameter in the descriptor. The settings are diversified into source transfer size, destination transfer size, and data size. The data size parameter (DATA_SIZE) sets the width of the bus for the transfer. The source and destination transfer sizes set by SCR_TRANSFER_SIZE and DST_TRANSFER_SIZE can have a value either the DATA_SIZE or 32 bit. DATA_SIZE can have a 32-bit, 16-bit, or 8-bit setting.

The source and destination transfer size for the DMAC must match the addressable width of the source and destination, regardless of the width of data that must be moved. The DATA_SIZE parameter will correspond to the width of the actual data. For example, if a 16-bit PWM is used as a destination for DMAC data, the DST_TRANSFER_SIZE must be set to 32 bit to match the width of the PWM register, because the peripheral register width for the TCPWM block (and most PSoC 6 MCU peripherals) is always 32-bit wide. However, in this example the DATA_SIZE for the destination may still be set to 16 bit because the 16-bit PWM only uses two bytes of data. SRAM and Flash are 8-bit, 16-bit, or 32-bit addressable and can use any source and destination transfer sizes to match the needs of the application.

Table 11-2 summarizes the possible combinations of the transfer size settings and its description.

Table 11-2. Transfer Size Settings

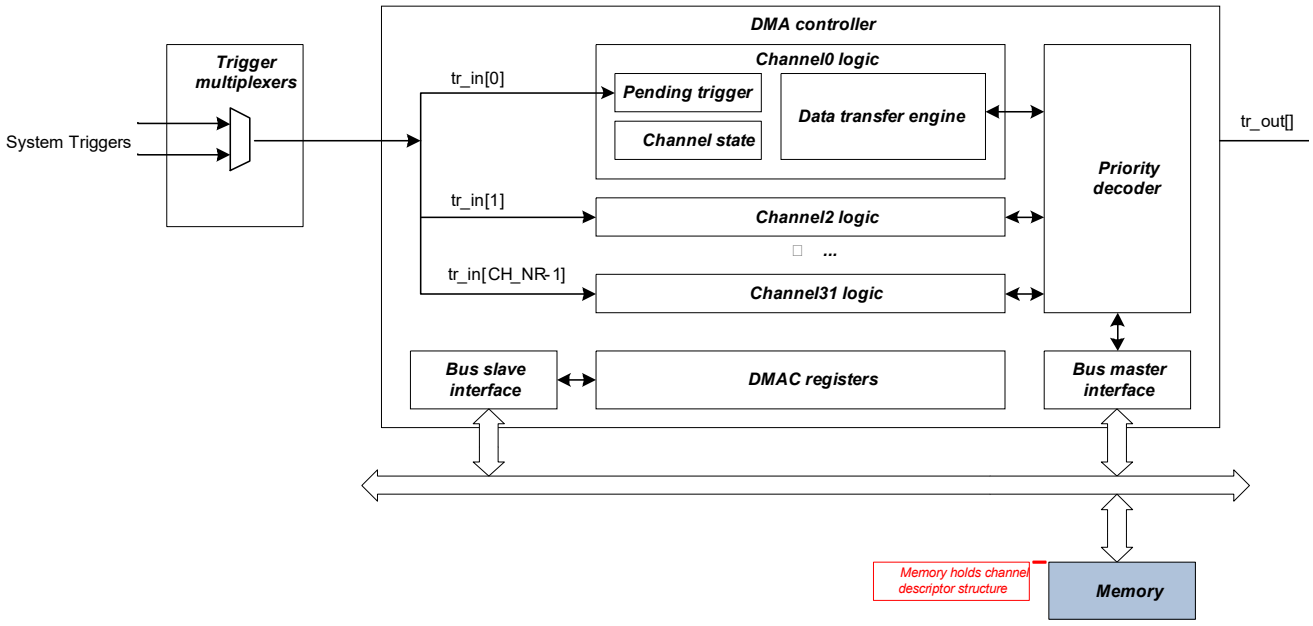
DATA_SIZE	SCR_TRANSFER_SIZE	DST_TRANSFER_SIZE	Description
8-bit	8-bit	8-bit	No data manipulation
8-bit	32-bit	8-bit	Higher 24 bits from the source dropped
8-bit	8-bit	32-bit	Higher 24 bits zero padded at destination
8-bit	32-bit	32-bit	Higher 24 bits from the source dropped and higher 24 bits zero padded at destination
16-bit	16-bit	16-bit	No data manipulation
16-bit	32-bit	16-bit	Higher 16 bits from the source dropped
16-bit	16-bit	32-bit	Higher 16 bits zero padded at destination
16-bit	32-bit	32-bit	Higher 16 bits from the source dropped and higher 16-bit zero padded at destination
32-bit	32-bit	32-bit	No data manipulation

11.4.3 Descriptor Chaining

Descriptors can be chained together. The DESCR_NEXT_PTR field contains a pointer to the next descriptor in the chain. A channel executes the next descriptor in the chain when it completes executing the current descriptor. The last descriptor in the chain has DESCR_NEXT_PTR set to ‘0’ (NULL pointer). A descriptor chain is also referred to as a descriptor list. It is possible to have a circular list; in a circular list, the execution continues indefinitely until there is an error or the channel or the controller is disabled by user code.

11.5 DMAC Controller

Figure 11-2. DMAC Controller Overview



11.5.1 Trigger Selection

Trigger signals can be generated from different sections of the chips. A trigger multiplexer block helps route these trigger signals to the destination. The DMAC is one such destination of triggers. The trigger multiplexer block is outside the DMAC block and is discussed in the Trigger Multiplexer Block chapter on page 294.

11.5.2 Channel Logic

The channel logic keeps track of pending triggers for each channel and initiates the transfer corresponding to the active descriptor, based on availability of the bus and arbitration by priority decoder.

The **priority decoder** determines the highest priority pending channel.

Master I/F is an AHB-Lite bus master that allows the DMAC controller to initiate AHB-Lite data transfers to the source and destination locations as well as to read the descriptor from memory.

Slave I/F is an AHB-Lite bus slave that allows the main CPU to access DMAC controller control/status registers.

11.5.3 Output Triggers

Each channel has an output trigger. This trigger is high for two slow clock cycles. The trigger is generated on the completion of a data transfer. At the system level, these output triggers can be connected to the trigger multiplexer component. This connection allows a DMAC controller output trigger to be connected to a DMAC controller input trigger. In other words, the completion of a transfer in one channel can activate another channel or even reactivate the same channel.

DMAC output triggers routing to other DMAC channels or other peripheral trigger inputs is achieved using the trigger multiplexer. Refer to the Trigger Multiplexer Block chapter on page 294.

12. Cryptographic Function Block (Crypto)

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The Cryptographic block (Crypto) provides hardware implementation and acceleration of cryptographic functions. Implementation in hardware takes less time and energy than the equivalent firmware implementation. In addition, the block provides True Random Number generation functionality in silicon, which is not available in firmware.

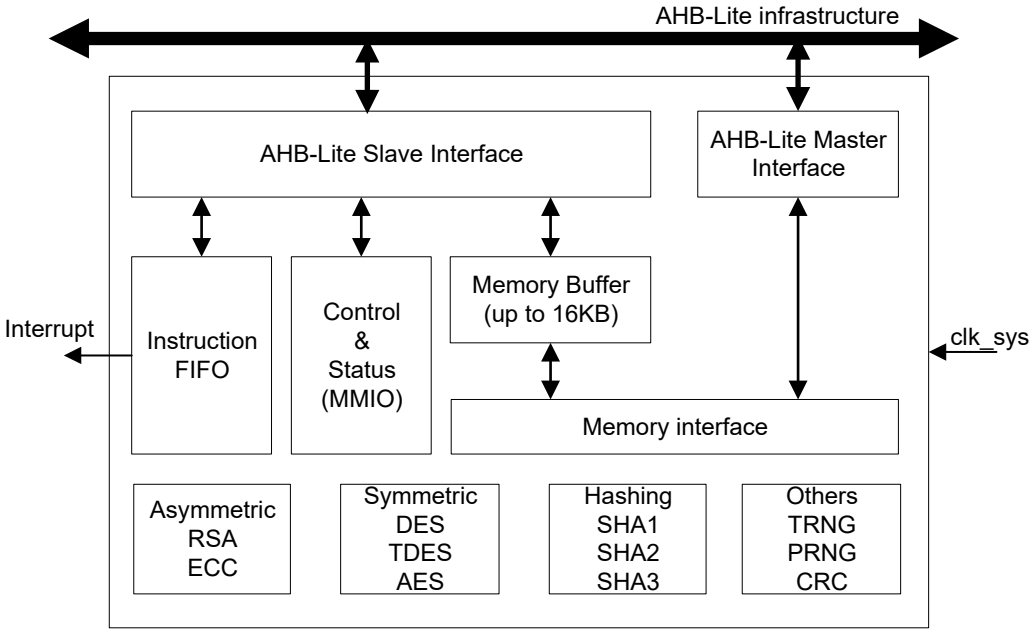
12.1 Features

- Advanced encryption standard (AES)
- Data Encryption and Triple Data Encryption Standards (DES, TDES)
- Secure Hash Algorithm (SHA)
- Cyclic redundancy checking (CRC)
- Pseudo random number generator (PRNG)
- True random number generator (TRNG)
- Vector unit (VU) to support asymmetric key cryptography, such as RSA and ECC.

12.2 Architecture

The following figure gives an overview of the cryptographic block.

Figure 12-1. Crypto Block Diagram



The crypto block has the following interfaces:

- An AHB-Lite slave interface connects the block to the AHB-Lite infrastructure. This interface supports 8/16/32-bit AHB-Lite transfers. MMIO register accesses are 32-bit only. Memory buffer accesses can be 8/16/32-bit.
- An AHB-Lite master interface connects the block to the AHB-Lite infrastructure. This interface supports 8/16/32-bit AHB-Lite transfers. The interface enables the crypto block to access operation operand data from system memories, such as Flash or SRAM.
- A single interrupt signal “interrupt” is used to signal the completion of an operation.
- A clock signal “clk_sys” interface connects to the SRSS.

The block has the following components:

- An AHB-Lite slave interface.
- An AHB-Lite master interface.
- A component that contains MMIO control and status registers.
- A memory buffer, for internal operation operand data.
- A memory interface that directs operation operand data requests to either the block internal memory buffer or to the AHB-Lite master interface.
- An instruction controller component that decodes instructions from an instruction FIFO. The controller issues the instructions to the function specific components.
- Cryptographic function specific components.

The following sections explain each component in detail.

12.3 Instruction Controller

The instruction controller consists of an instruction FIFO, an instruction decoder, and a general-purpose register file.

- The instruction FIFO is software-programmable through MMIO registers that are accessed through the AHB-Lite slave interface. Software writes instructions and instruction operand data to the instruction FIFO. The FIFO consists of eight 32-bit FIFO entries.
- The instruction decoder decodes the instructions (and the associated instruction operand data) from the instruction FIFO. The instruction decoder issues the decoded instruction to a specific functional component. The functional component is responsible for instruction execution.
- The general-purpose register file consists of sixteen 32-bit registers. An instruction specifies the specific use of register-file registers. Registers are used to specify instruction operand data, such as memory locations (for example, for DES, AES, and SHA instructions) or immediate data operations (for example, for vector unit instructions).

12.3.1 Instructions

An instruction consists of a sequence of one, two, or three instruction words. Most instructions are encoded by a single instruction word.

The instruction FIFO can hold up to eight 32-bit instruction words. A CPU writes instruction words to the instruction FIFO (INSTR_FF_WR register) and the crypto block decodes the instruction words to execute the instructions. INSTR_FF_STATUS.USED specifies how many of the eight instruction FIFO entries are used. The instruction FIFO decouples the progress of CPU execution from the crypto block execution: the CPU can write new instruction words to the FIFO, while the block executes previously written instructions.

There are multiple interrupt causes associated with the instruction FIFO and the instruction decoder:

- The INTR.INSTR_FF_OVERFLOW interrupt cause is activated on a write to a full instruction FIFO.
- THE INTR.INSTR_FF_LEVEL interrupt cause is activated when the number of used FIFO entries (INSTR_FF_STATUS.USED) is less than a specified number of FIFO entries (INSTR_FF_CTL.LEVEL).
- The INTR.INSTR_OPC_ERROR interrupt cause is activated when an instruction's operation code is not defined.
- The INTR.INSTR_CC_ERROR interrupt cause is activated when a vector unit instruction has an undefined condition code.

Most instructions perform specific cryptographic functionality. For example, the AES instruction performs and Advanced Encryption Standard (AES) block cipher operation. Some instructions perform more generic functionality: most generic instructions move operand data between different locations. Higher level symmetric cipher and hash functionality is implemented using a combination of cryptographic instructions and generic instructions. Higher level asymmetric cipher functionality is implemented using a set of vector unit (VU) instructions.

12.3.2 Instruction Operands

The instruction operands are found in one of the following locations:

- System memory
- Memory buffer
- Instruction FIFO instruction words
- Load and store FIFOs
- Register buffer
- Vector unit register-file

System memory. The system memory includes all memory-mapped memories attached to the bus infrastructure that are accessible by the crypto block through the master bus interface.

Memory buffer. The crypto block memory buffer is an internal SRAM with a capacity of up to 16 KB. This internal SRAM provides better latency and bandwidth characteristics than the system memory. Therefore, frequently accessed vector unit instruction operand data is typically located in the memory buffer.

Both the block's external system memory and internal memory buffer are accessed through the same memory interface component. The access address specifies if the access is to the system memory or the internal memory buffer (also see VU_CTL.ADDR[31:14]).

External bus masters can access both the system memory and the crypto's internal memory. The external bus masters access the internal memory through the slave bus interface.

Instruction FIFO. For some instructions, immediate operand data is provided by the instruction words. The limited 32-bit instruction words only allow for limited immediate operand data.

Load and store FIFOs. Most instructions have stream-like operand data: sequences of bytes that are specified by the access address of the start byte. The two load FIFOs provide access to source operand data and the single store FIFO provides access to destination operand data. Typically, vector unit instruction operand data is "streamed" from the crypto block memory buffer.

Register buffer. Most symmetric and hash cryptographic instructions benefit from a large (2048-bit) register buffer. This register buffer provides access flexibility that is not provided by the load and store FIFOs. The register buffer is shared by different instructions to amortize its cost (silicon area). After an Active reset or a crypto block reset (CTL.ENABLED), the register buffer is set to '0'.

Vector unit register file. Most vector unit instructions perform large integer arithmetic functionality. For example, the VU ADD instruction can add two 4096-bit numbers. Typically, operand data is "streamed" from the memory buffer. In addition, a vector unit register file with sixteen registers is provided. Each register specifies the location of a number (start word access address) and the size of the number (the size is in bits).

12.3.3 Load and Store FIFO Instructions

The load and store FIFOs provide access to operand data in a "streaming" nature. Operand data is streamed through the memory interface from or to either the internal memory buffer or the system memory.

Two independent load FIFOs provide access to streamed source operand data. Each FIFO has a multi-byte buffer to prefetch operand data.

One store FIFO provides access to streamed destination operand data. The FIFO has a multi-byte buffer to temporarily hold the data before it is written to the memory interface.

Streamed operand data is specified by a memory start address and an operand size in bytes.

Three FIFO instructions are supported: FF_START, FF_STOP, and FF_CONTINUE. The FF_START and FF_STOP instructions are supported for both the load and store FIFOs. The FF_CONTINUE instruction is only supported for the load FIFOs. The FF_START and FF_CONTINUE instructions consist of three instruction words. The FF_STOP instruction consists of a single instruction word.

Table 12-1. FF_START Instruction

Instruction Format	FF_START (ff_identifier, address[31:0], size[31:0])	
Encoding	IW0[31:24] = "operation code" IW0[3:0] = ff_identifier // "8": load FIFO 0, // "9": load FIFO 1, // "12": store FIFO IW1[31:0] = address[31:0] IW2[31:0] = size[31:0]	
Mnemonic	**Operation Code**	**Functionality**
FF_START	0x70	Clear the FIFO multi-byte buffer. Start streaming size[31:0] operand data bytes, starting at address[31:0]. This instruction is supported by both load and store FIFOs. For load FIFOs, data bytes are read through the memory interface. For the store FIFO, data bytes are written to the memory interface. The INSTR_OPC_ERROR interrupt cause is set when the ff_identifier is not a legal value.

Table 12-2. FF_STOP instruction

Instruction Format	FF_STOP (ff_identifier)	
Encoding	IW0[31:24] = “operation code” IW0[3:0] = ff_identifier // “8”: load FIFO 0, // “9”: load FIFO 1, // “12”: store FIFO	
Mnemonic	**Operation Code**	**Functionality**
FF_STOP	0x72	Stop streaming This instruction is supported by both load and store FIFOs. For load FIFOs, the multi-byte buffer is cleared. For the store FIFO, the multi-byte buffer is written to the memory interface; that is, the buffer is flushed. The INSTR_OPC_ERROR interrupt cause is set when the ff_identifier is not a legal value.

Table 12-3. FF_CONTINUE Instruction

Instruction Format	FF_CONTINUE (ff_identifier, address[31:0], size[31:0])	
Encoding	IW0[31:24] = “operation code” IW0[3:0] = ff_identifier // “8”: load FIFO 0, // “9”: load FIFO 1, IW1[31:0] = address[31:0] IW2[31:0] = size[31:0]	
Mnemonic	**Operation Code**	**Functionality**
FF_CONTINUE	0x71	Do not clear the FIFO multi-byte buffer. Continue streaming size[31:0] operand data bytes, starting at address[31:0]. This instruction can only be started when a previous FF_START or FF_CONTINUE instruction for the same load FIFO has read all its operand data bytes from the memory interface (see LOAD01/_FF_STATUS.BUSY). This instruction is only supported by the load FIFOs. The INSTR_OPC_ERROR interrupt cause is set when the ff_identifier is not a legal value.

The status of the load and store FIFOs is provided through the LOAD0_FF_STATUS, LOAD1_FF_STATUS, and STORE_FF_STATUS registers.

12.3.4 Register Buffer Instructions

The 2048-bit register buffer has two 1024-bit partitions: reg_buff[1023:0] and reg_buff[2047:1024]. The reg_buff[1023:0] partition has eight 128-bit subpartitions:

- block0[127:0] = reg_buff[0*128+127:0*128]
- block1[127:0] = reg_buff[1*128+127:1*128]
- block2[127:0] = reg_buff[2*128+127:2*128]
- block3[127:0] = reg_buff[3*128+127:3*128]
- block4[127:0] = reg_buff[4*128+127:4*128]
- block5[127:0] = reg_buff[5*128+127:5*128]
- block6[127:0] = reg_buff[6*128+127:6*128]
- block7[127:0] = reg_buff[7*128+127:7*128]

The reg_buff[1023:0] partition consists of 128 bytes: Byte offset 0 identifies reg_buff[7:0] and Byte offset 127 identifies reg_buff[1023:1016].

Some instructions work on the complete register buffer and some instructions work on 128-bit subpartitions.

Table 12-4. CLEAR Instruction

Instruction Format	CLEAR ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
CLEAR	0x64	reg_buff[2047:0] = 0; This instruction is used to set the register buffer to ‘0’. This instruction is useful to prevent information leakage from the register buffer. The instruction also sets DEV_KEY_STATUS.LOADED to '0'.

Table 12-5. SWAP Instruction

Instruction Format	SWAP ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**

SWAP	0x65	temp = reg_buff[1023:0];
		reg_buff[1023:0] = reg_buff[2047:1024];
		reg_buff[2047:1024] = temp;
		This instruction swaps/exchanges the two register buffer partitions.

Table 12-6. XOR Instruction

Instruction Format	XOR (offset[6:0], size[7:0])	
Encoding	IW[31:24] = “operation code” IW[14:8] = offset IW[7:0] = size // in the range [0,128]	
Mnemonic	**Operation Code**	**Functionality**

XOR	0x66	data = GetFifoData (LOAD0_FIFO, size);
		data = data << (offset*8);
		reg_buff[1023:0] = reg_buff[1023:0] ^ data;
		This instruction always uses load FIFO 0.
		Note: This instruction can only access the lower register buffer partition.

Table 12-7. STORE Instruction

Instruction Format	STORE (offset[6:0], size[7:0])	
Encoding	IW[31:24] = “operation code” IW[14:8] = offset IW[7:0] = size // in the range [0,128]	
Mnemonic	**Operation Code**	**Functionality**
STORE	0x67	data = reg_buff[1023:0] data = data >> (offset*8); SetFifoData (STORE_FIFO, size, data)

Table 12-8. BYTE_SET Instruction

Instruction Format	BYTE_SET (offset[6:0], byte[7:0])	
Encoding	IW[31:24] = “operation code” IW[14:8] = offset IW[7:0] = byte	
Mnemonic	**Operation Code**	**Functionality**
BYTE_SET	0x68	reg_buff[offset*8 + 7: offset*8] = byte;

Some instructions work on (up to) 128-bit subpartitions or blocks. In addition, these instructions can work on the load and store FIFOs. The instructions' source and destination operand identifiers are encoded as follows:

- 0: block0[127:0] = reg_buff[0*128+127:0*128]
- 1: block1[127:0] = reg_buff[1*128+127:1*128]
- 2: block2[127:0] = reg_buff[2*128+127:2*128]
- 3: block3[127:0] = reg_buff[3*128+127:3*128]
- 4: block4[127:0] = reg_buff[4*128+127:4*128]
- 5: block5[127:0] = reg_buff[5*128+127:5*128]
- 6: block6[127:0] = reg_buff[6*128+127:6*128]
- 7: block7[127:0] = reg_buff[7*128+127:7*128]
- 8: load FIFO 0
- 9: load FIFO 1
- 12: store FIFO

Table 12-9. BLOCK_MOV Instruction

Instruction Format	BLOCK_MOV (reflect[1], size[3:0], dst[3:0], src0[3:0])	
Encoding	IW[31:24] = “operation code” IW[23] = reflect IW[19:16] = size IW[15:12] = dst IW[3:0] = src0	
Mnemonic	**Operation Code**	**Functionality**

BLOCK_MOV	0x40	size = (size == 0) ? 16 : size;
		data1 = data0 = GetBlock (src0, size);
		if (reflect) { // assume size of 16 B / 128 bit
		  temp = data0;
		  for (i = 0; i < 16; i += 1) {
		  for (j = 0; j < 8; j += 1) {
		  // reflection of bits in a byte
		  data1[8*i + j] = data0[8*i + 7-j];
		  }
		  }
		}
		SetBlock (dst, size, data1);

Table 12-10. BLOCK_XOR Instruction

Instruction Format	BLOCK_XOR (size[3:0], dst[3:0], src1[3:0], src0[3:0])	
Encoding	IW[31:24] = “operation code” IW[19:16] = size IW[15:12] = dst IW[7:4] = src1 IW[3:0] = src0	
Mnemonic	**Operation Code**	**Functionality**

BLOCK_XOR	0x41	size = (size == 0) ? 16 : size;
		data0 = GetBlock (src0, size);
		data1 = GetBlock (src1, size);
		SetBlock (dst, size, data0 ^ data1);

Table 12-11. BLOCK_SET Instruction

Instruction Format	BLOCK_SET (size[3:0], dst[3:0], byte[7:0])	
Encoding	IW[31:24] = “operation code” IW[19:16] = size IW[15:12] = dst IW[7:0] = byte	
Mnemonic	**Operation Code**	**Functionality**
BLOCK_SET	0x42	size = (size == 0) ? 16 : size; SetBlock (dst, size, {16{byte[7:0]}})

Table 12-12. BLOCK_CMP Instruction

Instruction Format	BLOCK_CMP (size[3:0], src1[3:0], src0[7:0])	
Encoding	IW[31:24] = “operation code” IW[19:16] = size IW[7:4] = src1 IW[3:0] = src0	
Mnemonic	**Operation Code**	**Functionality**

BLOCK_CMP	0x43	size = (size == 0) ? 16 : size;
		data0 = GetSourceBlock (src0, size);
		data1 = GetSourceBlock (src1, size);
		RESULT.DATA |= (data0[size*8-1:0] != data1[size*8-1:0]);

Table 12-13. BLOCK_GCM Instruction

Instruction Format	BLOCK_GCM (when GCM parameter is '1')	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
BLOCK_GCM	0x43	Perform a GCM multiplication: ■ {block2, block1} = XMUL (block1, block3) ■ block1 = GCM_Reduce ({block2, block1}) The instruction uses three specific 128-bit blocks. The XMUL operation is part of the BLOCK_GCM instruction and is a carry-less multiplication. The reduction is specific for the GCM cipher mode.

GetBlock is defined as follows:

```
GetBlock (src, size) {
   switch (src) {
      0: return block0[127:0];
      1: return block1[127:0];
      2: return block2[127:0];
```

```
      3: return block3[127:0];
      4: return block4[127:0];
      5: return block5[127:0];
      6: return block6[127:0];
      7: return block7[127:0];
      default: return GetFifoData (src, size);
   }
}
```

SetBlock is defined as follows:

```
SetBlock (dst, size, data) {
   switch (dst) {
      0: block0[127:0] = data;
      1: block1[127:0] = data;
      2: block2[127:0] = data;
      3: block3[127:0] = data;
      4: block4[127:0] = data;
      5: block5[127:0] = data;
      6: block6[127:0] = data;
      7: block7[127:0] = data;
      default: SetFifoData (dst, size, data);
   }
}
```

The GetFifoData function is defined as loading “size” bytes from a load FIFO. Up to 16 bytes can be loaded as data[127:0]. The first (top) loaded FIFO entry is mapped on data[7:0], the second loaded FIFO entry is mapped on data[15:8], and so on. The SetFifoData function is defined as storing size bytes from a load FIFO. Up to 16 bytes can be stored as data[127:0]. The first stored FIFO entry is mapped on data[7:0], the second stored FIFO entry is mapped on data[15:8], and so on.

12.4 Hash Algorithms

12.4.1 SHA1 and SHA2

The SHA1 and SHA2 functionality includes three hash instructions per the SHA standard (FIPS 180-4).

- The SHA1 instruction provides all SHA1 functionality (SHA1 always uses 512-bit blocks).
- The SHA2_256 instruction provides SHA2 functionality for 512-bit blocks.
- The SHA2_512 instruction provides SHA2 functionality for 1024-bit blocks.

The instructions support different block sizes and hash sizes:

Instruction	Block Size	Hash Size
SHA1	512 bits	160 bits
SHA2_256	512 bits	256 bits
SHA2_512	1024 bits	512 bits

The SHA1 instruction supports a single algorithm with a specific message digest size. The SHA2_256 and SHA2_512 instructions support multiple algorithms with different message digest sizes.

Instruction	Algorithm	Hash Size	Message Digest Size
SHA1	SHA-1	160 bits	160 bits
SHA2_256	SHA-224	256 bits	224 bits
	SHA-256	256 bits	256 bits

Instruction	Algorithm	Hash Size	Message Digest Size
SHA2_512	SHA-384	512 bits	384 bits
	SHA-512	512 bits	512 bits
	SHA-512/224	512 bits	224 bits
	SHA-512/256	512 bits	256 bits

A SHA algorithm calculates a fixed-length hash value from a variable length message. The hash value is used to produce a message digest or signature. It is computationally impossible to change the message without changing the hash value. The algorithm is stateless: a given message always produces the same hash value. To prevent “replay attacks”, a counter may be included in the message.

The variable length message must be preprocessed: a ‘1’ bit must be appended to the message followed by ‘0’s and a bit size field. The preprocessed message consists of an integer multiple of 512 bit or 1024 bit blocks. The SHA component processes a single block at a time:

- The first SHA instruction on the first message block uses an initial hash value as defined by the standard (each SHA algorithm has a specific initial hash value).
- Subsequent SHA instructions on successive message blocks use the produced hash value of the previous SHA operation.

The SHA instruction of the last message block produces the final hash value. The message digest is a subset of this final hash value.

The SHA instructions do not perform the following functionality:

- Preprocessing of a message.
- Initialization of the register buffer with the algorithm’s specific initial hash value.
- Copy the algorithm’s message digest to memory.

Software is required to preprocess the message. The FIFO_START instruction can be used to load (load FIFO) the register buffer with the initial hash value and to store (store FIFO) the message digest.

A SHA instruction uses “round weights” that are derived from the message block. Each SHA round uses a dedicated round weight. The “round weights” are derived on-the-fly (a new round weight is calculated when needed, and replaces a round weight from a previous round). The following table provides the number of rounds.

Instruction	Rounds
SHA1	80
SHA2_256	64
SHA2_512	80

The instructions use register buffer operands. Specifically, the instructions use reg_buff[2047:0]:

- reg_buff[1023:0] is used for the round weights. Before an instruction, this region is written with the message block. The SHA1 and SHA2_256 instructions use reg_buff[511:0] and the SHA2_512 instruction uses reg_buff[1023:0].
- reg_buff[1535:1024] is used for the hash value. Before the first SHA instruction, this region is written with the algorithm’s initial hash value. The algorithms with hash values smaller than 512 bits only use the lower bits of this region.
- reg_buff[2047:1536] is used as a working copy of the hash value. This working copy is updated during the SHA rounds and copied to reg_buff[1535:1024] at the end of the instruction.

The instructions are described in the following tables.

Table 12-14. SHA1 Instruction

Instruction Format	SHA1 ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
SHA1	0x69	Perform a SHA1 function on a 512-bit message block in reg_buff[511:0] with the current 160-bit hash value in reg_buff[1183:1024]. The resulting hash value is provided in reg_buff[1183:1024]. At the end of the instruction, reg_buff[1023:0] is set to ‘0’.

Table 12-15. SHA2_256 Instruction

Instruction Format	SHA2_256 ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
SHA2_256	0x6a	Perform a SHA2 function on a 512-bit message block in reg_buff[511:0] with the current 256-bit hash value in reg_buff[1279:1024]. The resulting hash value is provided in reg_buff[1279:1024]. At the end of the instruction, reg_buff[1023:0] is set to ‘0’.

Table 12-16. SHA2_512 Instruction

Instruction Format	SHA2_512 ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
SHA2_512	0x6b	Perform a SHA2 function on a 1024-bit message block in reg_buff[1023:0] with the current 512-bit hash value in reg_buff[1535:1024]. The resulting hash value is provided in reg_buff[1535:1024]. At the end of the instruction, reg_buff[1023:0] is set to ‘0’.

12.4.2 SHA3

The Secure Hash Algorithm-3 (SHA-3) is a family of six algorithms:

- SHA3-224
- SHA3-256
- SHA3-384
- SHA3-512
- SHAKE128
- SHAKE256

Each of these algorithms relies on a specific instance of the Keccak-p[b, nr] permutation, with b = 1600 and nr = 24. The parameter b specifies the permutation bit width (1600 bits) and the parameter nr specifies the number of permutation rounds (24 rounds).

The permutation bit width b is the sum of:

- The rate r, which is the number of consumed message bits or produced digest bits per application of the Keccak permutation (SHA3 instruction).
- The capacity c, which is defined as b-r.

All six hash algorithms are constructed by padding a message M and applying the Keccak-p[1600, 24] permutation repeatedly. The algorithms differ in terms of the rate r and the padding.

The permutation's rate r determines the speed of the algorithm: a higher rate requires less applications of the permutation function (SHA3 instruction).

The permutation's capacity c determines the security of the algorithm: a higher capacity provides higher security.

Table 12-17 lists the algorithms' rate and capacity. In addition, it lists the size of the message digest. **Note:** The SHA3 hash algorithms have fixed-length digests and the SHAKE extendable output functions have variable-length digests.

Table 12-17. Algorithm Rate and Capacity

Algorithm	Rate, Capacity	Digest Length
SHA3-224	r = 1152 b, c = 448 b	224 b (28 B)
SHA3-256	r = 1088 b, c = 512 b	256 b (32 B)
SHA3-384	r = 832 b, c = 768 b	384 b (48 B)
SHA3-512	r = 576 b, c = 1024 b	512 b (64 B)
SHAKE128	r = 1344 b, c = 256 b	Variable length
SHAKE256	r = 1088 b, c = 512 b	Variable length

The padded message has a length that is an integer multiple of the rate r. The message is processed by repeatedly applying the SHA3 permutation instruction. Each instruction application uses a message block of r bits. The permutation function combines the message block of r bits with the current permutation state of b bits and calculates a new permutation state of b bits.

- The first SHA3 instruction on the first message block uses the first message block as the initial permutation state.
- Subsequent SHA3 instructions on successive message blocks first combine the message block with an exclusive or (XOR) with the current state and calculate a new permutation state.

The message digest is produced in a similar way as the message is consumed: each application of the permutation instruction produces a message digest of r bits.

The SHA3 functionality includes a SHA3 instruction. The instruction performs a permutation on the 1600-bit state in reg_buf[1599:0].

The instruction implements the permutation, all other functionality is implemented in sofftware (combination of message block with the permutation state and copying the message digest to memory).

Table 12-18 describes the instruction.

Table 12-18. SHA3 Instruction

Instruction Format	SHA3 ( )	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
SHA3	0x6c	Perform a SHA3 permutation on a 1600-bit state reg_buff[1599:0]. The resulting state is provided in reg_buff[1599:0].

12.5 DES and TDES

The (T)DES functionality includes block ciphers and inverse block ciphers according to the DES and triple DES standard (FIPS 46-3). Note that this standard was withdrawn, because it no longer provides the security that is needed to protect federal government information.

(T)DES is supported for backward compatibility.

- The DES block cipher encrypts a 64-bit block of plaintext data into a 64-bit block of ciphertext data.
- The DES inverse block cipher decrypts a 64-bit block of ciphertext data into a 64-bit block of plaintext data.

The DES symmetric key consists of 64 bit. Only 56 bits are used by the algorithm, the other 8 bits are used for parity checking.

Tripple-DES applies the DES block cipher three times. Each application uses a different key, which results in a 192-bit TDES key bundle consisting of three 64-bit keys SK1, SK2 and SK3.

- The TDES block cipher uses the DES block cipher with key SK1 to encrypt a 64-bit plaintext into a 64-bit block T1. Next, it uses the DES inverse block cipher with key SK2 to decrypt the 64-bit block T1 into a 64-bit block T2. Next, it uses the DES block cipher with key SK3 to encrypt the 64-bit block T2 into a 64-bit ciphertext.
- The TDES inverse block cipher uses the DES inverse block cipher with key SK3 to decrypt a 64-bit ciphertext into a 64-bit block T2. Next, it uses the DES block cipher with key SK2 to encrypt the 64-bit block T2 into a 64-bit block T1. Next, it uses the DES inverse block cipher with key SK1 to decrypt the 64-bit block T1 into a 64-bit plaintext.

There are four instructions: DES, DES_INV, TDES and TDES_INV, which use register buffer operands. Specifically, the instructions use the 128-bit subpartitions block0, block1, block4, block5, block6, and block7.

- Subpartitions block0 and block1 are used for plaintext and ciphertext data.
- Subpartitions block4, block5, block6, and block7 are used for key information (block5 and block7 are only used by the TDES and TDES_INV instructions).

Unlike the BLOCK_MOV instruction, the (T)DES instructions use predetermined/fixed subpartitions.

The instructions are described in the following tables.

Table 12-19. DES Instruction

Instruction Format	DES ( )	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
DES	0x52	Perform a DES block cipher. ■ The 64-bit plaintext is in block0[63:0]. ■ The resulting 64-bit ciphertext is in block1[63:0]. ■ The 64-bit input key is in block4[63:0]. ■ The 64-bit output key (round key of the final cipher round) is in block6[63:0]. The instruction is non-destructive: block0 and block4 are not changed by the instruction.

Table 12-20. DES_INV Instruction

Instruction Format	DES_INV ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
DES_INV	0x53	Perform a DES inverse block cipher. ■ The 64-bit ciphertext is in block0[63:0]. ■ The resulting 64-bit plaintext is in block1[63:0]. ■ The 64-bit input key is in block4[63:0]. ■ The 64-bit output key (round key of the final inverse cipher round) is in block6[63:0]. The instruction is non-destructive: block0 and block4 are not changed by the instruction.

Table 12-21. TDES Instruction

Instruction Format	TDES ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
TDES	0x54	Perform a TDES block cipher. ■ The 64-bit plaintext is in block0[63:0]. ■ The resulting 64-bit ciphertext is in block1[63:0]. ■ The 192-bit input key is in {block5[63:0], block4[127:0]} (= {SK3[63:0], SK2[63:0], SK1[63:0]}). ■ The 192-bit output key (round key of the final cipher round) is in {block7[63:0], block6[127:0]}. The instruction is non-destructive: block0, block4 and block5 are not changed by the instruction.

Table 12-22. TDES_INV Instruction

Instruction Format	TDES_INV ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
TDES_INV	0x55	Perform a TDES inverse block cipher. ■ The 64-bit ciphertext is in block0[63:0]. ■ The resulting 64-bit plaintext is in block1[63:0]. ■ The 192-bit input key is in {block5[63:0], block4[127:0]} (= {SK3[63:0], SK2[63:0], SK1[63:0]}). ■ The 192-bit output key (round key of the final cipher round) is in {block7[63:0], block6[127:0]}. The instruction is non-destructive: block0, block4 and block5 are not changed by the instruction.

The (T)DES instructions are used with the FF_START, FF_CONTINUE, FF_STOP, BLOCK_MOV, BLOCK_XOR, and BLOCK_SET instructions to implement different block cipher modes, such as EBC, CBC, OFB, CTR, and CMAC.

12.6 AES

The AES functionality includes a block cipher and an inverse block cipher per the AES standard (FIPS 197):

- The block cipher (AES instruction) encrypts a 128-bit block of plaintext data into a 128-bit block of ciphertext data.
- The inverse block cipher (AES_INV instruction) decrypts a 128-bit block of ciphertext data into a 128-bit block of plaintext data.

AES is a symmetric block cipher: it uses the same symmetric key for the block cipher and inverse block cipher. The (inverse) block cipher consists of multiple rounds. Each round uses a round key that is generated from the symmetric key.

The block cipher uses the symmetric key as the (start) round key for the first cipher round. The round key for second cipher round is generated from the symmetric key. The round key for the third cipher round is generated from the round key of the second cipher round, and so forth. The round key for the last cipher round is generated from the round key of the one-before-last cipher round.

The inverse block cipher uses the round key of the final block cipher round as the (start) round key for the first inverse cipher round. The round key for the second inverse cipher round is generated from the round key of the first inverse cipher round, and so forth. The round key for the last inverse cipher round is generated from the round key of the one-before-last inverse cipher round. The round key of the last inverse cipher round is the same as the round key of the first cipher round (the symmetric key).

Round key generation is independent of the plaintext data or ciphertext data. The AES instruction requires the symmetric key as input to the block cipher. The AES_INV instruction requires the round key of the final cipher round as input to the inverse block cipher.

The component supports 128 bit, 192 bit, and 256 bit keys. The key size is specified by AES_CTL.KEY_SIZE[1:0]. The key size determines the number of rounds. Table 12-23 gives the number of rounds.

Table 12-23. AES Cipher Rounds

AES Key Size	Rounds
AES128	10 rounds
AES192	12 rounds
AES256	14 rounds

AES and AES_INV are the two different AES instructions; they use register buffer operands. Specifically, the instructions use the 128-bit subpartitions block0, block1, block4, block5, block6, and block7.

- Subpartitions block0 and block1 are used for plaintext and ciphertext data.
- Subpartitions block4, block5, block6 and block7 are used for key information.

Unlike, for example, the BLOCK_MOV instruction, the AES and AES_INV instructions use predetermined/fixed subpartitions.

The instructions are described in the following tables.

Table 12-24. AES Instruction

Instruction Format	AES ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
AES	0x50	Perform an AES block cipher. ■ The 128-bit plaintext is in block0. ■ The resulting 128-bit ciphertext is in block1. ■ The input key is in block4 and block5. For a 128-bit key, the key is in block4[127:0]. For a 192-bit key, the key is in {block5[63:0], block4[127:0]}. For a 256-bit key, the key is in {block5[127:0], block4[127:0]}. ■ The output key (round key of the final cipher round) is in block6 and block7. For a 128-bit key, the key is in block6[127:0]. For a 192-bit key, the key is in {block7[63:0], block6[127:0]}. For a 256-bit key, the key is in {block7[127:0], block6[127:0]}. Note that the output key is the input key for an AES inverse block cipher. The instruction is non-destructive: block0, block4, and block5 are not changed by the instruction. Note that the AES instruction can be used to derive the AES_INV input key from the symmetric key.

Table 12-25. AES_INV Instruction

Instruction Format	AES_INV ()	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation Code**	**Functionality**
AES_INV	0x51	Perform an AES block cipher. ■ The 128-bit ciphertext is in block0. ■ The resulting 128-bit plaintext is in block1. ■ The input key is in block4 and block5. For a 128-bit key, the key is in block4[127:0]. For a 192-bit key, the key is in {block5[63:0], block4[127:0]}. For a 256-bit key, the key is in {block5[127:0], block4[127:0]}. ■ The output key (round key of the final inverse cipher round) is in block6 and block7. For a 128-bit key, the key is in block6[127:0]. For a 192-bit key, the key is in {block7[63:0], block6[127:0]}. For a 256-bit key, the key is in {block7[127:0], block6[127:0]}. Note that the output key is the input key for an AES block cipher. The instruction is non-destructive: block0, block4 and block5 are not changed by the instruction.

The AES and AES_INV instructions are used with the FF_START, FF_CONTINUE, FF_STOP, BLOCK_MOV, BLOCK_XOR, and BLOCK_SET instructions to implement different block cipher modes, such as EBC, CBC, OFB, CTR, and CMAC.

12.7 CRC

The CRC functionality performs a cyclic redundancy check with a programmable polynomial of up to 32 bits.

The load FIFO 0 provides the data (and the size of the data) on which the CRC is performed. The data must be laid out in little endian format (least significant byte of a multi-byte word should be located at the lowest memory address of the word).

CRC_DATA_CTL.DATA_XOR[7:0] specifies a byte pattern with which each data byte is XOR'd. This allows for inversion of the data byte value.

CRC_CTL.DATA_REVERSE allows for bit reversal of the data byte (this provides support for serial interfaces that transfer bytes in most-significant-bit first and least-significant bit first configurations).

CRC_POL_CTL.POLYNOMIAL[31:0] specifies the polynomial. The polynomial specification omits the high order bit and should be left aligned. For example, popular 32-bit and 16-bit CRC polynomials are specified as follows:

CRC32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

CRC_POL_CTL.POLYNOMIAL[31:0] = 0x04c11db7

CRC16-CCITT: $x^{16} + x^{12} + x^5 + 1$

CRC_POL_CTL.POLYNOMIAL[31:0] = (0x1021 << 16)

CRC16: $x^{16} + x^{15} + x^2 + 1$

CRC_POL_CTL.POLYNOMIAL[31:0] = (0x8005 << 16)

RESULT.DATA[31:0] holds the LFSR state of the CRC calculation. Before the CRC operation, this field should be initialized with the CRC seed value.

CRC_REM_CTL.REM_XOR[31:0] specifies a 32-bit pattern with which the RESULT.DATA[31:0] LFSR state is XOR'd.

CRC_CTL.REM_REVERSE allows for bit reversal of the XOR'd state.

CRC_REM_RESULT.REM[31:0] holds the result of the CRC calculation, and is derived from the end state of the CRC calculation (RESULT.DATA[31:0]).

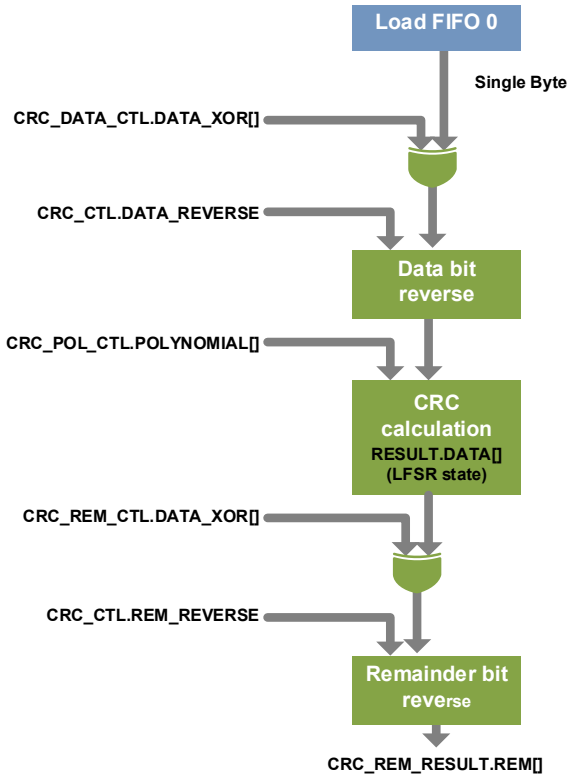
The CRC instruction is described in Table 12-26.

Table 12-26. CRC Instruction

Instruction Format	CRC ( )	
Encoding	IW[31:24] = "operation code"	
Mnemonic	**Operation Code**	**Functionality**
CRC	0x58	Perform the CRC function on all data in load FIFO 0. The result of the CRC function is provided through CRC_REM_RESULT.REM[31:0]. If the data is scattered in memory, a single FF_START and multiple FF_CONTINUE instructions should be used to gather the data in a load FIFO 0 Byte stream. Each FF_START and FF_CONTINUE instruction should be followed by a CRC instruction, which processes all the bytes of the associated FIFO instruction (the CRC instruction does not have a "size" operand).

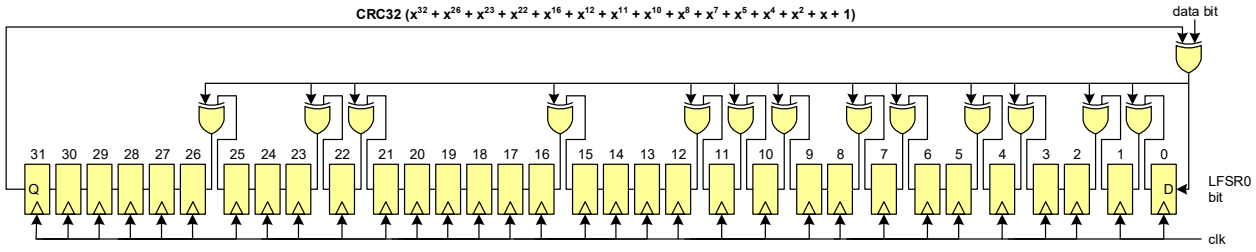
Figure 12-2 illustrates the CRC functionality.

Figure 12-2. CRC Functionality



The Linear Feedback Shift Register functionality operates on the LFSR state. It uses the programmed polynomial and consumes a data bit for each iteration (eight iterations are performed per cycle to provide a throughput of one data byte per cycle). Figure 12-3 illustrates the functionality for the CRC32 polynomial ($x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$).

Figure 12-3. CRC32 Functionality



Different CRC algorithms require different seed values and have different requirements for XOR functionality and bit reversal. Table 12-27 provides the proper settings for the CRC32, CRC16-CCITT, and CRC16 algorithms. The table also provides the remainder after the algorithms are performed on a five-byte array {0x12, 0x34, 0x56, 0x78, 0x9a}.

Table 12-27. CRC32, CRC16-CCITT, and CRC16 Algorithm Settings

MMIO Register Field	CRC32	CRC16-CCITT	CRC16
CRC_POL_CTL.POLYNOMIAL	0x04c11db7	0x10210000	0x80050000
CRC_CTL.DATA_REVERSE	1	0	1
CRC_DATA_CTL.DATA_XOR	0x00	0x00	0x00
RESULT.DATA (seed)	0xffffffff	0xffff0000	0xffff0000
CRC_CTL.REM_REVERSE	1	0	1
CRC_REM_CTL.REM_XOR	0xffffffff	0x00000000	0x00000000
CRC_REM_RESULT.REM	0x3c4687af	0xf8a00000	0x000048d0

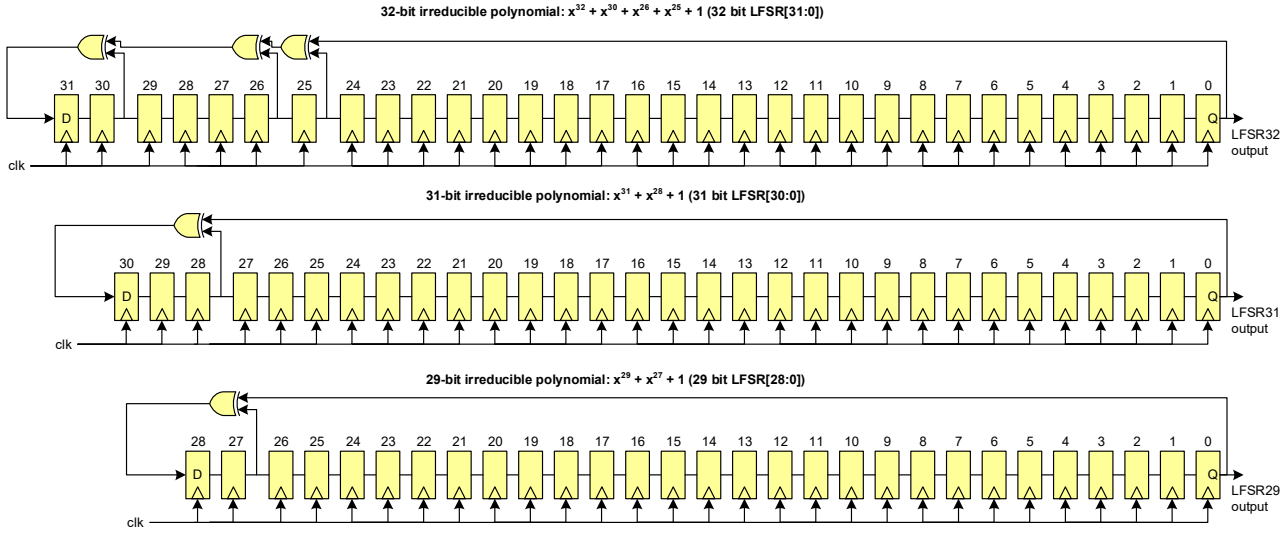
12.8 PRNG

The pseudo random number generation (PRNG) component generates pseudo random numbers in a fixed range [0, PR_MAX_CTL.DATA[31:0]]. The generator is based on three Fibonacci-based Linear Feedback Shift Registers (LFSRs). The following three irreducible polynomials (with minimum feedback) are used:

- 32-bit polynomial: $x^{32} + x^{30} + x^{26} + x^{25} + 1$
- 31-bit polynomial: $x^{31} + x^{28} + 1$
- 29-bit polynomial: $x^{29} + x^{27} + 1$

Figure 12-4 illustrates the LFSR functionality.

Figure 12-4. Fixed Fibonacci-based LFSRs

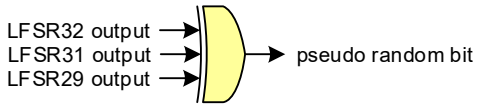


Software initializes the LFSRs with non-zero seed values. The PR_LFSR_CTL0, PR_LFSR_CTL1 and PR_LFSR_CTL2 registers are provided for this purpose. At any time, the state of these registers can be read to retrieve the state of the LFSRs. The 32-bit LFSR generates a repeating bit sequence of 232 – 1 bits, the 31-bit LFSR generates a repeating bit sequence of 231 – 1 and the 29-bit LFSR generates a repeating bit sequence of 229 – 1. As the

numbers 232-1, 231-1, and 229-1 are relatively prime, the XOR output is a repeating bit sequence of roughly 232+31+29.

The final pseudo random bit is the XOR of the three bits that are generated by the individual LFSRs.

Figure 12-5. XOR Reduction Logic



The pseudo random number generator uses a total of 33 pseudo random bits to generate a result in the range [0, PR_MAX_CTL.DATA[31:0]].

To generate a pseudo random number result, the following calculation is performed.

MAX[31:0] = PR_MAX_CTL.DATA[31:0]

MAX_PLUS1[32:0] = MAX[31:0] + 1;

product[63:0] = MAX_PLUS1[32:0] * pr[32:1] + MAX[31:0] * pr[0];

result = product[63:32];

PR_CMD.START is the PR command. The maximum value of the generated random number (in PR_RESULT.DATA) is specified by PR_MAX_CTL.DATA. The PR command can be executed in parallel with the instruction FIFO instructions and the TR command.

12.9 TRNG

The true random number generator component (TRNG) generates true random numbers. The bit size of these generated numbers is programmable (TR_CTL.SIZE is in the range [0,32]).

The TRNG relies on up to six ring oscillators to provide physical noise sources. A ring oscillator consists of a series of inverters connected in a feedback loop to form a ring. Due to (temperature) sensitivity of the inverter delays, jitter is introduced on a ring's oscillating signal. The jittered oscillating signal is sampled to produce a “digitized analog signal” (DAS). This is done for all multiple ring oscillators.

To increase entropy and to reduce bias in DAS bits, the DAS bits are further postprocessed. Post-processing involves two steps:

- An optional reduction step (over up to six ring oscillator DAS bits and over one or multiple DAS bit periods) to increase entropy.
- An optional “von Neumann correction” step to reduce a ‘0’ or ‘1’ bias.

  This correction step processes pairs of reduction bits as produced by the previous step. Given two reduced bits r0 and r1 (with r0 being produced before r1), the correction step is defined as follows:

  - {r0, r1} = {0, 0}: no bit is produced
  - {r0, r1} = {0, 1}: a ‘0’ bit is produced (bit r0)
  - {r0, r1} = {1, 0}: a ‘1’ bit is produced (bit r0)
  - {r0, r1} = {1, 1}: no bit is produced

In other words the correction step only produces a bit on a ‘0’ to ‘1’ or ‘1’ to ‘0’ transition. Note that for a random input bit sequence, the correction step produces an output bit sequence of roughly one-quarter the frequency of the input bit sequence (the input reduction bits are processed in non-overlapping pairs and only half of the pair encodings result in an output bit).

Post-processing produces bit samples that are considered true random bit samples. The true random bit samples are shifted into a register, to provide random values of up to 32 bits.

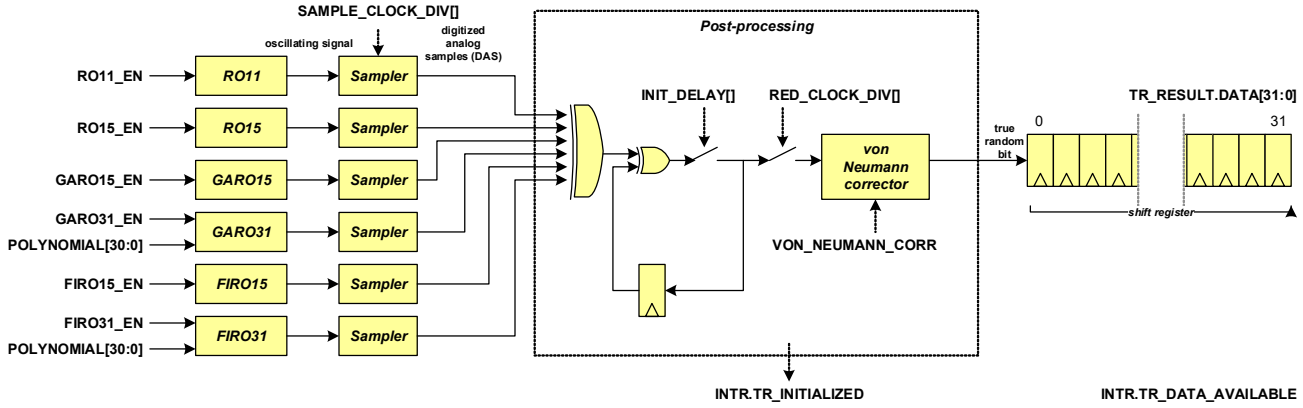
As a result of high-switching activity, ring oscillators consume a significant amount of power. Therefore, when the TRNG functionality is disabled, the ring is “broken” to prevent switching. When the TRNG functionality is enabled, a ring oscillator initially has predictable behavior. However, over time, infinitesimal environmental (temperature) changes cause an increasing deviation from this predictable behavior.

- During the initial delay, the ring oscillator is not a reliable physical noise source.
- After an initial delay, the same ring oscillator will show different oscillation behavior and provides a reliable physical noise source.

Therefore, the DAS bits can be dropped during an initialization period (TR_CTL.INIT_DELAY[]).

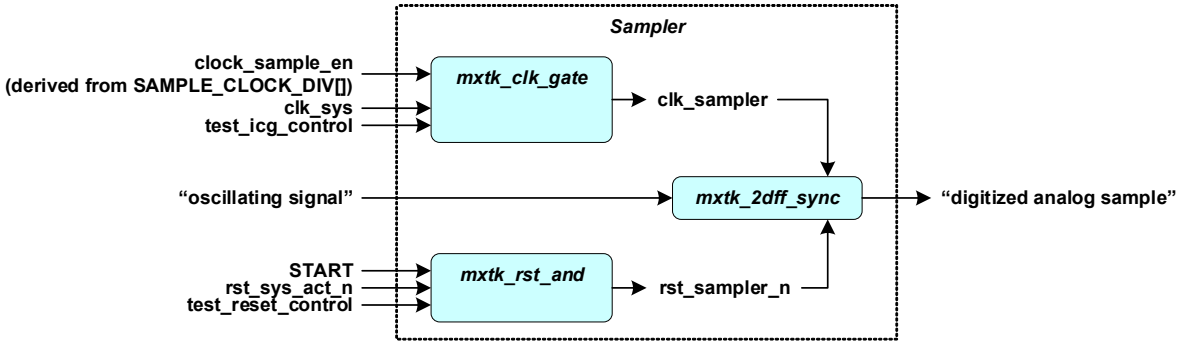
Figure 12-6 gives an overview of the TRNG component.

Figure 12-6. TRNG Overview



The “Sampler” logic digitizes an oscillating signal. Figure 12-7 illustrates the functionality (the complete logic is implemented using standard platform toolkit components).

Figure 12-7. Sampler Logic

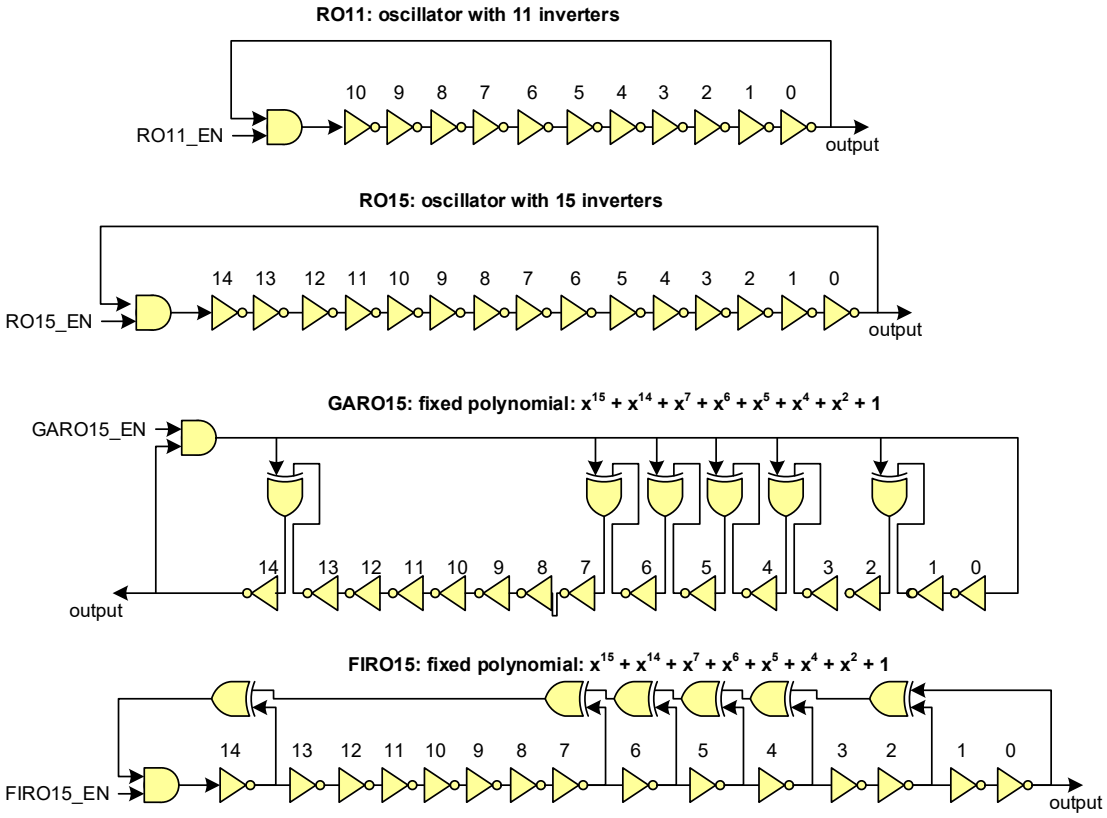


Note that when a ring oscillator is disabled, the synchronization logic is reset. Therefore, the ring oscillator contributes a constant ‘0’ to the reduction step of the post-processing. As mentioned, the TRNG relies on up to six ring oscillators:

- RO11: A fixed ring oscillator consisting of 11 inverters.
- RO15: A fixed ring oscillator consisting of 15 inverters.
- GARO15: A fixed Galois based ring oscillator of 15 inverters.
- GARO31: A flexible Galois based ring oscillator of up to 31 inverters. A programmable polynomial of up to order 31 provides the flexibility in the oscillator feedback.
- FIRO15: A fixed Fibonacci based ring oscillator of 15 inverters.
- FIRO31: A flexible Fibonacci based ring oscillator of up to 31 inverters. A programmable polynomial of up to order 31 provides the flexibility in the oscillator feedback.

Each ring oscillator can be enabled or disabled. When disabled, the ring is “broken” to prevent switching. The following Figures illustrate the schematics of the fixed ring oscillators.

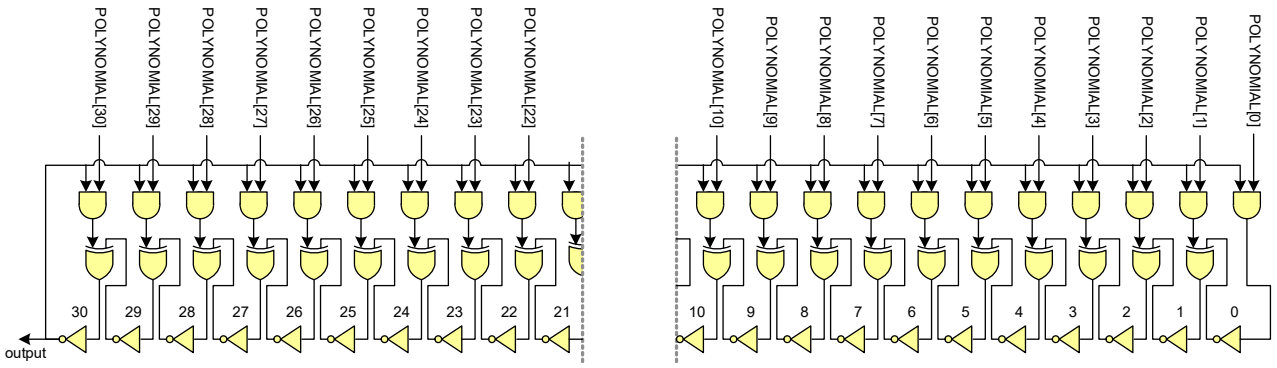
Figure 12-8. Four Fixed Ring Oscillators: RO11, RO15, GARO15, FIRO15



The XXX_EN enable signals originate from a MMIO register field.

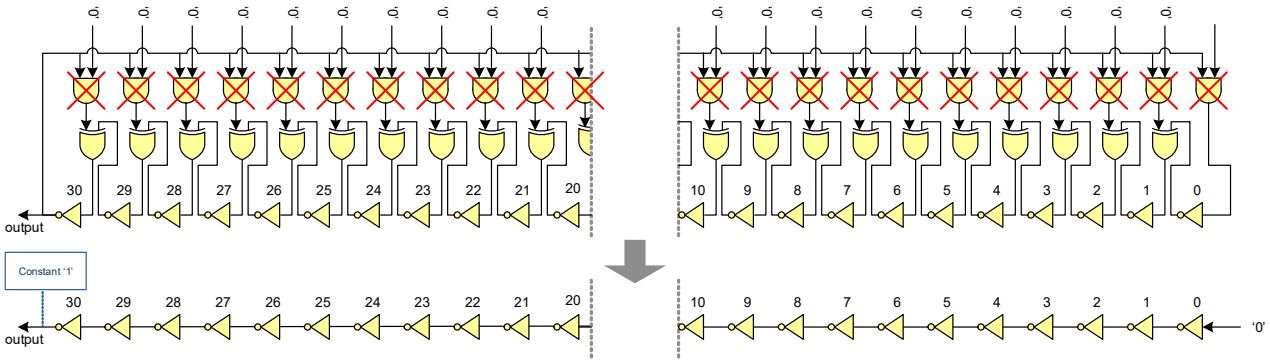
The flexible Galois- and Fibonacci-based ring oscillators rely on programmable polynomials to specify the oscillator feedback. This allows for rings of 1, 3, 5, …, 31 inverters (an odd number is required to generate an oscillating signal). Figure 12-9 gives an overview of the Galois-based ring oscillator.

Figure 12-9. Flexible Galois-based Ring Oscillators: GARO31



When the ring oscillator is disabled (GARO31_EN is ‘0’), the polynomial is forced to “0” and the ring is broken as illustrated by Figure 12-10.

Figure 12-10. GARO31 “stopped”



The programmable polynomial specifies the oscillator feedback. Figure 12-11 illustrates two examples.

Figure 12-11. GARO31 – Two Examples

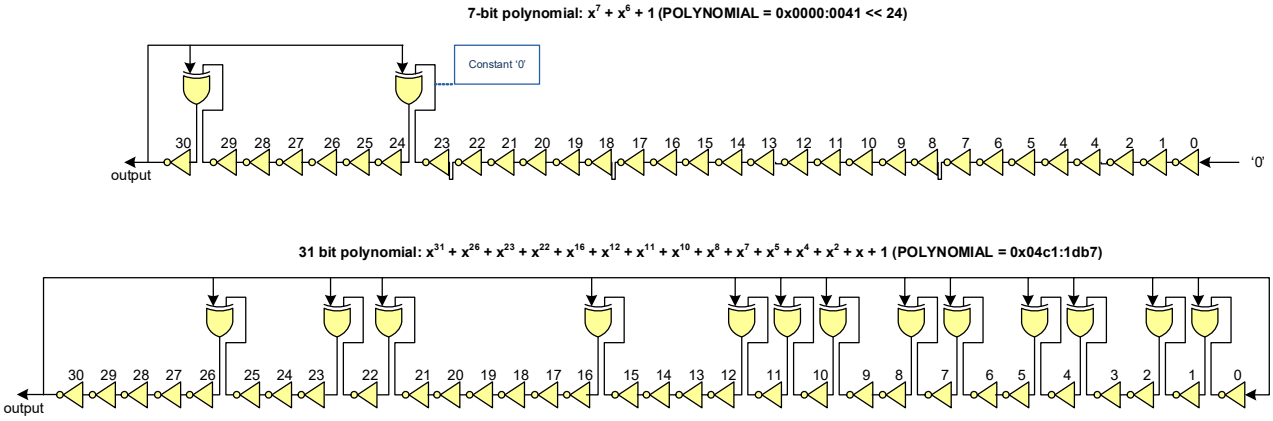
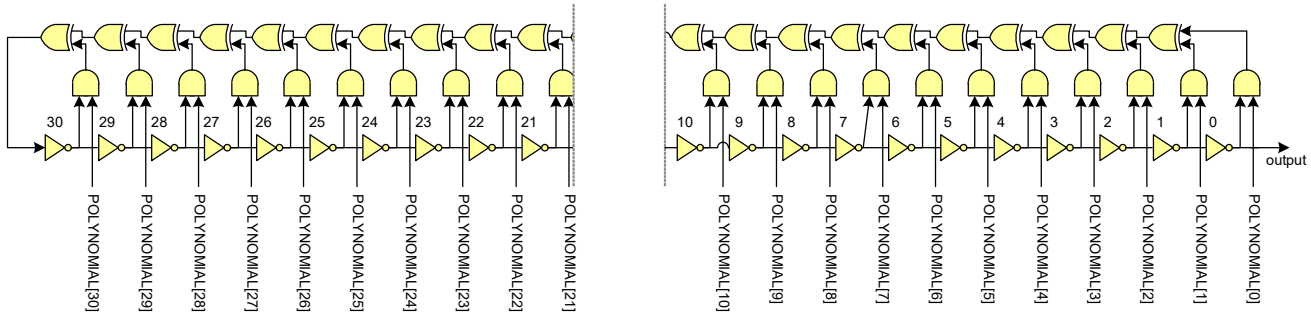


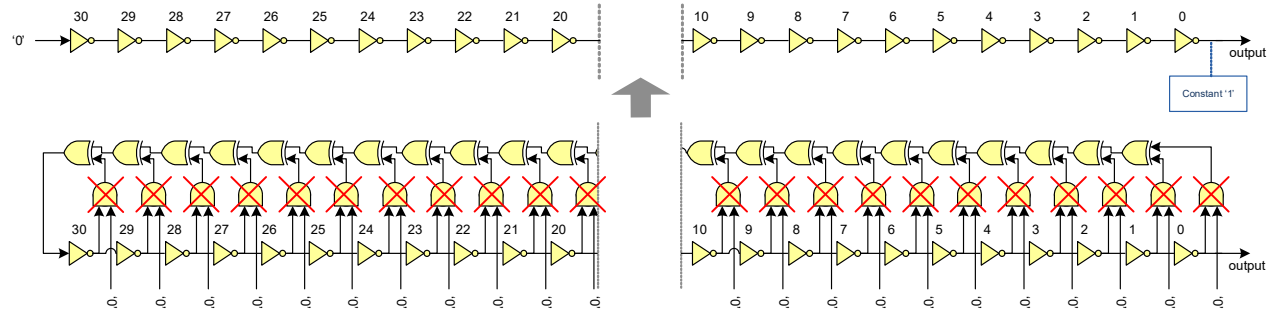
Figure 12-12 gives an overview of the Fibonacci-based ring oscillator.

Figure 12-12. Flexible Fibonacci-based Ring Oscillators: FIRO31



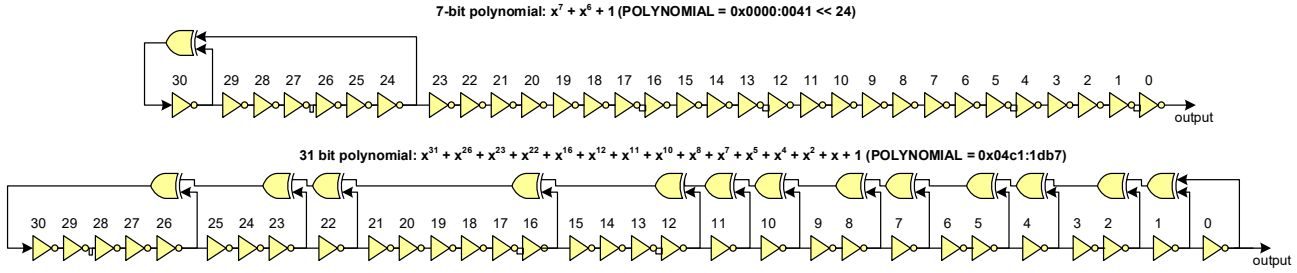
When the ring oscillator is disabled (FIRO31_EN is ‘0’), the polynomial is forced to “0” and the ring is broken as illustrated by Figure 12-13.

Figure 12-13. FIRO31 “stopped”



The programmable polynomial specifies the oscillator feedback. Figure 12-14 illustrates two examples.

Figure 12-14. FIRO31 – Two Examples



There is one TR command, TR_CMD.START, which can be executed in parallel with the instruction FIFO instructions and the PR command. The size of the generated random number (in TR_RESULT.DATA) is specified by TR_CTL.SIZE.

The TRNG has a built-in health monitor that performs tests on the digitized noise source to detect deviations from the intended behavior. For example, the health monitor detects “stuck at” faults in the digitized analog samples. The health monitor tests one out of three selected digitized bit streams:

- DAS bitstream. This is XOR of the digitized analog samples.
- RED bitstream. This is the bitstream of reduction bits. Note that each reduction bit may be calculated over multiple DAS bits.
- TR bitstream. This is the bitstream of true random bits (after the “von Neumann reduction” step).

The health monitor performs two different tests:

The repetition count test. This test checks for the repetition of the same bit value (‘0’ or ‘1’) in a bitstream. A detection indicates that a specific active bit value (specified by a status field BIT) has repeated for a pre-programmed number of bits (specified by a control field CUTOFF_COUNT[7:0]). The test uses a counter to maintain the number of repetitions of the active bit value (specified by a status field REP_COUNT[7:0]).

If the test is started (specified by START_RC field) and a change in the bitstream value is observed, the active bit value BIT is set to the new bit value and the repetition counter REP_COUNT[] is set to ‘1’. If the bitstream value is unchanged, the repetition counter REP_COUNT[] is incremented by “1”.

A detection stops the repetition count test (the START_RC field is set to ‘0’), sets the associated interrupt status field to ‘1’ and ensures that hardware does not modify the status fields. When the test is stopped, REP_COUNT[] equals CUTOFF_COUNT[].

A detection stops the TRNG functionality (all TR_CTL.XXX_EN fields are set to ‘0’) if TR_CTL.STOP_ON_RC_DETECT is set to ‘1’.

The adaptive proportion test. This test checks for a disproportionate occurrence of a specific bit value (‘0’ or ‘1’) in a bit stream. A detection indicates that a specific active bit value (specified by a status field BIT) has occurred a pre-programmed number of times (specified by a control field CUTOFF_COUNT[15:0]) in a bit sequence of a specific bit window size (specified by a control field WINDOW_SIZE[15:0]). The test uses a counter to maintain an index in the current window (specified by

WINDOW_INDEX[15:0]) and a counter to maintain the number of occurrences of the active bit value (specified by a status field OCC_COUNT[15:0]).

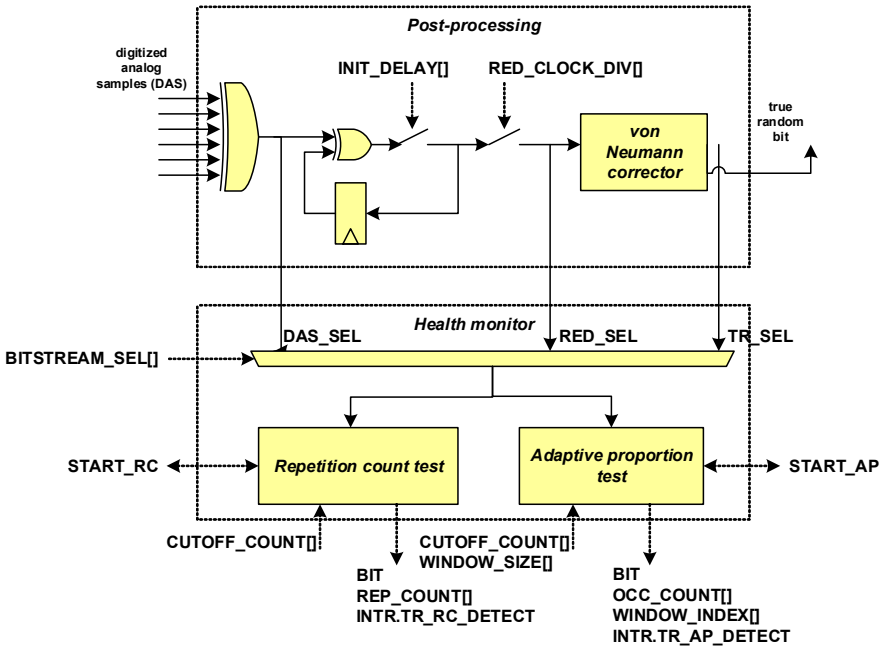
If the test is started (specified by START_AP field), the bitstream is partitioned in bit sequences of a specific window size. At the first bit of a bit sequence, the active bit value BIT is set to the first bit value, the counter WINDOW_INDEX is set to “0” and the counter OCC_COUNT is set to “1”. For all other bits of a bit sequence, the counter WINDOW_INDEX is incremented by “1”. If the new bit value equals the active bit value BIT, the counter OCC_COUNT[15:0] is incremented by “1”. Note that the active bit value BIT is only set at the first bit of a bit sequence.

A detection stops adaptive proportion test (the START_AP field is set to ‘0’), sets the associated interrupt status field to ‘1’ and ensures that hardware does not modify the status fields. When the test is stopped, OCC_COUNT[] equals CUTOFF_COUNT[] and the WINDOW_INDEX identifies the bit sequence index on which the detection occurred.

A detection stops the TRNG functionality (all TR_CTL.XXX_EN fields are set to ‘0’) if TR_CTL.STOP_ON_AP_DETECT is set to ‘1’.

Figure 12-15 illustrates the health monitor functionality.

Figure 12-15. Health Monitor Functionality



Implementation note. The ring oscillators have special design requirements. They are not synthesized logic, but manually constructed from selected cells from the standard cell library. Three types of standard cells are required:

- An inverter cell. The selected cell should have similar rise and fall time requirements. Clock tree inverter cells tend to have this property and are the preferred implementation of the inverter cell.
- A two-input XOR cell.
- A two-input AND cell.

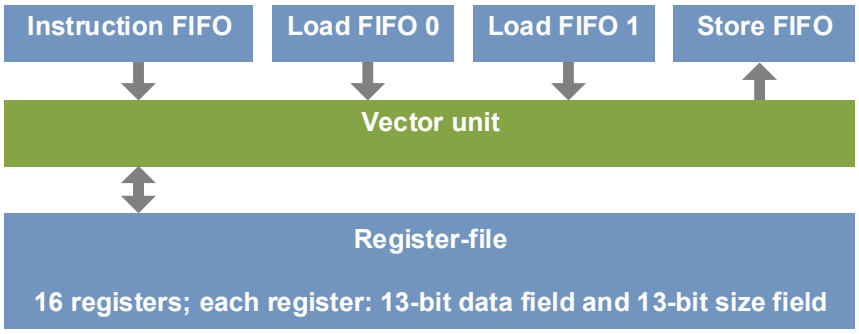
The design uses three Verilog design modules to instantiate the selected cells. This allows easy porting from one standard cell library to another standard cell library. The three Verilog design modules should not be changed by tools (similar to how the platform toolkit components are treated in the design flow).

A ring oscillator should be placed and routed in a self-contained rectangular area. This area should preferably be as small as possible.

12.10 Vector Unit

The vector unit (VU) addresses the requirements of asymmetric key cryptography. Similar to other cryptography functionality, the VU connects to the memory interface through the load and store FIFOs. The memory buffer or system memory contains memory operand data for the vector unit instructions.

Figure 12-16. Vector Unit



The VU addresses asymmetric key cryptography. Asymmetric key cryptography includes RSA, Diffie-Hellman key exchange, digital signature authentication, and elliptic curve cryptography (ECC). These algorithms share the requirement to efficiently perform computations in a Galois field on large integers of up to 1000's of bits.

The VU performs instructions on operand data.

- The VU instructions are provided by the instruction FIFO.
- The VU operand data is specified by any of the following (all encoded as part of the 32-bit instruction word):
  - instruction operation code
  - instruction register indices
  - instruction immediate values

The VU uses the VU register file. A VU register is a (data, size) pair: a 13-bit data field (bits 28 down to 16) and a 13-bit size field (bits 12 down to 0). The data field is either used as:

- Instruction operand data: the complete 13-bit data is used.
- An offset into memory: the 13-bit data value is a word offset wrt. a base address into memory.

The size field is only used when the data field is used as an offset into memory. In this case, the size field specifies the bit size (minus 1) of the memory operand (the 13-bit field specifies a size in the range of [1, 8192] bits).

The VU is a "trimmed down" application domain specific CPU:

- "Trimmed down". It does not have a CPU instruction fetch unit that supports non-sequential program flow. This functionality is typically not required for asymmetric key cryptography functions or needs to be provided by the external AHB-Lite bus master (typically a CPU). This design decision simplifies the VU.

  Note that the instruction FIFO provides the mechanism by which instructions are provided. This mechanism only supports a sequential VU program flow.
- Application domain specific. The VU instructions and operand data are tuned for asymmetric key cryptography. For example, instructions are provided for arithmetic over binary extension fields (GF(2m)) or prime fields (GF(p)), and memory operands allow for a large operand length of up to 8192 bits (although the VU internal data path is limited to 32 bit).
- CPU. The VU has CPU-like design components, such as an instruction decoder, a register file, and data path with multiple functional units.

The VU architecture is the result of a tradeoff between silicon area, design complexity, performance efficiency, and algorithmic flexibility. Note that most asymmetric key algorithms can be performed by the regular CPU. However, this typically comes at a lower performance level, possibly large code footprint (due to loop unrolling of code to improve performance when operating on large operand data), and possibly large data footprint (due to lookup tables to improve performance).

The improved performance of the VU (over a regular CPU) for asymmetric key algorithms also allows for more generic algorithmic implementations, as opposed to implementations targeting, such as specific primes or irreducible polynomials.

Before describing the architecture, a short C example is illustrates the VU functionality.

```
#define SIZE 4096
void Example ()
{
int a = 0; int b = 1; int c = 2; // assign register indices
uint8_t c_data[SIZE/8];
ALLOC_MEM (a, SIZE); // allocate 4096 bits of data
ALLOC_MEM (b, SIZE); // allocate 4096 bits of data
ALLOC_MEM (c, SIZE); // allocate 4096 bits of data
Crypto_WriteMemNumber (a, “0x21542555:fed35532: … :ffea2345”);
Crypto_writeMemNumber (b, “0xef45ac2a:34a312bc: … :000003ab”);
ADD (c, a, b);
Crypto_ReadMemNumber (c, c_data);
}
```

The example adds two 4096-bit numbers and produces a 4096-bit number. The example is explained as follows:

- Three local variables are assigned VU register indices:
  - local variable a uses register 0
  - local variable b uses register 1
  - local variable c used register 2
- Memory is allocated in the memory buffer for each variable: each variable needs 4096 bits.
- A function “Crypto_WriteMemNumber” is called to initialize the memory operand data for source variables a and b.
- A VU ADD instruction (long integer addition) is executed.
- A function “Crypto_WriteMemNumber” is called to read the result of the instruction from the memory.

The example illustrates that the full expressive power of the C language is available when writing software for the VU.

The example illustrates that relatively complex functionality can be expressed in only a few lines of C code (consider implementing the same functionality on a 32-bit Arm processor).

The “Crypto_” functions copy data to and from the memory. These functions are executed on the regular CPU and may take a significant number of cycles. Asymmetric key algorithm typically requires little copy functions and a lot of VU instructions. As a result, the cycle overhead of the copy functions is negligible. As an example, RSA requires exponentiation of a number by a second number modulo a third number. This RSA functionality requires only four copy functions (three to initialize three memory operands and one to read the result), but requires thousands of VU instructions.

The following sections provide an architectural overview of the VU.

12.10.1 VU Register File

The register file has sixteen registers (r0 through r15). Each register consists of a 13-bit data field and a 13-bit size field. The data field is either used as:

- Instruction operand data: the 13-bit data is used.
- An offset into memory: the 13-bit data value is a word offset for a base address into memory. Typically, memory operand data is located in the crypto memory buffer (this provides better latency/performance than memory operand data in system memory).

The size field is used only when the data field is used as an offset into memory. In this case, the size field specifies the bit size (minus 1) of the memory operand (the 13-bit field specifies a size in the range of [1, 8192] bits).

The CPU software decides what instructions use what registers. There is no compiler with specific conventions for register usage or to perform register allocation. To ensure some consistency and interoperability of the functions in a software library, the software programmer should introduce its own conventions. Some examples of conventions are:

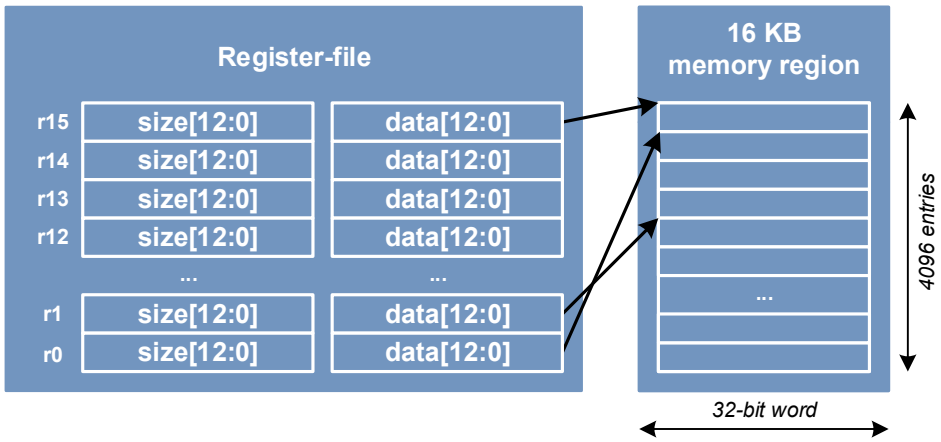
- Each non-leaf function saves all registers to the stack on function entry and restores all registers from the stack on function exit. This makes all registers available for use within the function.

- Each non-leaf function restricts its register use to registers r4 through r14.
- Each leaf function either restricts its register use to registers r0 through r3 or follows the same save/restore convention as non-leaf functions.

The register-file register r15 serves a specific purpose. Similar to the Arm architecture, register r15 is used as a stack pointer.

Figure 12-17 illustrates the register file and how the data fields are used as offsets in a 16 KB memory region (either in the crypto memory buffer or the system memory). The base address of the memory region is provided by VU_CTL.ADDR[31:8] and the memory region size is specified by VU_CTL.MASK[].

Figure 12-17. Register File



12.10.2 Stack

The VU stack resides in the memory region. Register r15 is used as a stack pointer. The stack has two purposes:

- It is used to save/restore registers r0 through r14. Each register (data field and size field) uses a single 32-bit word.

  The instruction PUSH_REG saves/pushes all registers r0 through r14 on the stack (and register r15 is decremented by 15).

  The instruction POP_REG restores/pops all registers r0 through r14 from the stack (and register r15 is incremented by 15).

  The use of the stack pointer (register r15) is implied by the PUSH_REG and POP_REG instructions.
- It is used to allocate memory operands.

  The instruction ALLOC_MEM (rx, size-1) allocates enough 32-bit words to hold a memory operand of “size” bits for register rx. The stack pointer (register r15) is decremented by the number of allocated 32-bit words. The data field of register rx is updated with the new stack pointer value.

  The instruction FREE_MEM (“15 bit pattern”) frees the 32-bit words that hold memory operand data associated with the registers identified by the bit pattern. Note that a register’s size field specifies the number 32-bit words that hold its associated memory operand. The stack pointer is incremented by the number of freed 32-bit words.

  The use of the stack pointer (register r15) is implied by the instruction. Note that to allocate multiple memory operands, multiple ALLOC_MEM instructions are required. However, to free multiple memory buffer operands, only a single FREE_MEM instruction is required.

Figure 12-18 illustrates the VU register-file state before and after the execution of instructions ALLOC_MEM (r0, 32-1) (32 bits requiring one 32-bit stack element) and ALLOC_MEM (r1, 80-1) (80 bits requiring three 32-bit stack elements).

Figure 12-18. Left: Register File State before ALLOC_MEM Instructions; Right: Register File after ALLOC_MEM Instructions

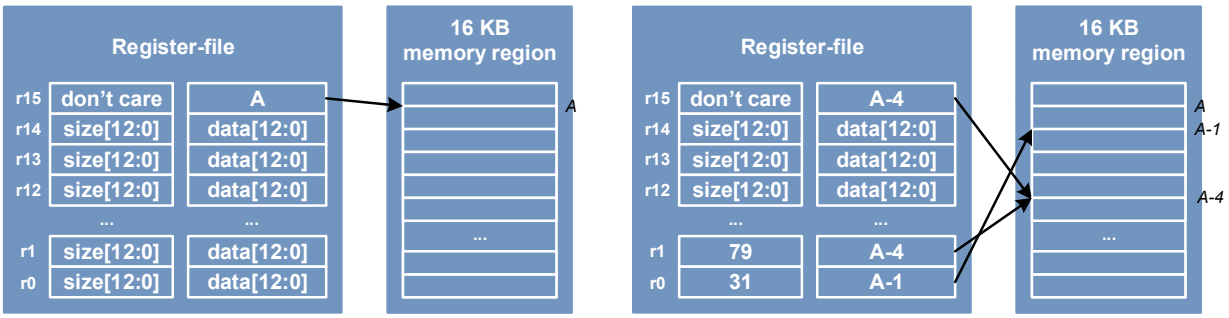
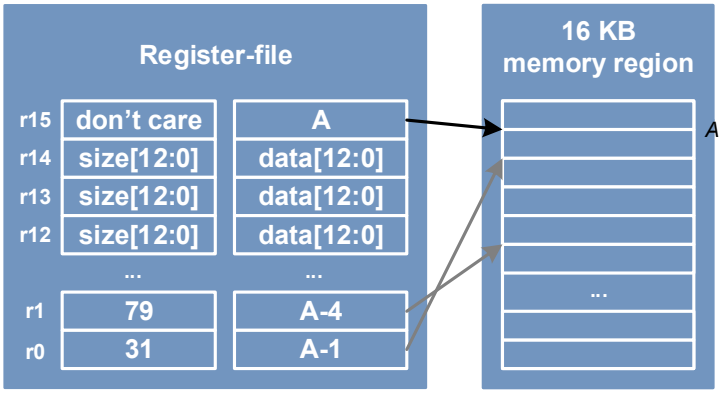


Figure 12-19 illustrates the VU component state after the execution of the instruction FREE_MEM ((1 << r1) | (1 << r0)).

Figure 12-19. Register File State after FREE_MEM Instruction



Freeing of stack elements should be in the reverse order of allocation of stack elements. Also see the description of the FREE_MEM instruction for the order in which registers are freed (lower registers are freed before higher registers).

12.10.3 Memory Operands

Asymmetric key cryptography requires computations on large integers of up to 1000's of bits. These integers are implemented using memory operands. The instruction registers' data values provide offsets in a memory region. The base address of the memory region is provided by VU_CTL.ADDR[31:8]. The memory region is located in either the IP memory buffer or the system memory. The memory region is accessed through the load and store FIFOs.

Memory operands are typically located in the crypto memory buffer (better access latency). Memory operand require one or multiple 32-bit words. The register size field specifies the bit size of a memory operand. This size is limited to the range [1, 8192] bits. Note that there is no size restriction within this range. A memory operand of n bits requires (n+31)/32 32-bit words.

12.10.4 Datapath

The VU instructions can operate on 13-bit register data values and/or large memory operands. The VU datapath is limited to 32 bits (the width of a single memory word).

To operate on large memory operands, multiple datapath iterations are required. Dependent on the instruction opcode, these iterations may be independent of each other (for example, a OR instruction) or may be dependent of each other (for example, an ADD instruction requires a carry). This complexity is completely hidden from software.

- The instruction opcode specifies whether datapath iterations are dependent or independent.
- The register size field provides the VU decoder and datapath with all the information it needs to determine the number of datapath iterations.

The VU datapath consists of three functional units:

- An ALU that performs addition, subtraction, and logical instructions.
- A barrel shifter that performs shift instructions.
- A multiplier that performs multiplication and squaring instructions.

In addition, a state machine is present for instruction decoding and controlling multiple datapath iterations. The state machine also performs administration instructions such as PUSH_REG, POP_REG, ALLOC_MEM, and FREE_MEM.

Note that the instruction execution time (in clock cycles) is typically independent of the instruction operand data values. However, the execution time is dependent on the size of memory operands (as specified by the register size field). This is an important characteristic, as the data value independence complicates differential power or execution time attacks.

12.10.5 Status Register

Similar to the Arm architecture, the VU has a STATUS register:

- It is affected by instruction execution.
- It controls instruction execution.

The STATUS register is a 4-bit field, with the following fields:

- CARRY field (bit 0). This field is set to ‘1’ if the result of an addition exceeds the destination operand size, if the result of a subtraction is positive or zero, or as the result of a shift instruction.
- EVEN field (bit 1). This field is set to ‘1’ (and cleared to ‘0’ otherwise) if the result of an instruction is even (bit 0 of the destination operand is ‘0’).
- ZERO field (bit 2). This field is set to ‘1’ (and cleared to ‘0’ otherwise) if the result of an instruction is “0”.
- ONE field (bit 3). This field is set to ‘1’ (and cleared to ‘0’ otherwise) if the result of an instruction is “1”.

The STATUS register fields are set and cleared as a result of instruction execution. Not all instructions affect all STATUS register fields.

Conditional execution. As mentioned, the STATUS register controls instruction execution. It does so through conditional instruction execution (again, similar to the Arm architecture). Almost all instructions (the SET_REG instruction is the exception) support conditional execution. Conditional execution only executes an instruction when a specific condition, as specified by a condition code in the instruction word, is met. The condition code is dependent on the STATUS register fields. The following table lists the condition codes and the conditions under which the conditional instruction is executed.

Table 12-28. Condition Codes

Abbreviation	Condition Code Value	Description	Condition
ALWAYS	0x0	Always	‘1’
EQ	0x1	Equal	ZERO
NE	0x2	Not equal	!ZERO
CS	0x3	Carry set/higher or same	CARRY
CC	0x4	Carry clear/lower	!CARRY
HI	0x5	Higher	CARRY & !ZERO
LS	0x6	Lower or same	!CARRY | ZERO
EVEN	0x7	Even	EVEN
ODD	0x8	Odd	!EVEN
ONE	0x9	One	ONE
NOT_ONE	0xa	Not one	!ONE

If a conditional instruction has a condition code that evaluates to ‘0’/FALSE, it does not change the VU functional state. This means that:

- a destination operand is not updated
- the STATUS register is not updated

12.10.6 Instructions

The VU instructions operate on either register operand data or memory operand data.

Operand types. Typically, VU instruction operands all have the same type: either register or memory operands. However, some exceptions do exist. For example, consider the following CTSAME (count trailing same) instruction.

```
CTSAME (r0, r1, r2)
```

This instruction counts the number of trailing bits (least significant bits), which are the same as the two memory operands that are identified by register r1 and r2. The instruction result is stored in the data field of register r0. This instruction has two source operands of the memory type and one destination operand of the register type.

Conditional execution. Conditional execution makes the execution of an instruction dependent on a condition. For example, consider the COND_CTSAME (HI, r0, r1, r2) instruction. This instruction uses the “HI” condition code. The CTSAME instruction is only executed when the “HI” condition code evaluates to ‘1’/TRUE. This is the case when the CARRY field of the STATUS register is ‘1’/TRUE and the ZERO field of the STATUS register is ‘0’/FALSE.

Unconditional execution uses the “ALWAYS” condition code, which always evaluates to ‘1’/TRUE. For brevity, and unconditional instruction XXX is not written as COND_XXX, but simply as XXX.

Note that a not executed conditional instruction does not update the STATUS register.

Memory operand data extension. Register operand data always has the same 13-bit size. Memory operand data has a size that is specified by a register’s size field. Therefore, it is possible to have memory operands with different sizes. This is intentional, but requires some rules in terms of how to deal with instructions that operate on memory operands with different sizes. These rules are as follows:

- If an instruction has a destination memory operand (for example, an ADD instruction), the size of this destination operand specifies the size of the instruction execution.
- If an instruction has no destination memory operand (for example, a CTSAME instruction), the maximum size of all source memory operands specifies the size of the instruction.
- If the instruction size (as determined by the previous two rules) is larger than a source memory operand, the memory operand is extended with leading ‘0’ bits (most significant ‘0’ bits).
- If the instruction size is smaller than a source memory operand, the memory operand’s leading bits (most significant bits) are dropped/ignored (the LSR, LSR1, and LSR1_WITH_CARRY are an exception to this rule).

To illustrate memory operand extension, we give a few examples.

```
CTSAME (r0, r1, r2)
r1 specifies a 40-bit memory operand 0x11:12345678
r2 specifies a 64-bit memory operand 0x11111111:ffffff78
```

The instruction size is 64 bits (maximum size of the two source memory operands). The memory operand 0x11:12345678 is extended to 0x00000011:12345678. The CTSAME instruction produces the result 8 in the register r0 data field.

```
ADD (r0, r1, r2)
r0 specifies a 48-bit memory operand
r1 specifies a 40-bit memory operand 0x11:12345678
r2 specifies a 64-bit memory operand 0xffffffff:ffffff78
```

The instruction size is 48 bits (size of the destination memory operand). The memory operand 0x11:12345678 is extended to 0x0011:12345678. The memory operand 0xffffffff:ffffff78 is reduced to 0xffff:ffffff78. The ADD instruction produces the 40-bit result 0x0011:123455f0 in the destination memory operand. Note that this instruction generates a carry and will set the CARRY field of the STATUS register to ‘1’/TRUE.

12.10.7 Instruction Set

This section describes all VU instructions. For each instruction, the following is described:

- The instruction format
  - Source operand types (register or memory operands)
  - Support for conditional execution
  - Encoding of the 32-bit instruction word (IW[31:0])

- Mnemonic
- Operation code
- Functionality

For instructions with memory operands, the source and destination memory operands may overlap, unless otherwise mentioned (the USQUARE, XSQUARE, UMUL, and XMUL instructions do not allow memory operand overlap).

Table 12-29. ALLOC_MEM Instruction

Instruction Format	**Mnemonic (rdst, imm13[12:0])** **rdst: register operand** **imm13: 13-bit immediate date**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[19:16] = rdst IW[12:0] = imm13	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

ALLOC_MEM	0x12	r15.data[12:0] = r15_data[12:0] – (imm13 >> 5) - 1;
		rdst.data[12:0] = r15_data[12:0];
		rdst.size[12:0] = imm13; // bit size minus ‘1’

Table 12-30. Instructions with Memory Operands, Category III

Instruction Format	**Mnemonic (imm16[15:0])** **imm16: 16-bit immediate date**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:0] = imm16	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

FREE_MEM	0x13	for (idx = 0; idx <= 15; idx++) {
		  if (imm16[idx]) {
		  imm13 = r[idx].size[12:0]
		  r15.data[12:0] = r15_data[12:0] + (imm13 >> 5) + 1;
		  }
		}

Table 12-31. Instructions with Register Operand Only, Category I

Instruction Format	**Mnemonic (rdst, imm13_0[13:0], imm13_1[11:0])** **rdst: register operand** **imm13_0: 13 bit immediate (for register data field)** **imm13_1: 13-bit immediate (for register size field)**	
Encoding	IW[31:30] = “operation code” IW[29:26] = rdst IW[25:13] = imm13_1 IW[12:0] = imm13_0	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
SET_REG	0x2	rdst.data[12:0] = imm13_1; rdst.size[12:0] = imm13_0; // bit size minus ‘1’

Table 12-32. MOV_REG Instruction

Instruction Format	**Mnemonic (rdst, rsrc) or COND_Mnemonic (cc, rdst, rsrc)** **rdst: register operand** **rsrc: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[3:0] = rsrc	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
MOV_REG	0x02	rdst.data = rsrc.data; rdst.size = rsrc.size;

Table 12-33. LD_REG Instruction

Instruction Format	**Mnemonic (rsrc1, rsrc0) or COND_Mnemonic (cc, rsrc1, rsrc0)** **rsrc1: register operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

LD_REG	0x00	word = GetMemData (r15.data+rsrc0, 32-1); // r15: stack pointer
		rsrc1.data[12:0] = word >> 16;
		rsrc1.size[12:0] = word;

Table 12-34. ST_REG Instruction

Instruction Format	**Mnemonic (rsrc1, rsrc0) or COND_Mnemonic (cc, rsrc1, rsrc0)** **rsrc1: register operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
ST_REG	0x01	word = rsrc1.data[12:0] << 16 | rsrc1.size[12:0] SetMemBuffData (r15.data+rsrc0, word, 32-1);

Table 12-35. Instructions with Register Operand Only, Category III

Instruction Format	**Mnemonic (rsrc1, rsrc0) or COND_Mnemonic (cc, rsrc1, rsrc0)** **rsrc0: register operand** **rsrc1: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

SWAP_REG	0x03	data = rsrc1.data; size = rsrc1.size;
		rsrc1.data = rsrc0.data; rsrc1.size = rsrc0.size;
		rsrc0.data = data; rsrc0.size = size;

Table 12-36. Instructions with Register Operand Only, Category IV

Instruction Format	**Mnemonic () or COND_Mnemonic (cc)** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

PUSH_REG	0x10	r15.data -= 15; // r15: stack pointer
		word = r0.data[12:0] << 16 | r0.size[12:0]; Mem-Buff[r15.data+0] = word;
		word = r1.data[12:0] << 16 | r1.size[12:0]; Mem-Buff[r15.data+1] = word;
		…
		word = r14.data[12:0] << 16 | r14.size[12:0]; Mem-Buff[r15.data+14] = word;

POP_REG	0x11	word = GetMemData (r15.data+0, 32-1);
		r0.data = word >> 16; r0.size = word;
		word = MemBuff[r15.data+1];
		r1.data[12:0] = word >> 16; r1.size[12:0] = word;
		…
		word = MemBuff[r15.data+14];
		r14.data[12:0] = word >> 16; r14.size[12:0] = word;
		r15.data += 15;

Table 12-37. Instructions with Register Operand Only, Category V

Instruction Format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: register operand** **rsrc1: register operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
ADD_REG	0x06	rdst.data[12:0] = rsrc1.data[12:0] + rsrc0.data[12:0];
SUB_REG	0x07	rdst.data[12:0] = rsrc1.data[12:0] – rsrc0.data[12:0];
OR_REG	0x08	rdst.data[12:0] = rsrc1.data[12:0] | rsrc0.data[12:0];
AND_REG	0x09	rdst.data[12:0] = rsrc1.data[12:0] & rsrc0.data[12:0];
XOR_REG	0x0a	rdst.data[12:0] = rsrc1.data[12:0] ^ rsrc0.data[12:0];
NOR_REG	0x0b	rdst.data[12:0] = ~(rsrc1.data[12:0] | rsrc0.data[12:0]);
NAND_REG	0x0c	rdst.data[12:0] = ~(rsrc1.data[12:0] & rsrc0.data[12:0]);
MIN_REG	0x0d	rdst.data[12:0] = Minimum (rsrc1.data[12:0], rsrc0.data[12:0]);
MAX_REG	0x0e	rdst.data[12:0] = Maximum (rsrc1.data[12:0], rsrc0.data[12:0]);

Table 12-38. MOV_REG_TO_STATUS

Instruction Format	**Mnemonic (rsrc) or COND_Mnemonic (cc, rsrc)** **rsrc: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[3:0] = rsrc	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
MOV_REG_TO_STATUS	0x04	STATUS.CARRY = rsrc.data[0]; STATUS.EVEN = rsrc.data[1]; STATUS.ZERO = rsrc.data[2]; STATUS.ONE = rsrc.data[3];

Table 12-39. MOV_STATUS_TO_REG

Instruction Format	**Mnemonic (rdst) or COND_Mnemonic (cc, rdst)** **rdst: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

MOV_STATUS_TO_REG	0x05	rdst.data= 0;
		rdst.data[0] = STATUS.CARRY;
		rdst.data[1] = STATUS.EVEN;
		rdst.data[2] = STATUS.ZERO;
		rdst.data[3] = STATUS.ONE;

Table 12-40. MOV_IMM_TO_STATUS

Instruction Format	**Mnemonic (imm4) or COND_Mnemonic (cc, imm4)** **imm4: immediate value** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[3:0] = imm4	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
MOV_IMM_TO_STATUS	0x01	STATUS.CARRY = imm[0]; STATUS.EVEN = imm[1]; STATUS.ZERO = imm[2]; STATUS.ONE = imm[3];

Table 12-41. Instructions with Mixed Operands, Category I

Instruction Format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: memory buffer operand** **rsrc1: memory buffer operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Shared Functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

LSL	0x20	src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, rdst.size);
		dst_data = src1_data << rsrc0.data[12:0];
		if (rsrc0.data[12:0] != 0) STATUS.CARRY = dst_-
		data[rdst.size +1];
		else STATUS.CARRY = 0
		SetMemBuffData (rdst.data, dst_data, rdst.size);

LSR	0x23	src1_data = GetMemData (rsrc1.data, rsrc1.size);
		dst_data = src1_data >> rsrc0.data[12:0];
		if (rsrc0.data[12:0] != 0) STATUS.CARRY = dst_data[-1];
		else STATUS.CARRY = 0;
		SetMemBuffData (rdst.data, dst_data, rdst.size);

Table 12-42. Instructions with Mixed Operands, Category II

Instruction Format	**Mnemonic (rdst, rsrc) or COND_Mnemonic (cc, rdst, rsrc)** **rdst: memory buffer operand** **rsrc: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc	
Shared Functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

LSL1	0x21	src_data = GetMemData (rsrc.data, rsrc.size);
		src_data = SizeAdjust (src_data, rdst.size);
		dst_data = (src_data << 1);
		STATUS.CARRY = dst_data[rdst.size +1];
		SetMemBuffData (rdst.data, dst_data, rdst.size);

LSL1_WITH_CARRY	0x22	src_data = GetMemData (rsrc.data, rsrc.size);
		src_data = SizeAdjust (src_data, rdst.size);
		dst_data = (src_data << 1) | STATUS.CARRY;
		STATUS.CARRY = dst_data[rdst.size +1];
		SetMemBuffData (rdst.data, dst_data, rdst.size);

LSR1	0x24	src_data = GetMemData (rsrc.data, rsrc.size);
		dst_data = src_data >> 1;
		STATUS.CARRY = dst_data[-1];
		SetMemBuffData (rdst.data, dst_data, rdst.size);

LSR1_WITH_CARRY	0x25	src_data = GetMemData (rsrc.data, rsrc.size); dst_data = src_data >> 1; dst_data[rdst.size] = STA- TUS.CARRY; STATUS.CARRY = dst_data[-1]; SetMemBuffData (rdst.data, dst_data, rdst.size);

Table 12-43. Instructions with Mixed Operands, Category III

Instruction Format	**Mnemonic (rdst, rsrc) or COND_Mnemonic (cc, rdst, rsrc)** **rdst: memory buffer operand** **rsrc: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[3:0] = rsrc	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

SET_BIT	0x28	dst_data = GetMemData (rdst.data, rdst.size);
		dst_data[rsrc.data[12:0]] = 1;
		SetMemBuffData (rdst.data, dst_data, rdst.size);

CLR_BIT	0x29	dst_data = GetMemData (rdst.data, rdst.size);
		dst_data[rsrc.data[12:0]] = 0;
		SetMemBuffData (rdst.data, dst_data, rdst.size);

INV_BIT	0x2a	dst_data = GetMemData (rdst.data, rdst.size);
		dst_data[rsrc.data[12:0]] = !dst_data[rsrc.data[12:0]];
		SetMemBuffData (rdst.data, dst_data, rdst.size);

Table 12-44. Instructions with Mixed Operands, Category IV

Instruction Format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: register operand** **rsrc1: memory buffer operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

GET_BIT	0x2b	src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_bit = src1_data[rsrc0.data[12:0]];
		rdst.data = src1_bit;
		STATUS.CARRY = src1_bit;

Table 12-45. Instructions with Mixed Operands, Category V

Instruction Format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: register operand** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Shared Functionality	STATUS.ZERO = (src0_data[instr_size:0] == src1_data[instr_size:0]) Note that for 8192-bit operands, two equal operands result in rdst.data[12:0] = “0”. The ZERO cause field can be used to identify this situation.	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

CLSAME	0x26	instr_size = Maximum (rsrc1.size, rsrc0.size);
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		temp_data = src1_data ^ src0_data;
		for (idx = instr_size-1; idx >= 0; idx--) {
		if (temp_data[idx] == 1) break;
		}
		rdst.data[12:0] = instr_size - idx - 1;

CTSAME	0x27	instr_size = Maximum (rsrc1.size, rsrc0.size);
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		temp_data = src1_data ^ src0_data;
		for (idx = 0; idx < instr_size; idx++) {
		if (temp_data[idx] == 1) break;
		}
		rdst.data[12:0] = idx;

Table 12-46. Instructions with Memory Operands, Category I

Instruction Format	**Mnemonic (rdst) or COND_Mnemonic (cc, rdst)** **rdst: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst	
Shared Functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

SET_TO_ZERO	0x34	instr_size = rdst.size;
		dst_data = 0;
		SetMemBuffData (rdst.data, dst_data, instr_size);

SET_TO_ONE	0x35	instr_size = rdst.size;
		dst_data = 1;
		SetMemBuffData (rdst.data, dst_data, instr_size);

Table 12-47. Instructions with Memory Operands, Category II

Instruction Format	**Mnemonic (rdst, rsrc) or COND_Mnemonic (cc, rdst, rsrc)** **rdst: memory buffer operand** **rsrc: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[3:0] = rsrc	
Shared Functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

MOV	0x30	instr_size = rdst.size;
		src_data = GetMemData (rsrc.data, rsrc.size);
		src_data = SizeAdjust (src_data, instr_size);
		dst_data = src_data;
		SetMemBuffData (rdst.data, dst_data, instr_size);

XSQUARE	0x31	instr_size = rdst.size;
		src_data = GetMemData (rsrc.data, rsrc.size);
		src_data = SizeAdjust (src_data, instr_size);
		dst_data = PolynomialMultiplication (src_data, src_data);
		SetMemBuffData (rdst.data, dst_data, instr_size);
		Note: The source and destination memory operands must not overlap.

USQUARE	0x2f	instr_size = rdst.size;
		src_data = GetMemData (rsrc.data, rsrc.size);
		src_data = SizeAdjust (src_data, instr_size);
		dst_data = src_data * src_data;
		SetMemBuffData (rdst.data, dst_data, instr_size);
		Note: The source and destination memory operands must not overlap.

Table 12-48. Instructions with Memory Operands, Category III

Instruction Format	**Mnemonic (rsrc1, rsrc0) or COND_Mnemonic (cc, rsrc1, rsrc0)** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

CMP_SUB	0x3d	instr_size = Maximum (rsrc1.size, rsrc0.size);
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		dst_data = src1_data – src0_data;
		STATUS.CARRY = (src1_data >= src0_data);
		STATUS.EVEN = (dst_data[0] == 0)
		STATUS.ZERO = (dst_data[instr_size:0] == 0)
		STATUS.ONE = (dst_data[instr_size:0] == 1)

CMP_DEGREE	0x3e	instr_size = Maximum (rsrc1.size, rsrc0.size);
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		STATUS.CARRY = PolynomialDegree (src1_data) >= Polynomi-
		alDegree (src0_data);
		STATUS.ZERO = PolynomialDegree (src1_data) == Polynomial-
		Degree (src0_data);

Table 12-49. Instructions with Memory Operands, Category IV

Instruction Format	**Mnemonic (rsrc) or COND_Mnemonic (cc, rsrc)** **rsrc: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[3:0] = rsrc	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

TST	0x3f	instr_size = rsrc.size;
		src_data = GetMemData (rsrc.data, rsrc.size);
		dst_data = src_data;
		STATUS.CARRY = 0; // always set to ‘0’!!!
		STATUS.EVEN = (dst_data[0] == 0)
		STATUS.ZERO = (dst_data[instr_size:0] == 0)
		STATUS.ONE = (dst_data[instr_size:0] == 1)

Table 12-50. Instructions with Memory Operands, Category V

Instruction Format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: memory buffer operand** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Shared Functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

XMUL	0x32	instr_size = rdst.size;
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		dst_data = PolynomialMultiplication (src1_data, src0_-
		data);
		SetMemBuffData (rdst.data, dst_data, instr_size);
		Note: The source and destination memory operands must not overlap. When one of the operands is 32 or less bits in size, performance is best when rsrc0 is used for this operand.

UMUL	0x33	instr_size = rdst.size;
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		dst_data = src1_data * src0_data;
		SetMemBuffData (rdst.data, dst_data, instr_size);
		Note: The source and destination memory operands must not overlap. When one of the operands is 32 or less bits in size, performance is best when rsrc0 is used for this operand.

ADD	0x36	instr_size = rdst.size;
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		dst_data = src1_data + src0_data;
		STATUS.CARRY = (dst_data >= (1 << instr_size));
		SetMemBuffData (rdst.data, dst_data, instr_size);

Table 12-50. Instructions with Memory Operands, Category V

Instruction Format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: memory buffer operand** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	

SUB	0x37	instr_size = rdst.size;
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		dst_data = src1_data - src0_data;
		STATUS.CARRY = (src1_data >= src0_data);
		SetMemBuffData (rdst.data, dst_data, instr_size);

ADD_WITH_CARRY	0x14	instr_size = rdst.size;
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		dst_data = src1_data + src0_data + STATUS.CARRY;
		STATUS.CARRY = (dst_data >= (1 << instr_size));
		SetMemBuffData (rdst.data, dst_data, instr_size);

SUB_WITH_CARRY	0x15	instr_size = rdst.size;
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		dst_data = src1_data - src0_data – !STATUS.CARRY;
		STATUS.CARRY = (src1_data >= src0_data);
		SetMemBuffData (rdst.data, dst_data, instr_size);

OR, AND, XOR, NOR, NAND	0x38, 0x39, 0x3a, 0x3b, 0x3c	instr_size = rdst.size;
		src0_data = GetMemData (rsrc0.data, rsrc0.size);
		src0_data = SizeAdjust (src0_data, instr_size);
		src1_data = GetMemData (rsrc1.data, rsrc1.size);
		src1_data = SizeAdjust (src1_data, instr_size);
		OR: dst_data = src1_data | src0_data;
		AND: dst_data = src1_data & src0_data;
		XOR: dst_data = src1_data ^ src0_data;
		NOR: dst_data = ~(src1_data | src0_data);
		NAND: dst_data = ~(src1_data & src0_data);
		SetMemBuffData (rdst.data, dst_data, instr_size);

Table 12-51. Instructions with Memory Operands, Category VI

Instruction Format	**Mnemonic (rdst, imm12) or COND_Mnemonic (cc, rdst, imm12)** **rdst: memory buffer operand** **rsrc: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[19:16] = rdst IW[12:0] = imm13	
Mnemonic	**Operation Code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**

SET_BIT_IMM	0x2c	dst_data = GetMemData (rdst.data, rdst.size);
		dst_data[imm13] = 1;
		SetMemBuffData (rdst.data, dst_data, rdst.size);

CLR_BIT_IMM	0x2d	dst_data = GetMemData (rdst.data, rdst.size);
		dst_data[imm13] = 0;
		SetMemBuffData (rdst.data, dst_data, rdst.size);

INV_BIT_IMM	0x2e	dst_data = GetMemData (rdst.data, rdst.size);
		dst_data[imm13] = !dst_data[imm13];
		SetMemBuffData (rdst.data, dst_data, rdst.size);

13. Program and Debug Interface

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU Program and Debug interface provides a communication gateway for an external device to perform programming or debugging. The external device can be a Cypress-supplied programmer and debugger, or a third-party device that supports programming and debugging. The serial wire debug (SWD) or the JTAG interface can be used as the communication protocol between the external device and PSoC 6 MCUs.

13.1 Features

- Supports programming and debugging through the JTAG or SWD interface.
- CM4 supports 4-bit ETM tracing, serial wire viewer (SWV), and printf() style debugging through the single-wire output (SWO) pin. CM0+ supports Micro Trace Buffer (MTB) with 4 KB dedicated RAM.
- Supports Cross Triggering Interface (CTI) and Cross Triggering Matrix (CTM).
- CM0+ supports four hardware breakpoints and two watchpoints. CM4 supports six hardware breakpoints and four watchpoints.
- Provides read and write access to all memory and registers in the system while debugging, including the Cortex-M4 and Cortex-M0+ register banks when the core is running or halted

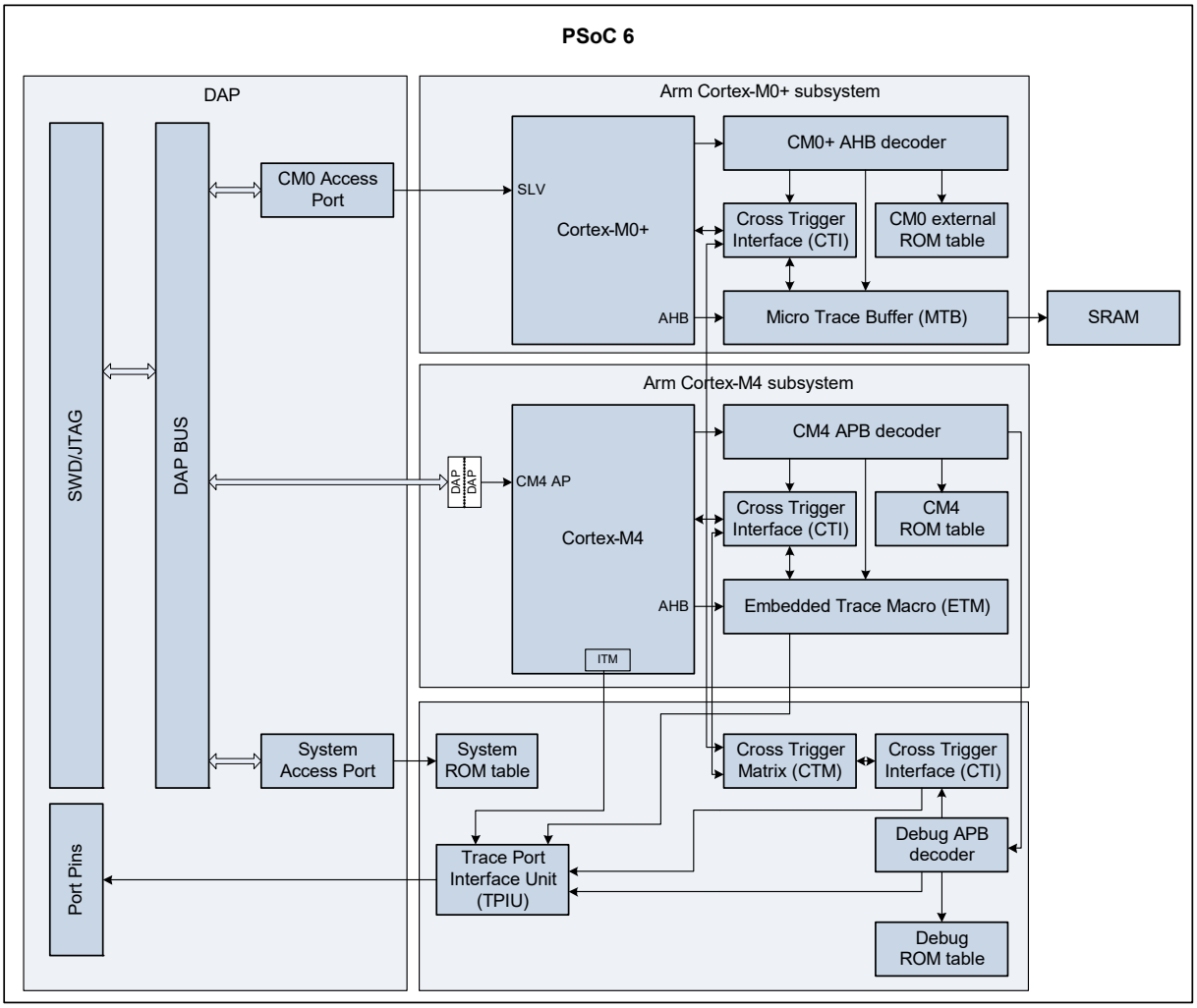
13.2 Architecture

Figure 13-1 shows the block diagram of the program and debug interface in the PSoC 6 MCU. The debug and access port (DAP) acts as the program and debug interface. The external programmer or debugger, also known as the “host”, communicates with the DAP of the PSoC 6 MCU “target” using either the SWD or JTAG interface. The debug physical port pins communicate with the DAP through the high-speed I/O matrix (HSIOM). See the I/O System chapter on page 261 for details on HSIOM.

The debug infrastructure is organized in the following four groups:

- DAP (provides pin interfaces through which the debug host can connect to the chip)
- Cortex-M0+ core debug components
- Cortex-M4 core debug components
- Other debug infrastructure (includes the CM4 tracing, the CTM, and the System ROM table)

Figure 13-1. Program and Debug Interface



The DAP communicates with the Cortex-M0+ CPU using the Arm-specified advanced high-performance bus (AHB) interface. AHB is the systems interconnect protocol used inside the device, which facilitates memory and peripheral register access by the AHB master. The PSoC 6 MCU has six AHB masters – Arm CM4 CPU core, Arm CM0 CPU core, Datawire0, Datawire1, Crypto, and DAP. The external host can effectively take control of the entire device through the DAP to perform programming and debugging operations.

The following are the various debug and trace components:

- Debug components
  - JTAG and SWD for debug control and access
- Trace source components
  - Micro trace buffer (MTB-M0+) for tracing Cortex-M0+ program execution
  - Embedded trace macrocell (ETM-M4) for tracing Cortex-M4 program execution
- Trace sink components
  - Trace port interface unit (TPIU) to drive the trace information out of the chip to an external trace port analyzer
- Cross-triggering components
  - Cross-trigger interface (CTI)
  - Cross-trigger matrix (CTM)
- ROM tables

13.2.1 Debug Access Port (DAP)

The DAP consists of a combined SWD/JTAG interface (SWJ) that also includes the SWD listener. The SWD listener decides whether the JTAG interface (default) or SWD interface is active. Note that JTAG and SWD are mutually exclusive because they share pins.

The debug port (DP) connects to the DAP bus, which in turn connects to one of three Access Ports (AP), namely:

- The CM0-AP, which connects directly to the AHB debug slave port (SLV) of the CM0+ and gives access to the CM0+ internal debug components. This also allows access to the rest of the system through the CM0+ AHB master interface. This provides the debug host the same view as an application running on the CM0+. This includes access to the MMIO of other debug components of the Cortex M0+ subsystem. These debug components can also be accessed by the CM0+ CPU, but cannot be reached through the other APs or by the CM4 core.
- The CM4-AP located inside the CM4 gives access to the CM4 internal debug components. The CM4-AP also allows access to the rest of the system through the CM4 AHB master interfaces. This provides the debug host the same view as an application running on the CM4 core. Additionally, the CM4-AP provides access to the debug components in the CM4 core through the External Peripheral Bus (EPB). These debug components can also be accessed by the CM4 CPU, but cannot be reached through the other APs or by the CM0+ core.
- The System-AP, which through an AHB mux gives access to the rest of the system. This allows access to the System ROM table, which cannot be reached any other way. The System ROM table provides the chip ID but is otherwise empty.

13.2.1.1 DAP Security

For security reasons all three APs each can be independently disabled. Each AP disable is controlled by two MMIO bits. One bit, CPUSS_AP_CTL.xxx_DISABLE (where xxx can be CM0 or CM4 or SYS), can be set during boot, before the debugger can connect, based on eFuse settings. After this bit is set it cannot be cleared.

The second bit, CPUSS_AP_CTL.xxx_ENABLE, is a regular read/write bit. This bit also resets to zero and is set to ‘1’ by either the ROM boot code or the flash boot code depending on the life-cycle stage. This feature can be used to block debug access during normal operation, but re-enable some debug access after a successful authentication.

In addition to the above, the System AP is also protected by an MPU. This can be used to give the debugger limited access to the rest of the system. Allow access to the System ROM table for chip identification. If debug access is restored after successful authentication, this MPU must be configured to allow authentication requests.

Note: The debug slave interfaces of both the CPUs bypass the internal CPU MPU.

13.2.1.2 DAP Power Domain

Almost all the debug components are part of the Active power domain. The only exception is the SWD/JTAG-DP, which is part of the Deep Sleep power domain. This allows the debug host to connect during Deep Sleep, while the application is ‘running’ or powered down. This enables in-field debugging for low-power applications in which the chip is mostly in Deep Sleep.

After the debugger is connected to the chip, it must bring the chip to the Active state before any operation. For this, the SWD/JTAG-DP has a register (DP_CTL_STAT) with two power request bits. The two bits are CDBGPWRUPREQ and CSYSPWRUPREQ, which request for debug power and system power, respectively. These bits must remain set for the duration of the debug session.

Note that only the two SWD pins (SWCLKTCK and SWDIOTMS) are operational during the Deep Sleep mode – the JTAG pins are operational only in Active mode. The JTAG debug and JTAG boundary scan are not available when the system is in Deep Sleep mode. JTAG functionality is available only after a chip power-on-reset.

13.2.2 ROM Tables

The ROM tables are organized in a tree hierarchy. Each AP has a register that contains a 32-bit address pointer to the base of the root ROM table for that AP. For PSoC 6 MCUs, there are three such root ROM tables.

Each ROM table contains 32-bit entries with an address pointer that either points to the base of the next level ROM table. Each ROM table also contains a set of ID registers that hold JEDEC compliant identifiers to identify the manufacturer, part number, and major and minor revision numbers. For all ROM tables in PSoC 6 MCUs, these IDs are the same. Each ROM table and CoreSight compliant component also contains component identification registers.

13.2.3 Trace

The micro trace buffer (MTB-M0+) component captures the program execution flow from Cortex-M0+ CPU and stores it in a local SRAM memory. This information can be read by an external debug tool through JTAG/SWD interface to construct the program execution flow.

The embedded trace macro (ETM) component connected to Cortex-M4 captures the program execution flow from Cortex-M4 CPU and generates trace output on its advanced trace bus (ATB) interface. The instrumentation trace macrocell (ITM), which is inside Cortex-M4, also generates trace output on its ATB interface. These two ATB interfaces (from ETM-M4 and ITM) are connected a trace port interface unit (TPIU).

The TPIU drives the external pins of a trace port (through IOSS interface), so that the trace can be captured by an external trace port analyzer (TPA). For more details, refer to the *Arm Debug Interface Architecture Specification ADIv5.0 to ADIv5.2*.

13.2.4 Embedded Cross Triggering

The Arm CoreSight includes Embedded Cross Triggering (ECT) to communicate events between debug components. These events are particularly useful with tracing and multicore platforms. For example trigger events can be used to:

- Start or stop both CPUs at (almost) the same time
- Start or stop instruction tracing based on trace buffer being full or not or based on other events

CoreSight uses two components to support ECT, namely a CTI and a CTM, both of which are used in PSoC 6 MCUs.

The CTI component interfaces with other debug components, sending triggers back and forth and synchronizing them as needed. The CTM connects several CTIs, thus allowing events to be communicated from one CTI to another.

The PSoC 6 MCU has three CTIs, one for each CPU and one for the trace components in the debug structure. These three CTIs are connected together through the CTM. The CM4 CTI is located in the fast clock domain and the other two CTIs and the CTM are all located in the same slow-frequency clock domain. For more details, refer to the Arm documentation.

13.3 Serial Wire Debug (SWD) Interface

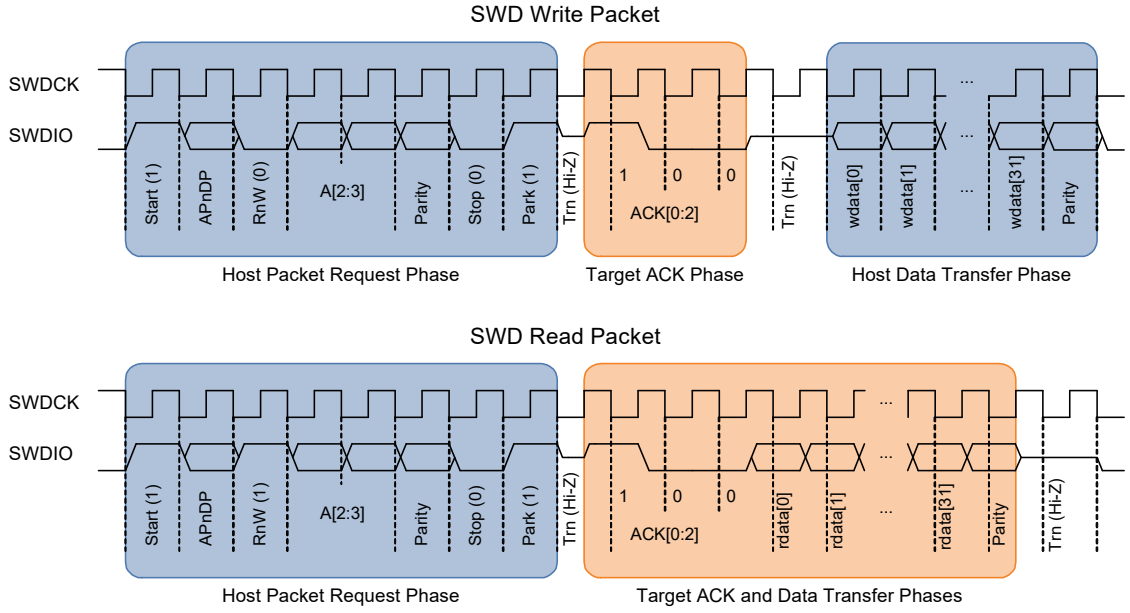
The PSoC 6 MCU supports programming and debugging through the SWD interface. The SWD protocol is a packet-based serial transaction protocol. At the pin level, it uses a single bidirectional data signal (SWDIO) and a unidirectional clock signal (SWDCK). The host programmer always drives the clock line, whereas either the host or the target drives the data line. A complete data transfer (one SWD packet) requires 46 clocks and consists of three phases:

- **Host Packet Request Phase** – The host issues a request to the PSoC 6 MCU target.
- **Target Acknowledge Response Phase** – The PSoC 6 MCU target sends an acknowledgement to the host.
- **Data Transfer Phase** – The host or target writes data to the bus, depending on the direction of the transfer.

When control of the SWDIO line passes from the host to the target, or vice versa, there is a turnaround period (Trn) where neither device drives the line and it floats in a high-impedance (Hi-Z) state. This period is either one-half or one and a half clock cycles, depending on the transition.

Figure 13-2 shows the timing diagrams of read and write SWD packets.

Figure 13-2. SWD Write and Read Packet Timing Diagrams



The sequence to transmit SWD read and write packets are as follows:

1. Host Packet Request Phase: SWDIO driven by the host
   a. The start bit initiates a transfer; it is always logic 1.
   b. The AP not DP (APnDP) bit determines whether the transfer is an AP access – 1b or a DP access – 0b.
   c. The Read not Write bit (RnW) controls which direction the data transfer is in. 1b represents a 'read from' the target, or 0b for a 'write to' the target.
   d. The Address bits (A[3:2]) are register select bits for AP or DP, depending on the APnDP bit value. **Note:** Address bits are transmitted with the LSb first.
   e. The parity bit contains the parity of APnDP, RnW, and ADDR bits. It is an even parity bit; this means, when XORed with the other bits, the result will be 0.

      If the parity bit is not correct, the header is ignored by the PSoC 6 MCU; there is no ACK response (ACK = 111b). The programming operation should be aborted and retried again by following a device reset.
   f. The stop bit is always logic 0.
   g. The park bit is always logic 1.
2. Target Acknowledge Response Phase: SWDIO driven by the target
   a. The ACK[2:0] bits represent the target to host response, indicating failure or success, among other results. See Table 13-1 for definitions. **Note:** ACK bits are transmitted with the LSb first.
3. Data Transfer Phase: SWDIO driven by either target or host depending on direction
   a. The data for read or write is written to the bus, LSb first.
   b. The data parity bit indicates the parity of the data read or written. It is an even parity; this means when XORed with the data bits, the result will be 0.

      If the parity bit indicates a data error, corrective action should be taken. For a read packet, if the host detects a parity error, it must abort the programming operation and restart. For a write packet, if the target detects a parity error, it generates a FAULT ACK response in the next packet.

According to the SWD protocol, the host can generate any number of SWDCK clock cycles between two packets with SWDIO low. Three or more dummy clock cycles should be generated between two SWD packets if the clock is not free-running or to make the clock free-running in IDLE mode.

The SWD interface can be reset by clocking the SWDCK line for 50 or more cycles with SWDIO high. To return to the idle state, clock the SWDIO low once.

13.3.1 SWD Timing Details

The SWDIO line is written to and read at different times depending on the direction of communication. The host drives the SWDIO line during the Host Packet Request phase and, if the host is writing data to the target, during the Data Transfer phase as well. When the host is driving the SWDIO line, each new bit is written by the host on falling SWDCK edges, and read by the target on rising SWDCK edges. The target drives the SWDIO line during the Target Acknowledge Response phase and, if the target is reading out data, during the Data Transfer phase as well. When the target is driving the SWDIO line, each new bit is written by the target on rising SWDCK edges, and read by the host on falling SWDCK edges.

Table 13-1 and Figure 13-2 illustrate the timing of SWDIO bit writes and reads.

Table 13-1. SWDIO Bit Write and Read Timing

SWD Packet Phase	SWDIO Edge	
	Falling	Rising
Host Packet Request	Host Write	Target Read
Host Data Transfer		
Target Ack Response	Host Read	Target Write
Target Data Transfer		

13.3.2 ACK Details

The acknowledge (ACK) bitfield is used to communicate the status of the previous transfer. OK ACK means that previous packet was successful. A WAIT response requires a data phase. For a FAULT status, the programming operation should be aborted immediately. Table 13-2 shows the ACK bit-field decoding details.

Table 13-2. SWD Transfer ACK Response Decoding

Response	ACK[2:0]
OK	001b
WAIT	010b
FAULT	100b
NO ACK	111b

Details on WAIT and FAULT response behaviors are as follows:

- For a WAIT response, if the transaction is a read, the host should ignore the data read in the data phase. The target does not drive the line and the host must not check the parity bit as well.
- For a WAIT response, if the transaction is a write, the data phase is ignored by the PSoC 6 MCU. But, the host must still send the data to be written to complete the packet. The parity bit corresponding to the data should also be sent by the host.
- A WAIT response means that the PSoC 6 MCU is processing the previous transaction. The host can try for a maximum of four continuous WAIT responses to see whether an OK response is received. If it fails, then the

programming operation should be aborted and retried again.

- For a FAULT response, the programming operation should be aborted and retried again by doing a device reset.

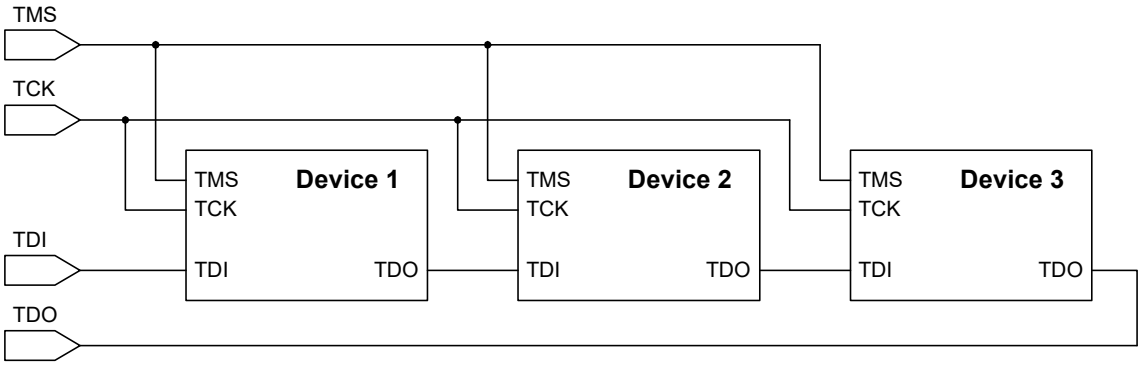
13.3.3 Turnaround (Trn) Period Details

There is a turnaround period between the packet request and the ACK phases, as well as between the ACK and the data phases for host write transfers, as shown in Figure 13-2. According to the SWD protocol, the Trn period is used by both the host and target to change the drive modes on their respective SWDIO lines. During the first Trn period after the packet request, the target starts driving the ACK data on the SWDIO line on the rising edge of SWDCK. This action ensures that the host can read the ACK data on the next falling edge. Thus, the first Trn period lasts only one-half cycle. The second Trn period of the SWD packet is one and a half cycles. Neither the host nor the PSoC 6 MCU should drive the SWDIO line during the Trn period.

13.4 JTAG Interface

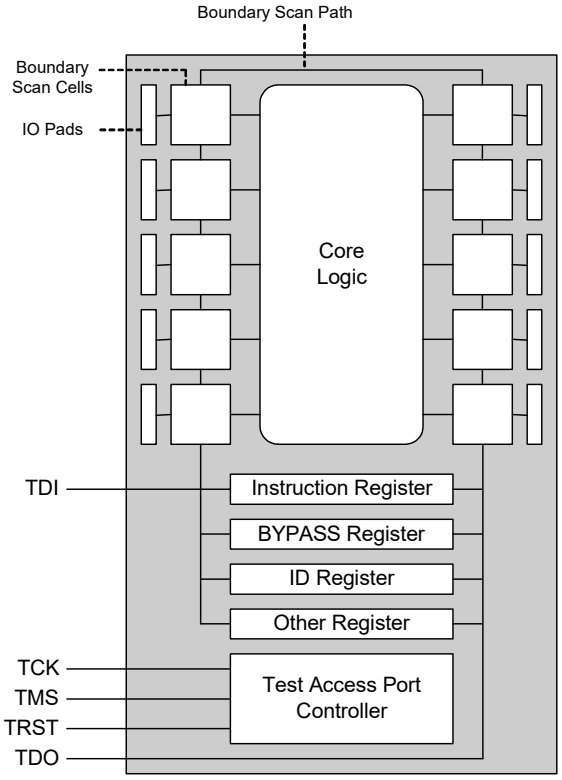
In response to higher pin densities on ICs, the Joint Test Action Group (JTAG) proposed a method to test circuit boards by controlling the pins on the ICs (and reading their values) via a separate test interface. The solution, later formalized as IEEE Standard 1149.1-2001, is based on the concept of a serial shift register routed across all of the pins of the IC – hence the name “boundary scan.” The circuitry at each pin is supplemented with a multipurpose element called a boundary scan cell. In PSoC 6 MCUs, most GPIO port pins have a boundary scan cell associated with them (see the GPIO block diagrams in the I/O System chapter on page 261). The interface used to control the values in the boundary scan cells is called the Test Access Port (TAP) and is commonly known as the JTAG interface. It consists of three signals: Test Data In (TDI), Test Data Out (TDO), and Test Mode Select (TMS). Also included is a clock signal (TCK) that clocks the other signals. TDI, TMS, and TCK are all inputs to the device and TDO is the output from the device. This interface enables testing multiple ICs on a circuit board, in a daisy-chain fashion, as shown in Figure 13-3.

Figure 13-3. JTAG Interface to Multiple ICs on a Circuit Board



The JTAG interface architecture within each device is shown in Figure 13-4. Data at TDI is shifted in, through one of several available registers, and out to TDO.

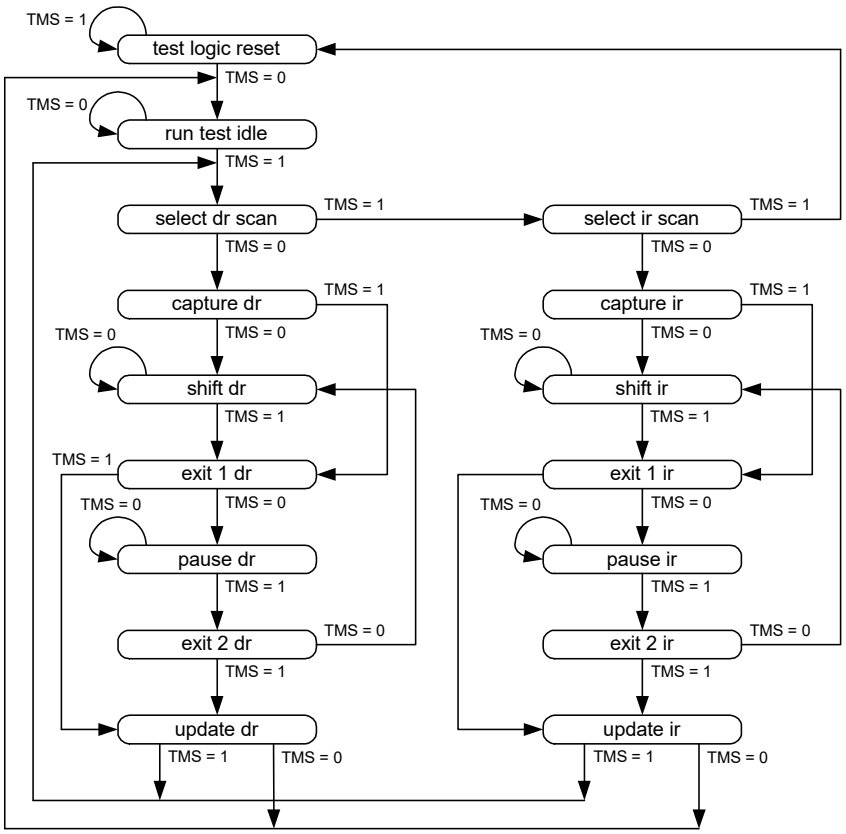
Figure 13-4. JTAG Interface Architecture



The TMS signal controls a state machine in the TAP. The state machine controls which register (including the boundary scan path) is in the TDI-to-TDO shift path, as shown in Figure 13-5. The following terms apply:

- IR - the instruction register
- DR - one of the other registers (including the boundary scan path), as determined by the contents of the instruction register
- capture - transfer the contents of a DR to a shift register, to be shifted out on TDO (read the DR)
- update - transfer the contents of a shift register, shifted in from TDI, to a DR (write the DR)

Figure 13-5. TAP State Machine



The registers in the TAP are:

- Instruction – Typically two to four bits wide, holds the current instruction that defines which data register is placed in the TDI-to-TDO shift path.
- Bypass – one bit wide, directly connects TDI with TDO, causing the device to be bypassed for JTAG purposes.
- ID – 32 bits wide, used to read the JTAG manufacturer/part number ID of the device.
- Boundary Scan Path (BSR) – Width equals the number of I/O pins that have boundary scan cells, used to set or read the states of those I/O pins.

Other registers may be included in accordance with the device manufacturer specifications. The standard set of instructions (values that can be shifted into the instruction register), as specified in IEEE 1149, are:

- EXTEST – Causes TDI and TDO to be connected to the BSR. The device is changed from its normal operating mode to a test mode. Then, the device's pin states can be sampled using the capture dr JTAG state, and new values can be applied to the pins of the device using the update dr state.
- SAMPLE – Causes TDI and TDO to be connected to the BSR, but the device remains in its normal operating mode. During this instruction, the BSR can be read by the capture dr JTAG state to take a sample of the functional data entering and leaving the device.
- PRELOAD – Causes TDI and TDO to be connected to the BSR, but the device is left in its normal operating mode. The instruction is used to preload test data into the BSR before loading an EXTEST instruction.

Optional, but commonly available, instructions are:

- IDCODE – Causes TDI and TDO to be connected to an IDCODE register.
- INTEST – Causes TDI and TDO to be connected to the BSR. While the EXTEST instruction allows access to the device pins, INTEST enables similar access to the corelogic signals of a device

For more information, see the IEEE Standard, available at www.ieee.org.

13.5 Programming the PSoC 6 MCU

The PSoC 6 MCU is programmed using the following sequence. Refer to the *PSoC 6 MCU Programming Specifications* for complete details on the programming algorithm, timing specifications, and hardware configuration required for programming.

1. Acquire the SWD port in the PSoC 6 MCU.
2. Enter the programming mode.
3. Execute the device programming routines such as Silicon ID Check, Flash Programming, Flash Verification, and Checksum Verification.

13.5.1 SWD Port Acquisition

13.5.1.1 SWD Port Acquire Sequence

The first step in device programming is for the host to acquire the target's SWD port. The host first performs a device reset by asserting the external reset (XRES) pin. After removing the XRES signal, the host must send an SWD connect sequence for the device within the acquire window to connect to the SWD interface in the DAP.

The debug access port must be reset using the standard Arm command. The DAP reset command consists of more than 49 SWDCK clock cycles with SWDIO asserted high. The transaction must be completed by sending at least one SWDCK clock cycle with SWDIO asserted low. This sequence synchronizes the programmer and the chip. Read_DAP() refers to the read of the IDCODE register in the debug port. The sequence of line reset and IDCODE read should be repeated until an OK ACK is received for the IDCODE read or a timeout (2 ms) occurs. The SWD port is said to be in the acquired state if an OK ACK is received within the time window and the IDCODE read matches with that of the Cortex-M0+ DAP.

13.5.2 SWD Programming Mode Entry

After the SWD port is acquired, the host must enter the device programming mode within a specific time window. This is done by setting the TEST_MODE bit (bit 31) in the test mode control register (MODE register). The debug port should also be configured before entering the device programming mode. Timing specifications and pseudo code for entering the programming mode are detailed in the *PSoC 6 MCU Programming Specifications* document.

13.5.3 SWD Programming Routine Executions

When the device is in programming mode, the external programmer can start sending the SWD packet sequence for performing programming operations such as flash erase, flash program, checksum verification, and so on. The programming routines are explained in the Nonvolatile Memory chapter on page 164. The exact sequence of calling the programming routines is given in the *PSoC 6 MCU Programming Specifications* document.

13.6 Registers

Table 13-3. List of Registers

Register Name	Description
CM0P_DWT	Cortex M0+ Data Watchpoint and Trace (DWT) registers
CM0P_BP	Cortex M0+ BreakPoint (BP) registers
CM0P_ROM	Cortex M0+ CPU Coresight ROM table
CM0P_CTI	Cortex M0+ Cross-Trigger Interface (CTI) registers
CM0P_MTB	Cortex M0+ Micro Trace Buffer (MTB) registers
CM4_ITM	Cortex M4 Instrumentation Trace Macrocell (ITM) registers
CM4_DWT	Cortex M4 Data Watchpoint and Trace (DWT) registers
CM4_FPB	Cortex M4 Flash Patch and Breakpoint (FPB) registers
CM4_SCS	Cortex M4 System Control Space (SCS) registers
CM4_ETM	Cortex M4 Embedded Trace Macrocell (ETM) registers
CM4_CTI	Cortex M4 Cross-Trigger Interface (CTI) registers
CM4_ROM	Cortex M4 CPU Coresight ROM table
TRC_TPIU	System Trace Coresight Trace Port Interface Unit (TPIU) registers

14. Nonvolatile Memory

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

Nonvolatile memory refers to the flash and SROM memory in the PSoC 6 MCU. This chapter explains the geometry and capability of the flash memory. It also lists the SROM API functions that are used to program the flash memory.

14.1 Flash Memory

The PSoC 6 flash offers high bulk program performance and supports ultra-low-power operation. Flash is typically used to store CPU instructions and data when the device power is off. Flash may be written, but the process is much slower and more restrictive than for SRAM.

14.1.1 Features

This section lists the features of PSoC 6 flash.

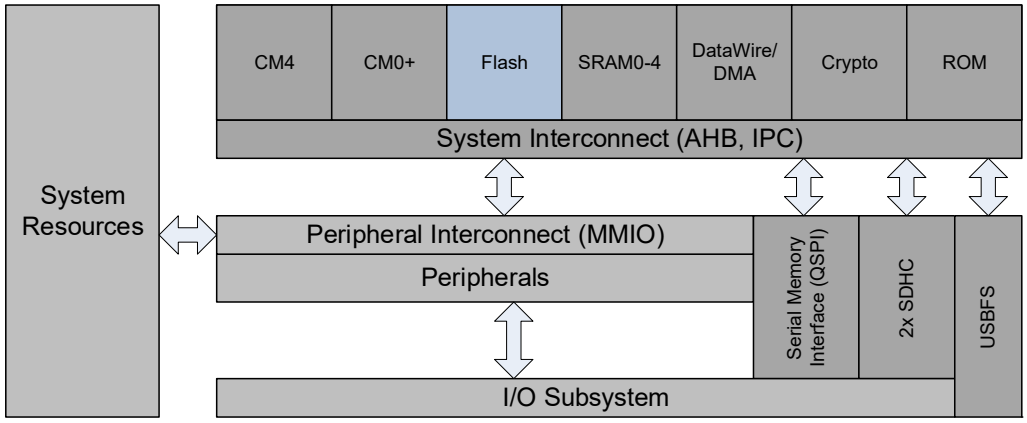
- 512-byte row size; minimum programmable unit
- Supports the Read While Write (RWW) feature with a RWW sector size of 256KB
- 10-year retention
- Endurance of 100 k program cycles

14.1.2 Configuration

14.1.2.1 Block Diagram

Flash is part of the CPU subsystem. The Cortex-M4 and Cortex M0+, as well as other bus masters, can access flash via the AHB.

Figure 14-1. Block Diagram



14.1.3 Flash Geometry

The flash is divided into three regions: the application region, and the supervisory flash region (SFlash), and the EE emulation flash region. The flash has an Erase Disturb mechanism in which writes to rows affect the endurance of other rows in the same sector. It is recommended that the EE emulation flash region is used for frequently-updated data. For data that changes infrequently or code images, the application flash region can be used.

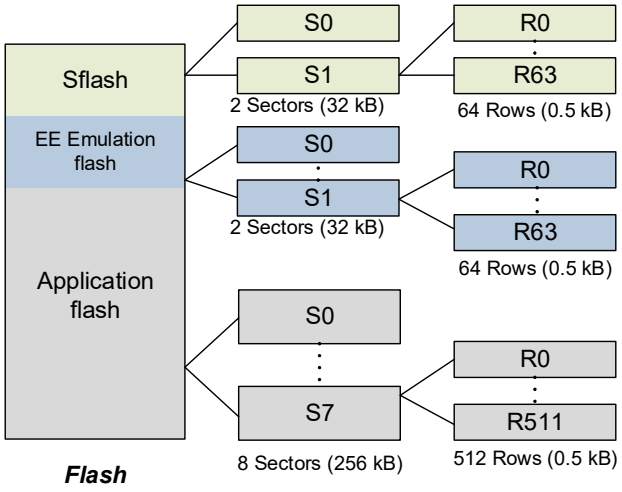
The SFlash region is used to store trim parameters, system configuration parameters, protection and security settings, boot code, and other Cypress proprietary information. Read access to this region is permitted, but program/erase access is limited. The application region is used to store code images or data.

Each region divides into sectors and pages. The sector is the largest division of the region and consists of a number of 512 byte rows.

Table 14-1. Flash Geometry

Application Flash						SFlash and EE Emulation Flash				
Read Width	KB	Sectors	KB/Sector	Rows/ Sector	KB/Row	KB	Sectors	KB/Sector	Rows/ Sector	KB/Row
128	2048	8	256	512	0.5	64	2	32	64	0.5

Figure 14-2. Flash Geometry Organization



14.1.4 Flash Controller

Access to the flash memory is enabled through the flash controller. The flash controller interfaces with the AHB-Lite bus and provides flash access for the CM0+, CM4, Crypto, DataWire, and debug. The flash controller generates a bus error if:

- A flash read access to a sector that is currently being programmed/erased
- A read access to a memory hole in the flash memory region. A memory hole is defined as a location that is not occupied by the application flash, EE Emulation flash, or SFlash. Note that the EE Emulation flash space and SFlash space can be non-powers of 2 for some flash macros

The flash controller also provides the registers which support configuration of flash accesses.

Table 14-2. Flash Controller Retention Registers

Register Name	Description
FLASHC_FLASH_CTL	Flash controller control register
FLASHC_FLASH_PWR_CTL	Flash enable control
FLASHC_FLASH_CMD	Flash commands for cache/buffer invalidation
FLASHC_CM0/CM4_STATUS	CM0/4 interface status
FLASHC_CM0/CM4_CA_CTL	CM0/4 cache control
FLASHC_*Peripheral*_BUFF_CTL	Buffer control register where *peripheral* may be Crypto, DMA, DWx (DataWire x = 0, 1), or EXT_MSx (External master; x = 0, 1)

14.1.4.1 Wait State Count

FLASHC_FLASH_CTL supports configuration of flash wait cycles. If clk_hf is greater than the maximum operating frequency of the flash memory, it is necessary to insert wait cycles when accessing the flash memory by setting the appropriate value in the FLASHC_FLASH_CTL.MAIN_WS register. Set the wait cycles as follows.

HF Clock Frequency	FLASH_CTL.MAIN_WS[3:0]
clk_hf ≤ 33 MHz	0
33 < clk_hf ≤ 66 MHz	1
66 < clk_hf ≤ 99 MHz	2
99 < clk_hf ≤ 133 MHz	3
133 < clk_hf ≤ 150 MHz	4

14.1.4.2 Power Modes

FLASHC_FLASH_PWR_CTL provides enable bits for the flash memory. Software can turn off all regions of the flash memory by setting the ENABLE and ENABLE_HV fields to 0.

The wakeup time of flash memory is 10 µs. Software should wait at least 10 µs before reading from flash after it is re-enabled through the FLASHC_FLASH_PWR_CTL register.

The flash controller provides functionality down to the Deep Sleep power mode. In the Deep Sleep power mode, the following flash controller information is retained:

- The retention MMIO registers (listed in Table 14-2)
- The cache data structure

Note that buffer information (in the AHB-Lite buffer interfaces and in the synchronization logic) is not retained. Losing buffer information after Deep Sleep transition has limited performance impact.

14.1.4.3 CPU Caches

The flash controller provides 8 kB caches for both the CM0+ and CM4 CPUs. Each cache is a four-way set associative with a least recently used (LRU) replacement scheme. Four-way set associativity means that each cache has four ways per set, with

each way containing a valid bit, tag, and data. The cache looks at all of the ways in a selected set, checking for validity and a matching tag. These caches can be enabled/disabled through the FLASHC_CM0/4_CA_CTL.CA_EN registers.

Register	Bit Field and Bit Name	Description
CM0_CA_CTL[32:0]	CA_EN[31]	Cache enable: 0: Disabled 1: Enabled
CM4_CA_CTL[32:0]	CA_EN[31]	Cache enable: 0: Disabled 1: Enabled

The cache supports faster flash memory reads when enabled. On a read transfer “miss”, however, a normal flash controller access will occur.

Cache Prefetch. The caches support pre-fetching through the CM0/4_CA_CTL.PREF_EN register.

Table 14-3. CM0/4 Cache Control Prefetch Enable Register Values and Bit Field

Register	Bit Field and Bit Name	Description
CM0_CA_CTL[32:0]	PREF_EN[30]	Prefetch enable: 0: Disabled 1: Enabled
CM4_CA_CTL[32:0]	PREF_EN[30]	Prefetch enable: 0: Disabled 1: Enabled

If prefetch is enabled, a cache miss results in a 16 B refill for the missing data and a 16 B prefetch for the next sequential data. The prefetch data is stored in a temporary buffer and is only copied to the cache when a read transfer “misses” and requires that data.

14.1.5 Read While Write (RWW) Support

The PSoC 6 MCU supports read operations on one area while programming/erasing in another area. This is implemented to support firmware upgrades and parallel tasks in the dual-core system. The application flash contains eight RWW sectors, UFLASH0 to UFLASH7, each 256KB in size. The EE emulation and SFlash are additional RWW sectors apart from the main flash.

The RWW feature is available between sectors – you can read/execute from one sector while there is an ongoing write/erase operation in another sector. However, when the code execution/read is in the last 16 bytes of a given sector (say sector 0) and the flash write/erase operation is in the next sector (sector 1), an RWW violation may occur if prefetch is enabled. This is because prefetch will fetch the next 16 bytes of data, which is part of sector 1 while a write operation is underway in the same sector. This will result in a fault and should be considered during firmware design. Firmware can be designed to place dead code in the last 16 bytes of every sector making sure the last 16 bytes of a sector are never accessed or can disable prefetch during a flash write/erase operation.

14.2 Flash Memory Programming

14.2.1 Features

- SROM API library for flash management through system calls such as Program Row, Erase Flash, and Blow eFuse
- System calls can be performed using CM0+, CM4, or DAP

14.2.2 Architecture

Flash programming operations are implemented as system calls. System calls are executed out of SROM in Protection Context 0. System calls are executed inside CM0+ NMI. The system call interface makes use of IPC to initiate an NMI to CM0+.

System calls can be performed by CM0+, CM4, or DAP. Each of them have a reserved IPC structure (used as a mailbox) through which they can request CM0+ to perform a system call. Each one acquires the specific mailbox, writes the opcode and argument to the data field of the mailbox, and notifies a dedicated IPC interrupt structure. This results in an NMI interrupt in CM0+. The following diagram illustrates the system call interface using IPC.

Figure 14-3. System Call Interface Using IPC

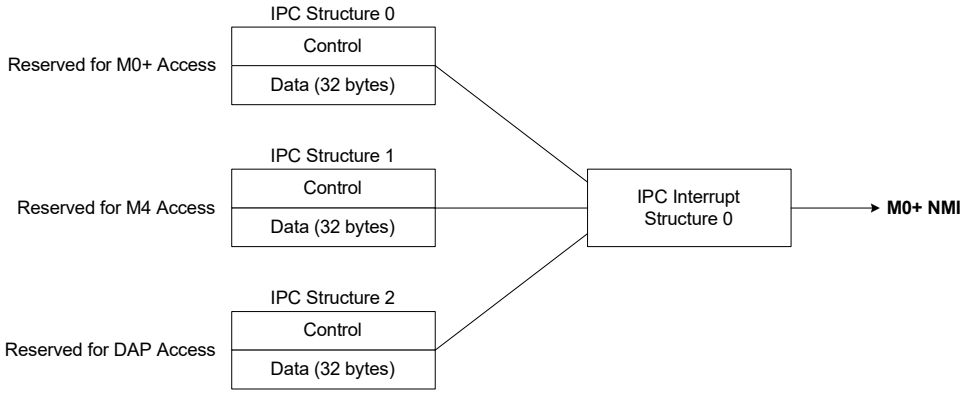


Table 14-4. IPC Structure

Master	IPC Structure Resource Used
M0+	IPC_STRUCT0
M4	IPC_STRUCT1
DAP	IPC_STRUCT2
Access Point	**IPC Structure Resource Used**
CM0+ NMI interrupt	IPC_INTR_STRUCT0

The PSoC 6 MCU's IPC component carries only a single 32-bit argument. This argument is either a pointer to SRAM or a formatted opcode or argument value that cannot be a valid SRAM address. The encoding used for DAP and the CM4 or CM0+ is slightly different.

DAP. If (opcode + argument) is less than or equal to 31 bits, store them in the data field and set the LSb of the data field as '1'. Upon completion of the call, a return value is passed in the IPC data register. For calls that need more argument data, the data field is a pointer to a structure in SRAM (aligned on a word boundary) that has the opcode and the argument. So it is a pointer if and only if the LSb is 0.

CM4 or CM0+. A pointer is always used to structure SRAM. Commands that are issued as a single word by DAP can still be issued by CM0+ or CM4, but use an SRAM structure instead.

The NMI interrupt handler for system calls works as follows.

- If the ROM boot process code is not initialized in the protection state (PROTECTION is still at its default/reset value UNKOWN), the NMI calls have no effect and the handler returns.
- A jump table is used to point to the code in ROM or flash. This jump table is located in ROM or flash (as configured in SFlash).

The IPC mechanism is used to return the result of the system call. Two factors must be considered.

- The result is to be passed in SRAM: CM0+ writes the result in SRAM and releases the IPC structure. The requester knows that the result is ready from the RELEASE interrupt.
- The result is scalar (≤ 32 bits) and there is no SRAM to pass the result: in this case, the CM0+ writes the result to the data field of the IPC structure and releases it. The requester can read the data when the IPC structure lock is released. The requester polls the IPC structure to know when it is released.

External programmers program the PSoC 6 MCU flash memory using the JTAG or SWD protocol by sending the commands to the DAP. The programming sequence for PSoC 6 MCUs with an external programmer is given in the *PSoC 6 MCU Programming Specifications*. Flash memory can also be programmed by the CM4/CM0+ CPU by accessing the IPC interface. This type of programming is typically used to update a portion of the flash memory as part of a bootload operation, or other application requirement, such as updating a lookup table stored in the flash memory. All write operations to flash memory, whether from the DAP or from the CPU, are done through the CM0+.

14.3 System Call Implementation

14.3.1 System Call via CM0+ or CM4

System calls can be made from the CM0+ or CM4 at any point during code execution. CM0+ or CM4 should acquire the IPC_STRUCT reserved for them and provide arguments

in either of the methods described above and notify IPC interrupt 0 to trigger a system call.

14.3.2 System Call via DAP

If the device is acquired, then the boot ROM enters “busy-wait loop” and waits for commands issued by the DAP. For a detailed description on acquiring the device see the *PSoC 6 MCU Programming Specifications*.

14.3.3 Exiting from a System Call

When the API operation is complete, CM0+ will release the IPC structure that initiated the system call. If an interrupt is required upon release, the corresponding mask bit should be set in IPC_INTR_STRUCT_INTR_MASK.RELEASE[i]. The exit code must also restore the CM0+ protection context (PROT_MPU_MS_CTL.PC) to the one that was backed up in PROT_MPU_MS_CTL.PC_SAVED.

14.3.4 SRAM Usage

2KB of SRAM [TOP_OF_SRAM – 2KB, TOP_OF_SRAM] is reserved for system calls and 1KB of SRAM [TOP_OF_SRAM – 3KB, TOP_OF_SRAM – 2KB] is used by SROM boot and should not be part of noinit RAM. TOP_OF_SRAM is the last address of SRAM.

14.4 SROM API Library

SROM has two categories of APIs:

- Flash management APIs – These APIs provide the ability to program, erase, and test the flash macro.
- System management APIs – These APIs provide the ability to perform system tasks such as checksum and blowing eFuse.

Table 14-5 shows a summary of the APIs.

Table 14-5. List of System Calls

System Call	Opcode	Description	API Category	Access Allowed		
				Normal[a]	Secure	Dead
Cypress ID	0x00	Returns die ID, major/minor ID, and protection state	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
Blow eFuse Bit	0x01	Blows the addressed eFuse bit	SYS	DAP	None	None
Read eFuse Byte	0x03	Reads addressed eFuse byte	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP	None
Write Row	0x05	Pre-program, erase, and program the addressed flash row	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
Program Row	0x06	Programs the addressed flash page	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
Erase All	0x0A	Erases all flash	FLS	CM0+, CM4, DAP	None	DAP
Checksum	0x0B	Reads either the whole flash or a row of flash, and returns the sum of each byte read	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
FmTransitionToLpUlp	0x0C	Configures flash macro as per desired low-power mode	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
Compute Hash	0x0D	Computes the hash value of the mentioned flash region	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
ConfigureRegionBulk	0x0E	Copies data from addressed source to addressed destination	SYS	None	None	None
DirectExecute	0x0F	Executes the code located at the provided address	SYS	DAP[b]	None	None
Erase Sector	0x14	Erases the addressed flash sector	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
Soft Reset	0x1B	Provides system reset to either or both cores	SYS	CM0+, CM4, DAP	DAP[c]	CM0+, CM4
Erase Row	0x1C	Erases the addressed flash row	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
Erase Subsector	0x1D	Erases the addressed flash sector	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
GenerateHash	0x1E	Generates the hash of the objects indicated by the table of contents	SYS	CM0+, CM4, DAP	None	None
ReadUniqueID	0x1F	Returns the unique ID of the die from SFlash	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
CheckFactoryHash	0x27	Checks if FACTORY HASH is valid	SYS	CM0+, CM4, DAP	None	None
TransitionToRMA	0x28	Convert part to RMA life cycle	SYS	None	CM0+, CM4, DAP	None
ReadFuseByteMargin	0x2B	Marginally reads eFuse	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP[d]	None
TransitionToSecure	0x2F	Blows required eFuses to transition to SECURE or SECURE_WITH_DEBUG	SYS	CM0+, CM4, DAP[e]	None	None

a. Refer to Device Security chapter on page 212.
b. Allowed only if SFLASH.DIRECT_EXECUTE_DISABLE = 0.
c. DAP has no access if secure eFuse is blown.
d. DAP has no access if secure eFuse is blown, but access is allowed in SECURE_WITH_DEBUG life cycle
e. Only allowed in the NORMAL_PROVISIONED life-cycle stage.

14.5 System Calls

Table 14-5 lists all the system calls supported in PSoC 6 MCUs along with the function description and availability in device protection modes. See the Device Security chapter on page 212 for more information on the device protection settings. Note that some system calls cannot be called by the CM4, CM0+, or DAP as given in the table. The following sections provide detailed information on each system call.

14.5.1 Cypress ID

This function returns a 12-bit family ID, 16-bit silicon ID, 8-bit revision ID, and the current device protection mode. These values are returned to the IPC_STRUCT_DATA register if invoked with IPC_STRUCT_DATA[0] set to '1'. Parameters are passed through the IPC_STRUCT_DATA register.

Note that only 32 bits are available to store the return value in the IPC structure. Therefore, the API takes a parameter ID type based on which it will return family ID and revision ID if the ID type is set to '0', and silicon ID and protection state if the ID type is set to '1'.

Table 14-6. Cypress ID

Cypress IDs	Memory Location	Data
Family ID [7:0]	0xF0000FE0	Part Number [7:0]
Family ID [11:8]	0xF0000FE4	Part Number [3:0]
Major Revision	0xF0000FE8	Revision [7:4]
Minor Revision	0xF0000FEC	Rev and Minor Revision Field [7:4]
Silicon ID	SFlash	Silicon ID [15:0]
Protection state	MMIO	Protection [3:0]

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x00	Silicon ID opcode.
Bits [15:8]	0 - returns 0. Read family ID and revision ID from SFlash 1 - returns 16-bit silicon ID and protection state	ID type.
Bits [0]	0x1	Indicates that all the arguments are passed in DATA.

Parameters if CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x00	Silicon ID opcode.
Bits[15:8]	0 - returns 12-bit family ID and revision ID 1 - returns 16-bit silicon ID and protection state	ID type.
Bits[0]	0xXX	Not used (don't care).

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [7:0]	If ID type = 0, Family ID Lo If ID type = 1, Silicon ID Lo	See the PSoC 61 datasheet/PSoC 62 datasheet for silicon ID values for different part numbers.
Bits [15:8]	If ID type = 0, Family ID Hi If ID type = 1, Silicon ID Hi	
Bits [19:16]	If ID type = 0, Minor Revision ID If ID type = 1, Protection state 0: UNKNOWN 1: VIRGIN 2: NORMAL 3: SECURE 4: DEAD	See the PSoC 61 datasheet/PSoC 62 datasheet for these values.
Bits [23:20]	If ID Type = 0 Major Revision ID If ID Type = 1 Life-cycle stage 0: VIRGIN 1: NORMAL 2: SEC_W_DBG 3: SECURE 4: RMA	
Bits [27:24]	0xXX	Not used (don't care).
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

Return if CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [7:0]	If ID type = 0, Family ID Lo If ID type = 1, Silicon ID Lo	See the PSoC 61 datasheet/PSoC 62 datasheet for silicon ID values for different part numbers.
Bits [15:8]	If ID type = 0, Family ID Hi If ID type = 1, Silicon ID Hi	
Bits [19:16]	If ID type = 0, Minor Revision ID If ID type = 1, Protection state 0: UNKNOWN 1: VIRGIN 2: NORMAL 3: SECURE 4: DEAD	
Bits [23:20]	If ID Type = 0 Major Revision ID If ID Type = 1 Life-cycle stage 0: VIRGIN 1: NORMAL 2: SEC_W_DBG 3: SECURE 4: RMA	
Bits [27:24]	0xXX	Not used (don't care).
Bits [31:28]	0xA	Success status code.

14.5.2 Blow eFuse Bit

This function blows the addressed eFuse bit. The read value of a blown eFuse bit is '1' and that of an unblown eFuse bit is '0'. These values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

The Blow eFuse Bit function should be called with default boot clock configuration settings. This can be achieved by setting SFLASH.TOC2_FLAGS.CLOCK_CONFIG as 0x3. For more information on the boot code configuration, see the Boot Code chapter on page 193.

Other valid clock settings depend on the use of the FLL:

- If you are not using the FLL, set CLK_HF[0] = 8 MHz.
- If you are using the FLL and CLK_ROOT_SELECT.ROOT_DIV = DIV_BY_2, valid CLK_HF[0] frequencies are 25 MHz and 50 MHz.

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x01	Blow efuse bit opcode.
Bits [23:16]	Byte Address	Refer to the Device Security chapter on page 212 for more details.
Bits [15:12]	Macro Address	
Bits [10:8]	Bit Address	
Bits [0]	0x1	Indicates that all the arguments are passed in DATA.

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.3 Read eFuse Byte

This function returns the eFuse contents of the addressed byte. The read value of a blown eFuse bit is '1' and that of an unblown eFuse bit is '0'. These values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x03	Read eFuse bit opcode.
Bits [23:8]	eFuse Address	Refer to the eFuse Memory chapter on page 210 for more details.
Bits [0]	0x1	Indicates all arguments are passed in DATA.

Parameters if CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x03	Read eFuse bit opcode.
Bits [23:8]	eFuse Address	Refer to the eFuse Memory chapter on page 210 for more details.
Bits [7:0]	0xXX	Not used (don't care).

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [23:0]	eFuse byte	Byte read from eFuse if status is success; otherwise, error code.
Bits [27:24]	0xXX	Not used (don't care).
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

Return if CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [23:0]	eFuse byte	Byte read from eFuse if status is success; otherwise, error code.
Bits [27:24]	0xXX	Not used (don't care).
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.4 Write Row

This function is used to program the flash. You must provide data to be loaded and the flash address to be programmed. The WriteRow parameter performs pre-program and erase, and then programs the flash row with contents from the row latch.

The PSoC 6 MCU supports the Read While Write (RWW) feature, which allows flash to be read from a RWW sector that is not programmed/erased during a program/erase of another RWW sector. Each row is of 512 bytes in size.

The API is implemented in three phases to make it non-blocking. The first phase sets up the flash for pre-program and erase operations and returns to the user code by exiting from NMI without releasing the IPC structure that invoked the API. Now, user code and interrupts can be handled but no NMI can be invoked.

Upon completion of the erase, an interrupt is generated by the flash macro, which will invoke the second phase of the WriteRow API to complete the ongoing erase operation successfully and start the program operation. API returns from NMI to user code after it sets up for program operation.

Upon completion of the program, an interrupt is generated by the flash macro, which will invoke the third phase of the WriteRow API to complete the ongoing program operation successfully; this completes the WriteRow API. SROM API will now return the pass or fail status and releases the IPC structure.

This API can also be called in blocking mode by setting the blocking parameter as '1', in which case the API will return only after all flash operation completes.

The API returns a fail status if you do not have write access to flash according to SMPU settings. See the CPU Subsystem (CPUSS) chapter on page 32 for more details. After flash program operation is complete, the API will optionally compare the flash row with the contents in row latch for data integrity check. The function returns 0xF0000022 if the data integrity check fails.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to Power Supply on page 220 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Note that the device should not be reset or transitioned into Hibernate or Deep Sleep power modes until the flash write is complete.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x05	Write Row opcode.
Bits[23:16]	0xXX	Not used
Bits[15:8]	Blocking: 0x01 – API blocks CM0+ Other values - Non-blocking	
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/ programmed 1 - Read operation is stalled until the erase/program operation is complete	
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	

Address	Value to be Written	Description
SRAM_SCRATCH Register + 0x04		
Bits [23:16]	Verify row: 0 - Data integrity check is not performed 1 - Data integrity check is performed	
Bits [31:24], Bits [15:0]	0xXX	Not used (don't care).
SRAM_SCRATCH Register + 0x08		
Bits [31:0]		Flash address to be programmed. This should be provided in 32-bit system address format. For example, to program the second half-word, provide either of the byte address 0x1000003 or 0x1000004.
SRAM_SCRATCH Register + 0x0C		
Bits [31:0]	Data word 0 (data provided should be proportional to the data size provided, data to be programmed into LSbs)	
SRAM_SCRATCH Register + 0x0C + n*0x04		
Bits [31:0]	Data word n (data to be programmed)	

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS/Program command ongoing in background 0xF = ERROR	Status code (see System Call Status for details).

14.5.5 Program Row

This function programs the addressed flash row. You must provide the data to be loaded and flash address to be programmed. The flash row should be in the erased state before calling this function.

The function is implemented in two phases to make it non-blocking. The first phase sets up the flash for program operation and returns to user code by exiting from NMI without releasing the IPC structure that invoked the API. Now user code and interrupts can be handled but no NMI can be invoked.

Upon completion of the program operation, an interrupt is generated by the flash macro, which will invoke the second phase of the ProgramRow API to complete the ongoing program operation successfully. The SROM API will return the pass or fail status and releases the IPC structure.

After flash program operation is complete, the API will optionally compare the flash row with the contents in the row latch for data integrity check. It returns STATUS_PL_ROW_COMP_FA if data integrity check fails. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to Power Supply on page 220 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Note that the device should not be reset or transitioned into Hibernate or Deep Sleep power modes until the flash write is complete.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits[31:24]	0x06	Program Row opcode.
Bits[23:16]	Skip blank check: 0x01 – Skips the blank check step Other – Perform blank check	
Bits[15:8]	Blocking: 0x01 – API blocks CM0+ Other values - Non-blocking	
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/ programmed 1 - Read operation is stalled until the erase/program operation is complete	
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	
SRAM_SCRATCH Register + 0x04		
Bits [23:16]	Verify row: 0 - Data integrity check is not performed 1 - Data integrity check is performed	
Bits [15:8]	Data location: 0 - row latch 1 - SRAM	
Bits[7:0]	0xXX	Not used (don't care).
SRAM_SCRATCH Register + 0x08		
Bits [31:0]		Flash address to be programmed. This should be provided in 32-bit system address format. For example, to program the second half-word, provide either of the byte address 0x1000003 or 0x1000004.
SRAM_SCRATCH Register + 0x0C		
Bits [31:0]	Data word 0 (data provided should be proportional to data size provided, data to be programmed into LSbs)	
SRAM_SCRATCH Register + 0x0C + n*0x04		
Bits [31:0]	Data word n (data to be programmed)	

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS/Program command ongoing in background 0xF = ERROR	Status code (see System Call Status for details).

14.5.6 Erase All

This function erases the entire flash macro specified. This API will erase only the main flash array. The API will check whether all data is '0' to confirm whether the erase is successful. It will return CHECKSUM_NON_ZERO error status if a non-zero word is encountered in the available flash.

The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operation.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to Power Supply on page 220 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x0A	Erase All opcode.
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/ programmed 1 - Read operation is stalled until the erase/program operation is complete	
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	
Bits [0]	1	Indicates that all the arguments are passed in DATA.

Parameters if CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0A	Erase All opcode.
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [23:20]	0x00	
Bits [27:24]	0xXX	Not used (don't care).
Bits [31:28]	0xA	Success status code

Return if CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [23:20]	0x00	
Bits [27:24]	0xXX	Not used (don't care).
Bits [31:28]	0xA	Success status code

14.5.7 Checksum

This function reads either the entire flash or a row of flash, and returns the sum of each byte read. Bytes 1 and 2 of the parameters select whether the checksum is performed on the entire flash or on a row of flash. This function will inherit the identity of the master that called the function. Hence if a non-secure master requests for either the whole or page checksum of a secured flash, then the fault exception will be raised by the hardware.

The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x0B	Checksum opcode.
Bits [23:22]	0 - application 1 - EE emulation other - supervisory	Flash region.
Bits [21]	0 - row 1 - whole flash	Whole flash.
Bits [20:8]		Row ID.
Bits [0]	1	Indicates that all the arguments are passed in DATA.

Parameters if CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0B	Checksum opcode.
Bits [23:22]	0 - application 1 - EE emulation 2 - supervisory	Flash region.
Bits [21]	0 - row 1 - whole flash	Whole flash.
Bits [20:8]		Row ID.
Bits [0]	0	

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [23:0]	Checksum	Checksum if status is SUCCESS
Bits [27:24]	0xXX	Not used (don't care)
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

Return if CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Checksum	Checksum if status is SUCCESS
Bits [27:24]	0xXX	Not used (don't care)
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.8 FmTransitionToLpUlp

This system call configures the required flash macro from trim bits when transitioning from LP to ULP mode or vice-versa. This function takes approximately 20 µs; reads from flash are stalled during this time.

Parameters if DAP/CM0+/CM4 is Master and SRAM is used

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0C	Opcode for FmTransitionToLpUlp.
Bits [23:2]	0xXX	Not used (don't care)
Bit [1]	0 - Transition to LP 1 - Transition to ULP	Transition mode
Bits [0]	0xXX	Not used (don't care)

Parameters if DAP/CM0+/CM4 is Master and SRAM is not used

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x0C	Opcode for FmTransitionToLpUlp.
Bits [23:2]	0xXX	Not used (don't care)
Bit [1]	0 - Transition to LP 1 - Transition to ULP	Transition mode
Bit [0]	0 - Parameters are in address pointed to by IPC_DATA 1 - Parameters are in IPC_DATA	Parameter location

Return if DAP/CM0+/CM4 Invoked the System Call and SRAM is used

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code
Bits [23:0]	Error code (if any)	Error code

Return if DAP/CM0+/CM4 Invoked the System Call and SRAM is not used

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code
Bits [23:0]	Error code (if any)	Error code

14.5.9 Compute Hash

This function generates the hash of the flash region provided using the formula:

H(n+1) = {H(n)*2+Byte}% 127; where H(0) = 0

This function returns an invalid address status if called on an out-of-bound flash region. Note that CM0+ will inherit the protection context of the master, which invoked it before performing hash. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0D	Compute hash opcode.
Bits[23:16]	0xXX	Not used (don't care).
Bits[15:8]	0x01 - CRC8 SAE others - Basic hash	Type
Bits[7:0]	0xXX	Not used (don't care)
SRAM_SCRATCH Register + 0x04		
Bits [31:0]		Start address (32-bit system address of the first byte of the data).
SRAM_SCRATCH Register + 0x08		
Bits [31:0]	0 – 1 byte 1 – 2 bytes, and so on	Number of bytes.

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Hash of the data	Hash of data if status is SUCCESS.
Bits [27:24]	0xXX	Not used (don't care).
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.10 ConfigureRegionBulk

This API writes a 32-bit data value to a set of contiguous addresses. It cannot be used to configure protected registers or flash. The Start and End addresses of the region are configurable but must be within a writable area. The region must also be 32-bit aligned. The data will be written to the region starting at the start address up to and including the memory at the end address. The start address must be lower than the end address or the API will return an error status.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0E	Configure region bulk opcode.
Bits [23:0]	0xXX	Not used (don't care).
SRAM_SCRATCH Register + 0x04		
Bits [31:0]	Start address	32-bit system address of the first byte of the data
SRAM_SCRATCH Register + 0x08		
Bits [31:0]	End address	32-bit system address of the first byte of the data
SRAM_SCRATCH Register + 0x0C		
Bits [31:0]	Data	Data to be written to each 32-bit address from start to end

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Error Code (if any)	Error Code
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.11 DirectExecute

This function directly executes code located at a configurable address. This function is only available in normal life-cycle state if the DIRECT_EXECUTE_DISABLE bit is 0.

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x0F	Direct Execute Opcode
Bits [23:2]	Address [21:0]	Address of function to execute
Bit [1]	0 - SRAM 1 - Flash	Location of function to execute
Bit [0]	1	Indicates that all arguments are passed through DATA

Return if DAP Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Error Code (if any)	Error Code
Bits [31:28]	0xF = ERROR	Does not return any status on success.

14.5.12 Erase Sector

This function erases the specified sector. Each sector consists of eight rows. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to Power Supply on page 220 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Note that the device should not be reset or transitioned into Hibernate or Deep Sleep power modes until the erase is complete.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x14	Erase Sector opcode.
Bits[23:16]	0x01 - Set FM interrupt mask Other - Do not set FM interrupt mask	Interrupt mask
Bits[15:8]	Blocking: 0x01 – API blocks CM0+ Other values – Non-blocking	

Address	Value to be Written	Description
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/ programmed 1 - Read operation is stalled until the erase/program operation is complete	
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	
SRAM_SCRATCH Register + 0x04		
Bits [31:0]		Flash address to be erased. Should be provided in 32-bit system address format. For example, to erase the second sector you need to provide the 32-bit system address of any of the bytes in the second sector.

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.13 Soft Reset

This function resets the system by setting CM0+ AIRCR system reset bit. This will result in a system-wide reset, except debug logic. This API can also be used for selective reset of only the CM4 core based on the 'Type' parameter. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x1B	Soft Reset opcode.
Bits [7:1]	0 - System reset 1 - Only CM4 resets	Type of reset
Bits[0]	0x1	Indicates all arguments are passed in DATA.

Parameters if CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x1B	Soft Reset opcode.
Bits [7:1]	0 - System reset 1 - Only CM4 resets	Type of reset
Bits [0]	0xXX	Not used (don't care).

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

Return if CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.14 Erase Row

This function erases the specified row. You must provide the address of the row that needs to be erased. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to Power Supply on page 220 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x1C	Erase row opcode.
Bits[23:16]	0xXX	
Bits[15:8]	0x01 - API blocks CM0+ Other - Non-blocking	
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/programmed 1 - Read operation is stalled until the erase/program operation is complete	
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	
SRAM_SCRATCH Register + 0x04		
Bits [31:0]	Address	Flash address to be erased. This should be provided in the 32-bit system address format. For example, to erase the second row you need to provide the 32-bit system address of any of the bytes in the second row.

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.15 Erase Subsector

This function erases the specified subsector. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to Power Supply on page 220 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x1D	Erase subsector opcode.
Bits [23:16]	0xXX	Not used (don't care).
Bits [15:8]	0x01- API blocks CM0+ Other - Non-blocking	
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/ programmed 1 - Read operation is stalled until the erase/program operation is complete	
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	
SRAM_SCRATCH Register + 0x04		
Bits [31:0]	Address	Flash address to be erased. This should be provided in 32-bit system address format. For example, to erase the second subsector you need to provide the 32-bit system address of any of the bytes in the second subsector.

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.16 GenerateHash

This function returns the truncated SHA-256 of the Flash boot programmed in SFlash. This function gets the Flash Boot size from the TOC. This function is typically called to confirm that the hash value to be blown into eFuse matches what the ROM Boot expects.

This function returns the number of zeros of the SECURE_HASH

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits[31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits[31:24]	0x1E	Opcode for GenerateHash
Bits[15:8]	0x1 - Return the factory hash Other - Return hash of all objects per TOC1 and TOC2	

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits[23:0]	Error code (if any)	See System Call Status for details
Bits[31:28]	0xA = SUCCESS 0xF = ERROR	Status code (See System Call Status for details)
SRAM_SCRATCH Register + 0x04		
Bits[31:0]	HASH_WORD0	
SRAM_SCRATCH Register + 0x08		
Bits[31:0]	HASH_WORD1	
SRAM_SCRATCH Register + 0x0C		
Bits[31:0]	HASH_WORD2	
SRAM_SCRATCH Register + 0x10		
Bits[31:0]	HASH_WORD3	
SRAM_SCRATCH Register + 0x14		
Bits[31:0]	HASH_ZEROS	

14.5.17 ReadUniqueID

This function returns the unique ID of the die from SFlash.

Parameters if DAP/ CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits[31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits[31:24]	0x1F	Opcode for ReadUniqueID
Bits[15:0]	Not Used (Don't care)	

Return if DAP/ CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits[23:0]	DIE_LOT if SUCCESS	See System Call Status for details
Bits[31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details)
SRAM_SCRATCH Register + 0x04		
Bits[31:0]	DIE_ID0	
SRAM_SCRATCH Register + 0x08		
Bits[31:0]	DIE_ID1	

14.5.18 CheckFactoryHash

This function generates the FACTORY_HASH according to the TOC1 and compares the value with the FACTORY1_HASH eFuse value.

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x27	CheckFactoryHash Opcode
Bit [0]	1	Indicates all arguments are passed in DATA.

Parameters if CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x27	CheckFactoryHash Opcode

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

Return if CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.19 TransitionToRMA

This function converts the part from SECURE, SECURE_WITH_DEBUG, or NORMAL to the RMA life-cycle stage. This API performs eFuse programming, VDD should be set to 2.5 V for successful programming.

This function uses Flash Boot functions. The stack consumed by these functions is around 700 bytes. This function returns STATUS_EMB_ACTIVE failure code if any active embedded flash operations are going on.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:0]	0x28	Transition to RMA opcode
SRAM_SCRATCH + 0x04		
Bits [31:0]	Object Size in Bytes (including itself)	
SRAM_SCRATCH + 0x08		
Bits [31:0]	Command ID (0x120028F0)	
SRAM_SCRATCH + 0x0C		
Bits [31:0]	Unique ID word 0	
SRAM_SCRATCH + 0x10		
Bits [31:0]	Unique ID word 1	
SRAM_SCRATCH + 0x14		
Bits [31:0]	Unique ID word 2	
SRAM_SCRATCH + 0x18		
Bits [31:0]	Signature of RMA certificate	

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

Return if CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.5.20 ReadFuseByteMargin

This API returns the eFuse contents of the addressed byte read marginally. The read value of a blown bit is '1' and of a not blown bit is '0'.

Parameters if DAP is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x2B	Opcode for ReadFuseByteMargin
Bits [23:20]	0 - Low resistance, –50% from nominal 1 - Nominal resistance (default read condition) 2 - High resistance (+50% from nominal) Other - 100% from nominal	Margin control
Bits [19:8]	eFuse address (0, 1… 511)	
Bit [0]	1 - Indicates all arguments are passed from DATA	

Parameters if CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x2B	Opcode for ReadFuseByteMargin
Bits [23:20]	0 - Low resistance, –50% from nominal 1 - Nominal resistance (default read condition) 2 - High resistance (+50% from nominal) Other - 100% from nominal	Margin control
Bits [19:8]	eFuse address (0, 1… 511)	
Bit [0]	1 - Indicates all arguments are passed from DATA	

Return if DAP Invoked the System Call

Address	Return Value	Description
IPC_STRUCT_DATA Register		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).
Bits [23:0]	Byte read from eFuse	See System Call Status for details.

Return if CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).
Bits [23:0]	Byte read from eFuse	See System Call Status for details.

14.5.21 TransitionToSecure

This API validates the FACTORY_HASH value and programs the SECURE_HASH, secure access restrictions, and dead access restrictions into eFuse. This function programs the SECURE or SECURE_WITH_DEBUG eFuse to transition to the SECURE or SECURE_WITH_DEBUG life-cycle stage. If the FACTORY_HASH or the TOC1 magic number is invalid, then the API fails and returns status code 0xF00000b5. Note that if the Factory_HASH in eFuse is invalid, then the TransitionToSecure API fails and the error code is written to the IPC[2].DATA register.

Parameters if DAP/CM0+/CM4 is Master

Address	Value to be Written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x2F	Transition to Secure opcode
Bits [15:8]	1 - Blow Debug Fuse Other - Blow Secure Fuse	Select SECURE_WITH_DEBUG or SECURE life-cycle Stage
SRAM_SCRATCH + 0x04		
Bits [31:0]	SECURE_ACCESS_RESTRICT	
SRAM_SCRATCH + 0x08		
Bits [31:0]	DEAD_ACCESS_RESTRICT	

Return if DAP/CM0+/CM4 Invoked the System Call

Address	Return Value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See System Call Status for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see System Call Status for details).

14.6 System Call Status

At the end of every system call, a status code is written over the arguments in the IPC_DATA register or the SRAM address pointed by IPC_DATA. A success status is 0xAXXXXXXX, where X indicates don’t care values or return data for system calls that return a value. A failure status is indicated by 0xF00000XX, where XX is the failure code.

Note that for system calls that require access to restricted memory regions, if the client does not have access to the restricted region, the system call may return 0xF000_0013 rather than the expected 0xF000_0008. Because of this, it is important to ensure that arguments are passed correctly and that the memory region to access is available for read or write access.

Table 14-7. System Call Status

Status Code	Description
0xAXXXXXXX	Success – The X denotes a don't care value, which has a value of '0' returned by the SROM.
0xF0000001	Invalid protection state – This API is not available in current protection state.
0xF0000002	Invalid eFuse address.
0xF0000003	Invalid flash page latch address.
0xF0000004	Wrong or out-of-bound flash address.
0xF0000005	Row is write protected.
0xF0000006	Client did not use its reserved IPC structure for invoking the system call.
0xF0000008	Client does not have access to the addressed memory location.
0xF0000009	Command in progress.
0xF000000A	Checksum of flash resulted in non-zero.
0xF000000B	The opcode is not valid.
0xF0000013	An argument to the system call is not in a valid location.
0xF0000021	Sector erase requested on SFlash region.
0xF0000041	Bulk erase failed.
0xF0000042	Sector erase failed.
0xF0000043	Subsector erase failed.
0xF0000044	Verification of Bulk, Sector, or Subsector fails after program/erase.
0xF000000F	Returned when invalid arguments are passed to the API. For example, calling Silicon ID API with ID type of 0x5.
0xF000000b5	Invalid FACTORY_HASH

15. Boot Code

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

System boot is defined as the process of validating and starting the product firmware. PSoC 6-2M has 64KB of embedded SROM, which stores the firmware to begin the boot process. This firmware is called ROM boot. The second stage of the boot process takes place in SFlash and is called Flash boot. The main function of the boot process is to configure the system (apply trims, configure access, and set protection settings according to the product life-cycle stage), authenticate the application, and transfer control to the application.

15.1 Features

The PSoC 6 boot code supports the following features:

- After reset, the boot code starts execution from ROM on the CM0+.
- The boot process consists of two parts – ROM boot process and Flash boot process.
- The ROM boot code applies life-cycle stage and protection state.
- The ROM boot code validates the integrity of the Flash boot process before starting it.
- The Flash boot code validates the integrity of the user application before starting it when the device in the SECURE life-cycle stage. User application validation may optionally be enabled in the NORMAL life-cycle stage.

15.2 ROM Boot

The boot code starts execution from ROM. The ROM boot code applies trims and configurations, and validates the integrity of the flash boot process before starting it.

15.2.1 Data Integrity Checks

ROM boot uses a Table of Contents (TOC) data structure to locate various objects that are stored in both application flash and SFlash. The location of some of these objects is fixed in the factory while other object locations are fixed at the original equipment manufacturer (OEM). The TOC is split into two parts – TOC1 and TOC2 – that correspond to objects fixed in the factory and objects fixed at OEM, respectively. TOC1 and TOC2 are stored in SFlash.

The data integrity of selected objects listed at the beginning of TOC1 must be verified at the OEM before transitioning the part to the SECURE life-cycle stage. The FACTORY_HASH is generated using SHA-256, which is the concatenation of the objects to be verified in TOC1. The FACTORY_HASH is stored in eFuse before the device leaves Cypress. The device validates the FACTORY_HASH as part of the TransitionToSecure system call.

The data integrity of selected objects in both TOC1 and TOC2 must be verified as part of authentication of Flash boot by ROM boot in the SECURE life-cycle stage. A 128-bit truncated SHA-256 value known as the SECURE_HASH is used to check the

integrity of these objects. If the TOC structures fail integrity checks, then the RTOC structures are checked for validity. If the RTOC structures are valid, then the boot process uses the RTOC structures in place of the TOC structures. The SECURE_HASH is programmed automatically during the TransitionToSecure system call. If the TransitionToSecure system call fails, it will exit before programming the SECURE_HASH in eFuse. See System Call Status on page 192 for more information.

A summary of the objects used in the data integrity checks is shown in Figure 15-1. The TOC structure formats are outlined in Table 15-1 to Table 15-3.

Figure 15-1. Objects, Memory Regions, and Stages of Boot Process for Data Integrity Checks

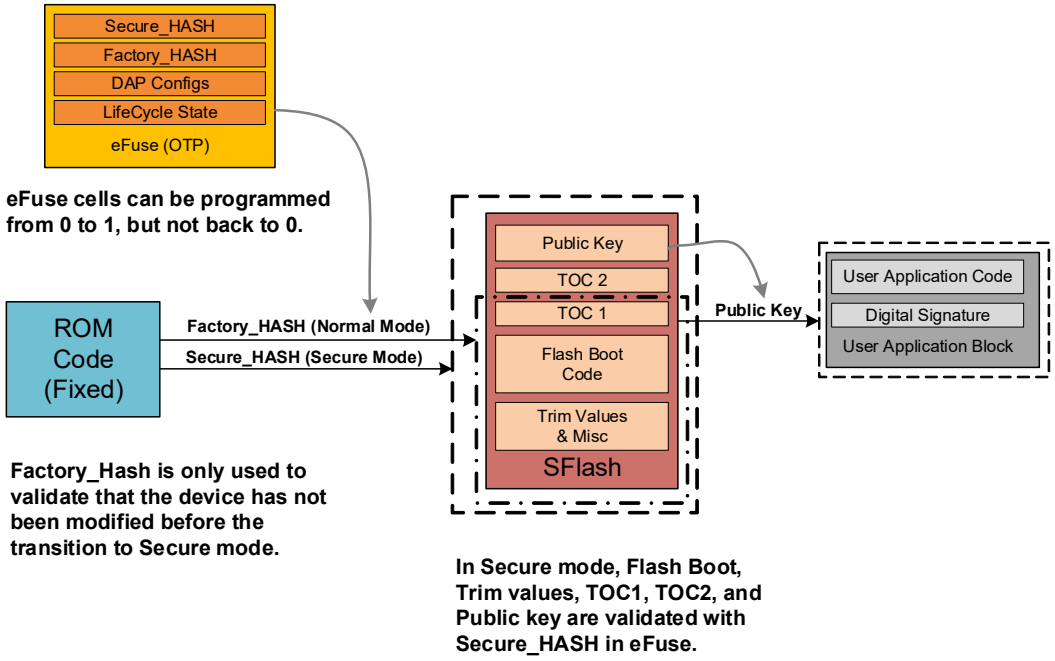


Table 15-1. TOC1 Format

Offset from Beginning of Row	Purpose	Comments
0x00	Object size in bytes for CRC calculation starting from offset 0x00	TOC1 filled in VIRGIN protection state by Cypress
0x04	Magic number (0x01211219)	
0x08	Number of objects starting from offset 0xC to be verified for FACTORY_HASH	
0x0C	Unused	
0x10	Address of Unique ID (fixed size of 12 bytes) stored in SFlash	
0x14	Address of Flash boot object that includes Flash Patch object stored in SFlash	
0x18	Unused	
0x1C-0x1F8	Unused	
0x1FC	CRC16-CCITT (the upper two bytes contain the CRC little endian value and the lower two bytes are 0)	

Table 15-2. TOC2 Format

Offset from Beginning of Row	Absolute Address	SFlash Register	Purpose
0x00	0x16007C00	TOC2_OBJECT_SIZE	Object size in bytes for CRC calculation starting from offset 0x00
0x04	0x16007C04	TOC2_MAGIC_NUMBER	Magic Number (0x01211220)
0x08	0x16007C08	TOC2_KEY_BLOCK_ADDR	Address of Key Storage Flash Blocks. Note that this also marks the top of flash available.
0x0C	0x16007C0C	TOC2_SMIF_CFG_-STRUCT_ADDR	Null terminated table of pointers representing the SMIF configuration structure
0x10	0x16007C10	TOC2_FIRST_USER_APP_ADDR	Address of First User Application Object
0x14	0x16007C14	TOC2_FIRST_USER_APP_FOR-MAT	First Application Object Format (4 bytes). 0 means Basic Application Format, 1 means Cypress Secure Application Format, and 2 means Simplified Secure Application Format
0x18	0x16007C18	TOC2_SECOND_USER_AP-P_ADDR	Address of Second User Application Object (0's if none)
0x1C	0x16007C1C	TOC2_SECOND_USER_APP_-FORMAT	Second Application Object Format (4 bytes).
0x20	0x16007C20	TOC2_SHASH_OBJECT	Number of additional objects (In addition to the object included in the FACTORY_HASH) starting from offset 0x24 to be verified for SECURE_HASH. The maximum number of additional objects is 15.
0x24	0x16007C24	TOC2_SIGNATURE_VERIF_KEY	Address of signature verification key (0 if none). The object is signature scheme specific. It is the public key in case of RSA.
0x1F8	0x16007DF8	TOC2_FLAGS	Flash boot parameters. These fields are listed in Table 15-3.
0x1FC	0x16007DFC	TOC2_CRC_ADDR	CRC16-CCITT (the upper 2 bytes contain the CRC little endian value and the lower 2 bytes are 0)

Table 15-3. TOC2_FLAGS Bits, Default Settings, and Descriptions

TOC2_FLAGS Bits	Name	Default	Description
1:0	CLOCK_CONFIG	2	Indicates the CM0+ CPU clock frequency configuration. The clock will remain at this setting after flash boot execution. 0 = 8 MHz IMO with no FLL 1 = 25 MHz, IMO with FLL 2 = 50 MHz, IMO with FLL (Default) 3 = Use ROM boot clocks configuration (100 MHz)
4:2	LISTEN_WINDOW	0	Determines the wait window to allow sufficient time to acquire the debug port. 0 = 20 ms (Default) 1 = 10 ms 2 = 1 ms 3 = 0 ms (No listen window) 4 = 100 ms
6:5	SWJ_PINS_CTL	2	Determines if the SWJ pins are configured in SWJ mode by flash boot. SWJ pins can be enabled later in the application code. 0 = Do not enable SWJ pins in flash boot. Listen window is skipped. 1 = Do not enable SWJ pins in flash boot. Listen window is skipped. 2 = Enable SWJ pins in flash boot (default) 3 = Do not enable SWJ pins in flash boot. Listen window is skipped.
8:7	APP_AUTH_DISABLE	0	Determines if the application digital signature verification is performed. 0 = Authentication is enabled (default). 1 = Authentication is disabled. 2 = Authentication is enabled (recommended). 3 = Authentication is enabled.
10:9	FB_BOOTLOADER_DISABLE	0	Reserved.

The wait window time is configurable; however, the CRC value used to validate the TOC2 must be updated after changing the value. New entries that need HASH can grow from the top starting at offset 0x28; entries that do not need HASH can grow from the bottom starting at offset 0x1F8. When the device is in the SECURE life-cycle state, the TOC2 can no longer be modified.

15.2.2 Life-cycle Stages and Protection States

SECURE and SECURE_WITH_DEBUG life-cycle stages are governed by eFuse. All life-cycle stages are irreversible once set. Table 15-4 shows the eFuse locations that store information, access restriction settings, and life-cycle stage settings.

Table 15-4. Object Location in eFuse

Offset	# of Bytes	Name	Description
0x00	20	Reserved	Reserved for PSoC 6 MCU system usage
0x14	16	SECURE_HASH	Secure objects 128-bit hash
0x24	2	Reserved	Reserved
0x26	1	SECURE_HASH_ZEROS	Number of zeros in SECURE_HASH
0x27	2	DEAD_ACCESS_RESTRICT	Access restrictions in Dead life-cycle stage

Table 15-4. Object Location in eFuse

Offset	# of Bytes	Name	Description
0x29	2	SECURE_ACCESS_RESTRICT	Access restrictions in Secure life-cycle stage
0x2B	1	LIFECYCLE_STAGE	Normal, Secure, and Secure with Debug fuse bits
0x2C	16	FACTORY_HASH	FACTORY_HASH value
0x40	64	CUSTOMER_DATA	Customer data

Normal mode access restrictions to be applied on DAP are stored in SFlash.

Table 15-5. Normal Life-cycle Stage DAP Restriction Register in SFlash

Offset	# of Bytes	Name	Description
0x1A00	2	NORMAL_ACCESS_RESTRICTIONS	Access restrictions applied to DAP in NORMAL life-cycle state.

The LIFECYCLE_STAGE object in eFuse is used to set the life-cycle state of the device. Table 15-6 shows the bits and the associated settings of the LIFECYCLE_STAGE location.

Table 15-6. eFuse Bits and Life-Cycle States

Bits	Field Name	Description
0	NORMAL	A '1' indicates that all factory trimming and testing is complete. The device is in NORMAL mode.
1	SECURE_WITH_DEBUG	A '1' indicates that the device is in SECURE_WITH_DEBUG mode. Before the device is transitioned to this stage, the SECURE_HASH must have been programmed in eFuse and valid application code must have been programmed in the main flash.
2	SECURE	A '1' indicates that the device is in SECURE mode. The SECURE_HASH must have been programmed in eFuse and valid application code has been programmed in the main flash.
3	RMA	This life-cycle stage allows Failure Analysis (FA). The part is transitioned to the RMA life-cycle stage when the customer wants Cypress to perform failure analysis. The customer erases all sensitive data before invoking the system call that transitions the part to RMA.

DAP settings for Secure and Dead protection states are set in the SECURE_ACCESS_RESTRICT0 and DEAD_ACCESS_RESTRICT0 registers. DAP settings for the Normal protection state is set in NORMAL_ACCESS_RESTRICTIONS in SFlash. Table 15-7 shows the format used to set the DAP settings for these states. For more information about the protection states, see the Device Security chapter on page 212.

Table 15-7. DAP Access Restriction Registers for Normal, Secure, and Dead Protection States (all default to 0)

Bits	Name	Description
0	CM0_AP_DISABLE	A '1' indicates that this device does not allow access to the M0+ debug access port.
1	CM4_AP_DISABLE	A '1' indicates that this device does not allow access to the M4 debug access port.

Table 15-7. DAP Access Restriction Registers for Normal, Secure, and Dead Protection States (all default to 0)

Bits	Name	Description
2	SYS_AP_DISABLE	A '1' indicates that this device does not allow access to the system debug access port.
3	SYS_AP_MPU_ENABLE	A '1' indicates that the MPU on the system debug access port must be programmed and locked according to the settings in the next four fields. The SYS_DISABLE bit must be left at '0' for this setting to matter. If the SYS_DISABLE bit is set to '1', then the next four fields are invalid. This affects only the SYS_AP. It does not affect the CM0/4 AP.
5:4	SFLASH_ALLOWED	This field indicates what portion of Supervisory Flash is accessible through the system debug access port. Only a portion of Supervisory Flash starting at the bottom of the area is exposed. Encoding is as follows: 0: entire region 1: one-half 2: one-quarter 3: nothing For example, for an encoding of "2: one-quarter", the valid code region starts at address 0x16000000 and will go up to one-quarter of the SFlash memory region.
7:6	MMIO_ALLOWED	This field indicates what portion of the MMIO region is accessible through the system debug access port. Encoding is as follows: 0: All MMIO registers 1: Only IPC MMIO registers accessible (system calls) 2, 3: No MMIO access

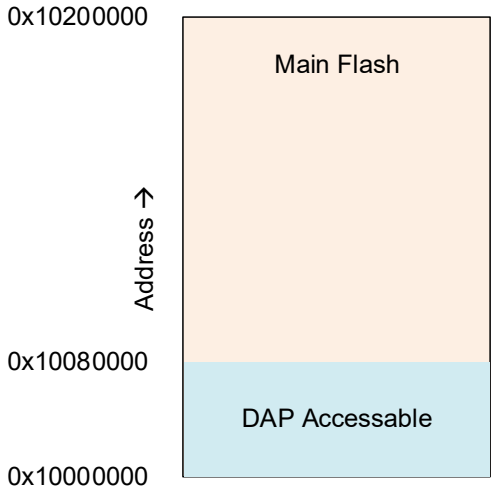
The SECURE_ACCESS_RESTRICT1, DEAD_ACCESS_RESTRICT1, and NORMAL_ACCESS_RESTRICTIONS [15:8] (SFlash) registers control debug access restriction to specific memory regions. Table 15-8 shows the format used to set these restrictions.

Table 15-8. SECURE_ACCESS_RESTRICT1, DEAD_ACCESS_RESTRICT1, and NORMAL_ACCESS_RESTRICTIONS [15:8] Registers (All default to 0)

Bits	Name	Description
2:0	FLASH_ALLOWED	This field indicates what portion of the main flash is accessible through the system debug access port. Only the portion starting at the bottom of flash (0x1000_0000) is exposed to the system DAP. The encoding is as follows: 0: entire region 1: seven-eighth 2: three-fourth 3: one-half 4: one-quarter 5: one-eighth 6: one-sixteenth 7: nothing See Figure 15-2 for an example of the encoding for FLASH_ALLOWED.
5:3	SRAM_ALLOWED	This field indicates what portion of SRAM 0 is accessible through the system debug access port. Only the portion starting at the bottom of the area is exposed. Encoding is the same as FLASH_ALLOWED.
6	UNUSED	UNUSED
7	DIRECT_EXECUTE_DISABLE	Disables DirectExecute system call functionality (implemented in software). See the Nonvolatile Memory chapter on page 164 for more information.

As an example, if you want to limit the lower one-quarter of flash of a device with 2MB of flash, a "4" is written into the FLASH_ALLOWED field. Only the flash area between 0x1000_0000 and 0x1007_FFFF will be accessible via the SYS_DAP. The remaining areal between 0x1008_0000 to 0x101F_FFFF is not accessible. This means that only the lower quarter of flash can be reprogrammed, erased, or read by the debug port.

Figure 15-2. FLASH_ALLOWED with encoding “4”



As shown in Figure 15-2, if you want to limit the lower fourth of flash of a device with 2MB of flash, write a ‘4’ into the FLASH_ALLOWED field. Only the flash area between 0x10000000 and 0x1007FFFF will be accessible via the SYS_DAP. The remaining area between 0x10080000 to 0x101FFFFF would not be accessible. This means that only the lower fourth of flash can be reprogrammed, erased, or read by the debug port.

15.2.3 Secure Boot in ROM Boot

At the end of the ROM boot process, the SECURE eFuse bit is checked. If the life-cycle is SECURE, then ROM boot reads the SECURE access restrictions in eFuse and deploys the settings using the SYS_AP MPU. If the APP_AUTH_DISABLE field of the TOC2 is not disabled, then ROM boot calculates the SECURE_HASH value from the contents of Flash boot, the TOC1 and TOC2 structures, and the public key and compares it to the stored hash value (SECURE_HASH) in eFuse. If they match, the Flash boot code and data are authentic. If the Flash boot code is authentic, then ROM boot will hand over the boot process to Flash boot, which resides in SFlash. If the Flash boot is not authentic, then the boot code sets the device into the DEAD protection state.

15.2.4 Protection Setting

ROM boot sets some of the protections units (SMPU, PPU, and MPU) during the boot process. It is recommended that these settings are not modified.

SYS_AP DAP access restrictions are deployed by MPU15 based on access restrictions encoding stored in SFlash or eFuse. Eight MPU structures implement the SYS_AP DAP access restrictions. If the SYS_AP MPU is disabled, the following regions are available for access through the system debug access port.

Table 15-9. Protection Setting

Protection Structure	Attribute Settings	Protected Region	Region Base	Region Size (Bytes)
PROT_MPU15_MPU_STRUCT0	ADDR: 0x00000000 ATT: 0x9E000000	ALL	0x00000000	2G
PROT_MPU15_MPU_STRUCT1	ADDR: 0x402200F8 ATT: 0x8700007F	IPC	0x40220000	96

Table 15-9. Protection Setting

Protection Structure	Attribute Settings	Protected Region	Region Base	Region Size (Bytes)
PROT_MPU15_MPU_STRUCT2	–	Work flash region protection option is not present in PSoC 6	–	–
PROT_MPU15_MPU_STRUCT3	ADDR: 0x16000000 ATT: 0x8E00007F	SFlash	0x16000000	32K (SFlash size)
PROT_MPU15_MPU_STRUCT4	ADDR: 0x10000000 ATT: 0x9400007F	Flash	0x10000000	2M (Flash Size)
PROT_MPU15_MPU_STRUCT6	ADDR: 0x8000000 ATT: 0x9200007F	RAM0	0x08000000	512K (SRAM0 size)

Other protection units are set up during ROM boot to restrict access to various regions to ensure device security. The following tables list those structures, the regions they protect, and the protection settings applied to those regions. The protection settings are applied in all life-cycle stages except VIRGIN.

Table 15-10. Region Access

Region	Description
CPUSS_BOOT	Contains CM0+ protection context 0-3, debug port control, protection status, RAM, and ROM trim control registers.
EFUSE_CTL	EFUSE_CTL register. Specifies retention for eFuse registers.
EFUSE_DATA	Contains registers that support life-cycle setting, key storage, and other one-time-programmable data.
FM_CTL	Specifies flash hardware block control settings including direct memory cell access addressing, write enable, and interface selection.
Bottom 2KB of SRAM	Contains the system call stack.
SROM	Contains system firmware that supports ROM boot and system calls.
CRYPTO MMIO	Contains registers that support error correction, random number generation, device key locations, and other general cryptography information.
IPC MMIO	Contains registers that support inter-processor communication

Table 15-11. Protection Structures

Protection Structure	Protection (Read/Write)	Attribute Settings	Protected Region	Region Base	Region Size (Bytes)
PERI_MS_PPU_FX18	Write only in PC0 Read Always	ATT0: 0x1515151F ATT1: 0x15151515	CPUSS_BOOT	0x40202000	512
PERI_MS_PPU_FX59	Write only in PC0 Read Always	ATT0: 0x1515151F ATT1: 0x15151515	MPU15_MAIN DAP MPU	0x40237C00	1K
PERI_MS_PPU_FX70	R/W only in PC0	ATT0: 0x0000001F	FM_CTL	0x4024F000	4K

Table 15-11. Protection Structures

Protection Structure	Protection (Read/Write)	Attribute Settings	Protected Region	Region Base	Region Size (Bytes)
PROT_SMPU_SMPU_STRUCT15	Read only in PC0	ATT0: 0x8AFFFE00 ATT1: 0x80FFFE40	Last 2KB of SRAM	0x080FF800	2K
PROT_SMPU_SMPU_STRUCT14	Read only in PC0	ATT0: 0x8E00FF00 ATT1: 0x8700FF49	SROM area containing boot code, Crypto, and Flash macro controls.	0x00000000	24K
PERI_MS_PPU_PR0	Disable access during system calls with Crypto enabled (all access allowed otherwise)	ATT0: 1F1F1F1F ATT1: 1F1F1F1F	Entire Crypto MMIO region during any crypto operation	0x40100000	64K
PERI_MS_PPU_PR1	Disable access during system calls (all access allowed otherwise)	ATT0: 1F1F1F1F ATT1: 1F1F1F1F	IPC MMIO (during system calls)	0x40220000	32

15.2.5 SWD/JTAG Repurposing

When the user-configured access restrictions disable access to the debug access ports, the boot process does not access or change the SWD/JTAG pins. User firmware can, at any time, change the configuration of the SWD/JTAG pins to another mode, peripheral, configuration, or purpose. To allow debugging of such applications, a configurable ‘listen window’ is provided in the TOC2 at offset 0x1F8 bits [4:2]. The boot process will connect and enable the JTAG/SWD interface and wait for a specified time before starting application firmware. It is expected that application firmware checks the CPUSS DP_STATUS.SWJ_CONNECTED bit and repurposes the pins only when no SWD/JTAG connection is available.

15.2.6 Waking up from Hibernate

Waking up from Hibernate mode will result in system boot. The integrity checks on the SFlash trim values and SWD/JTAG connection delay (listen window in Flash boot) are skipped when waking from hibernate.

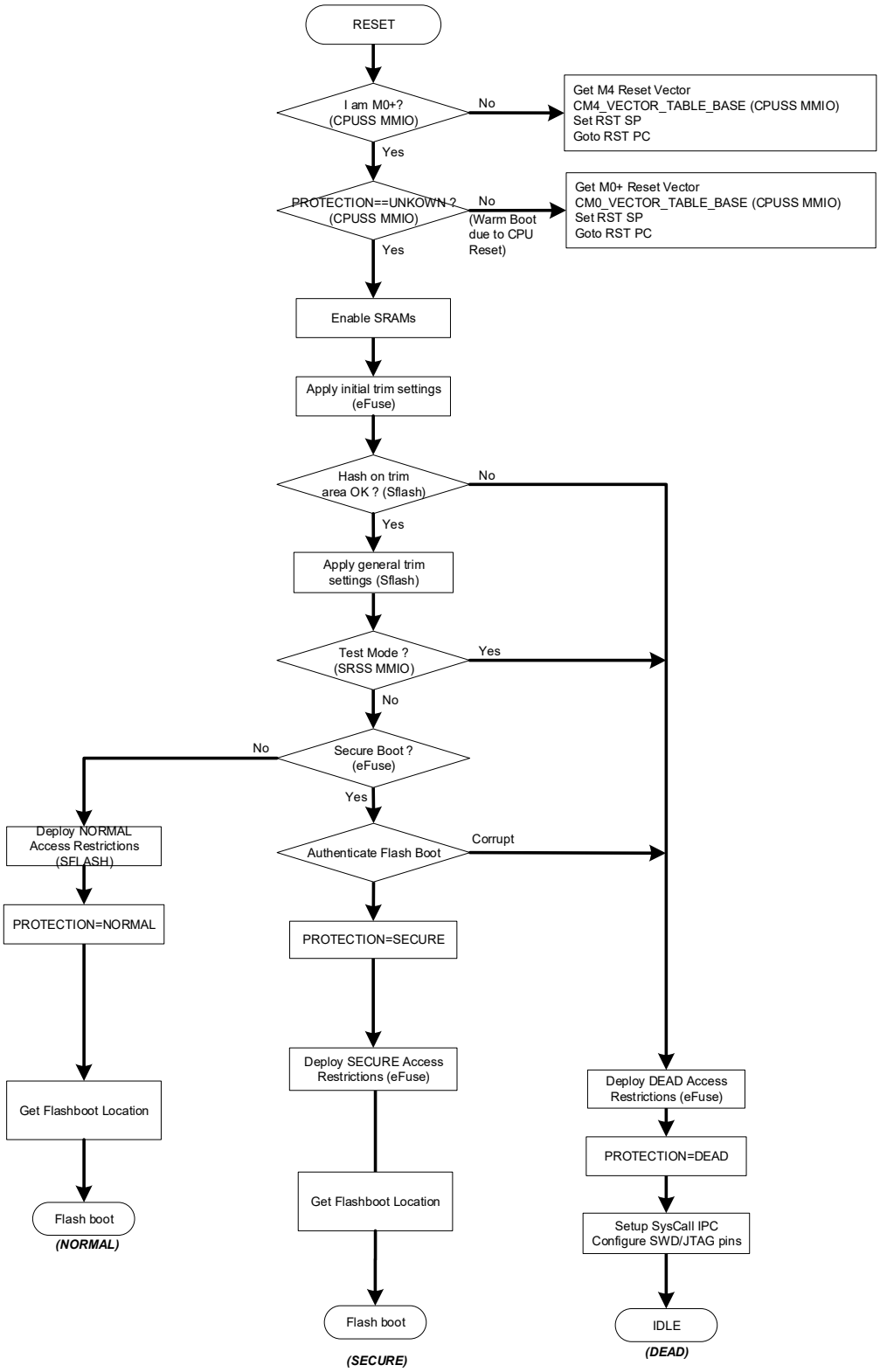
15.2.7 Disable Watchdog Timer

The ROM boot code will disable the watchdog timer (WDT) if the eFuse.WDT_DISABLE bit is set.

15.2.8 ROM Boot Flow Chart

Figure 15-3 shows the ROM boot flow chart.

Figure 15-3. ROM Boot Flow Chart



15.3 Flash Boot

15.3.1 Overview

The second stage of boot, Flash boot, is entered after ROM boot has authenticated the Flash boot image if the device is in the SECURE or SECURE_WITH_DEBUG life-cycle stage, or if the APP_AUTH_DISABLE field is not set in TOC2. In the SECURE or SECURE_WITH_DEBUG life-cycle states, flash boot prevents an application that does not pass integrity checks from running. The integrity checks can optionally be enabled in the NORMAL mode by setting the APP_AUTH_DISABLE field in TOC2 to any 2-bit value other than 1. This ensures that the application originates from a valid source and has not been altered.

The Flash boot firmware:

- Runs on the security processor (Cortex-M0+)
- Validates the contents of the TOC2
- Validates the user application in the Secure life-cycle state or optionally in the Normal life-cycle state
- Sets up the debug access port (DAP)
- Enables system calls
- Hands control to the user application

15.3.2 Features of Flash Boot

- RSASSA PKCS1 V1.5 with SHA-256 User Application Signature Verification
- 2048-bit RSA public key with pre-calculated key coefficients for RSA calculation speed increase
- Support for all device life-cycle stages
- Configurable debug listen window

15.3.3 Using Flash Boot

Flash boot can be used to allow only authenticated firmware to run on a given platform.

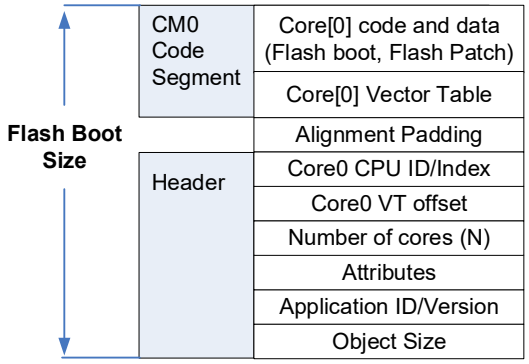
Flash boot determines the life-cycle stage, validates the TOC2, and optionally validates the firmware before transferring execution to the user's CM0+ application code. The user's CM0+ application enables the CM4 core.

The user firmware will not execute if it is modified by third-party malicious software. The user firmware in flash is validated using the RSA algorithm. The 2048-bit public key in SFlash and the digital signature stored with the user application are located using the table of contents, which is located at the end of SFlash. The public key is hashed by SECURE_HASH and is validated during boot time. If the code is defined as insecure, the device will go to dead mode with limited debug capabilities determined by the eFuse settings.

15.3.4 Flash Boot Layout

Flash boot consists of a Header field and a Code Segment field. For hash calculation, the object size must be a multiple of 16B so zero padding is applied in the Code Segment.

Figure 15-4. Flash Boot Layout



15.3.4.1 Header

The Flash boot header consists of:

- The Flash boot object size
- The Flash boot application version
- The number of cores (set to '1' for Flash boot)
- CM0+ vector table offset
- Cypress ID and CPU Core Index

15.3.4.2 Code Segment

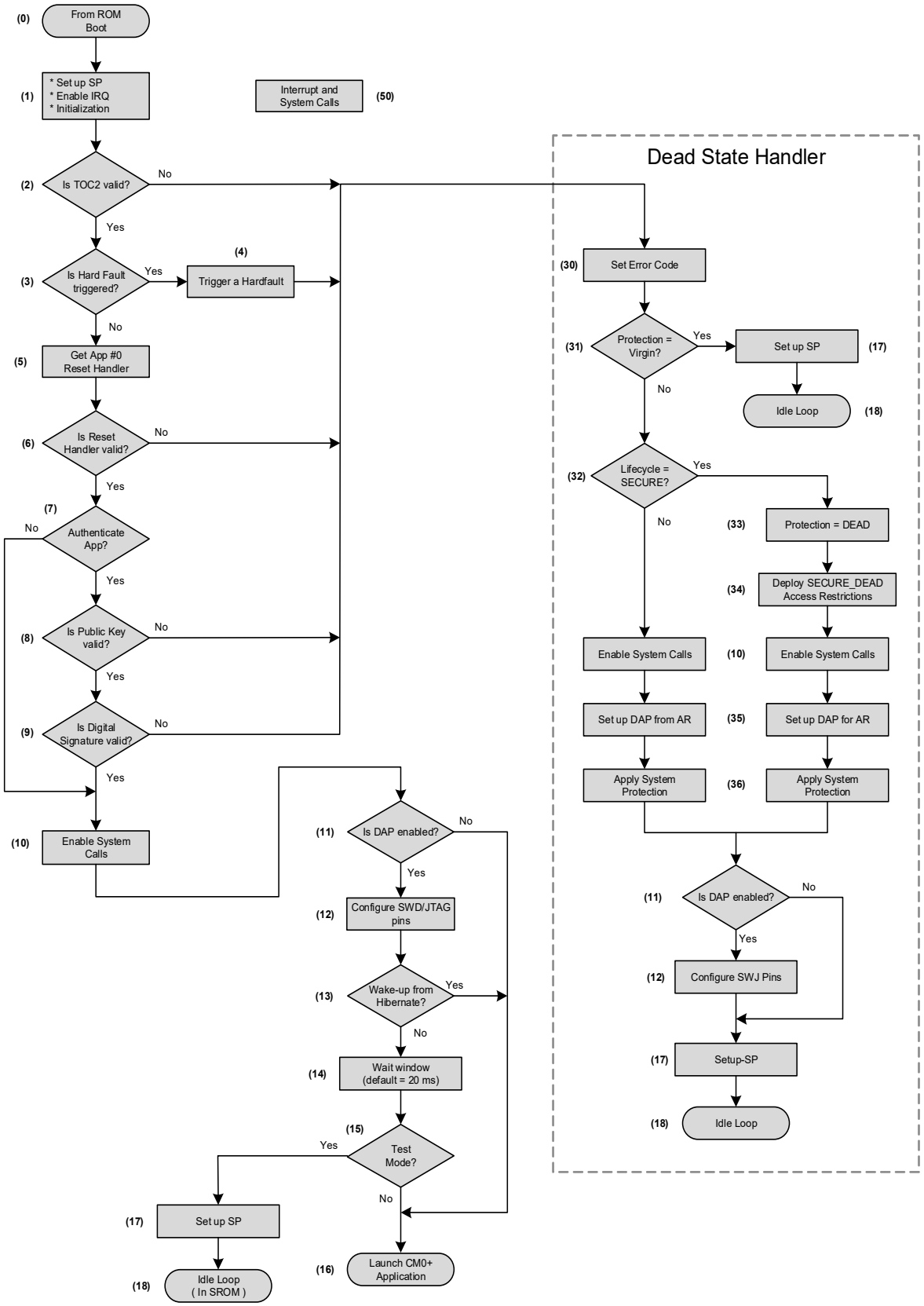
The CM0 code segment consists of:

- CM0+ vector table
- CM0+ Flash boot code and data
- Zero padding to make the Flash boot object a multiple of 16B

15.3.5 Flash Boot Flow Chart

The Flash boot program flow is shown in Figure 15-5. The entry point is a fixed offset in SFlash. Each section of the flow chart is labeled with an index number. In the sections following the flow chart, explanations of each step are provided with the associated index number.

Figure 15-5. Flash Boot Flow Chart



15.3.5.1 Entry from ROM Boot (0)

ROM boot transfers control to Flash boot after it validates the SFlash block and TOC1 in the user flash.

15.3.5.2 Basic Initialization (1)

This stage sets the value of the stack pointer during runtime. To support recovery from Hard-Fault exceptions during Flash boot, this stage also enables interrupts.

15.3.5.3 Is TOC2 Valid? (2)

The TOC2 may be in three states:

- VALID: The TOC2 structure and CRC are valid. This is the default state of the TOC2.
- CORRUPTED: Either the TOC2 structure or CRC value are incorrect.
- ERASED: The first two 32-bit words at the start of TOC2 are equal to the SFlash erase value. The erased value is 0x0000_0000.

If the PROTECTION is SECURE then ERASED state is a part of CORRUPTED state.

If the PROTECTION is not SECURE then ERASED state is a special case of VALID state. In this case, Flash boot treats all the TOC2 entries as having a default value:

- SFLASH_TOC2_FIRST_USER_APP_ADDR is 0x1000_0000 (the start of Flash).
- SFLASH_TOC2_FIRST_USER_APP_FORMAT is 0 (Basic Application Format).
- SFLASH_TOC2_FLAGS = 0x0000_0000

The other TOC2 entries will not be used when TOC2 is in ERASED or corrupted states.

15.3.5.4 Is Hard Fault Triggered? (3)

The Flash boot test system requires a way to trigger a hard fault in the Flash boot code. The condition to trigger a hard-fault is as follows:

1. SFLASH_FLAGS bit FB_HARDFAULT is set.
2. A 32-bit word in TOC2 at offset + 500 contains an address to a 32-bit word named HardFaultTrigger.
3. HardFaultTrigger value is 0x0000_0001.

15.3.5.5 Trigger a Hard Fault (4)

Flash boot can be used to trigger a hard fault for a testing purposes.

This can be done, for example, by a 32-bit word read or write to an unaligned address, or to an invalid memory region.

The hard fault handler recovers the MCU into the DEAD state branch. In this branch, an error code is set and the DAP is enabled.

15.3.5.6 Get App #0 Reset Handler (5)

Flash boot reads the application start address from the TOC2 entry:

- SFLASH_TOC2_FIRST_USER_APP_ADDR for App#0.

The application format is stored in the TOC2 entry:

- SFLASH_TOC2_FIRST_USER_APP_FORMAT for App #0.

The address of reset handler inside the application depends on the application format.

15.3.5.7 Is Reset Handler Valid? (6)

Flash boot checks if the address of the reset handler for the user application is in RAM, SFlash, application flash, or EE Emulation flash.

15.3.5.8 Authenticate App? (7)

Flash boot optionally authenticates a digital signature for the application image if the TOC2_FLAGS bit APP_AUTH_DISABLE = 1.

If this field is set then the authentication step is skipped.

15.3.5.9 Is Public Key Valid (8)

The public key structure is filled by the user. Thus, it must be validated to ensure the correctness of the its entries before being used.

15.3.5.10 Is Digital Signature Valid? (9)

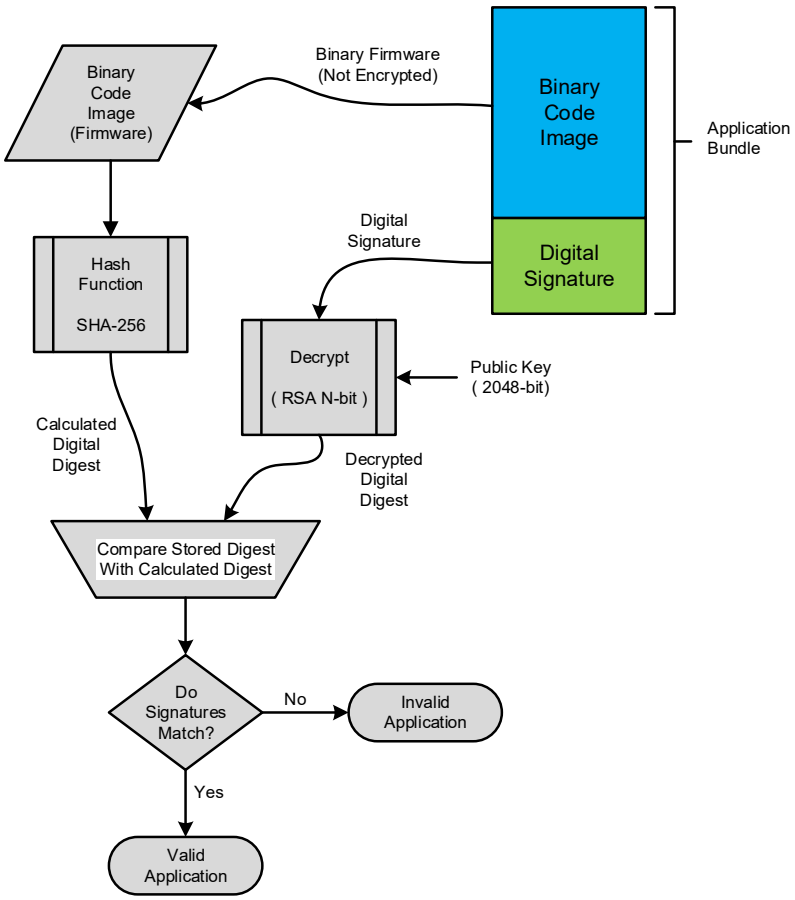
The user application located in main flash is validated using the standard RSA 2048-bit algorithm (RSASSA-PKCS1-v1_5-2048). The public key used for this operation is stored in the user public key area of SFlash.

The application can be in one of three formats, only two of which are validated using a public key.

The Crypto block is used for validation and is only enabled during the application validation process to conserve power.

The following figure shows the application validation flow.

Figure 15-6. Application Validation



15.3.5.11 Enable System Calls (10)

At this point the system calls are enabled. A table of the system calls is found in SROM API Library on page 170. The function calls are access via IPC that communicate to the CM0+ with interrupts. The SROM function EnableSystemCall() is called to enable these system calls.

15.3.5.12 Is DAP Enabled (11)

Determines if the Debug Access Port (DAP) is enabled by reading the CPUSS_AP_CTL register, that was set during ROM BOOT. The DAP consist of three sections: CM4 AP, CM0+ AP, and System AP. If any of these APs are enabled, then the DAP is considered to be enabled.

15.3.5.13 Configure SWD/JTAG Pins (12)

The SWD and JTAG GPIOs are configured to work with the DAP. The CPUSS_AP_CTL register will be used to set DAP.

15.3.5.14 Wake from Hibernate? (13)

If the reason for the reset was wake from hibernate, skip the wait window and test mode check.

15.3.5.15 Wait Window (14)

The CPU delays execution to allow the debug hardware to acquire the CM0+. The default delay time is 20 ms, but the user can set other delay options. For a SECURE mode device, it is recommended to set the delay to 0 as debug will not be enabled.

This delay allows the debug hardware to acquire the debug interface and set the operation to test mode.

15.3.5.16 Test Mode Enable? (15)

After the listen window delay, the firmware checks if the SRSS_TST_MODE register has either TEST_MODE or TEST_KEY_DFT_EN bit set. If either bit is set, execution is transferred to an endless loop in SROM. This is done by calling the BusyWaitLoop() ROM boot function.

15.3.5.17 Launch CM0+ Application (16)

If the TOC2 is erased and the life-cycle stage is NORMAL, Flash boot assumes that an application is in the “Basic Application Format” and the application start address is the start of the user flash.

If the TOC2 is valid and the life-cycle stage is NORMAL, then the application can be in either the Basic or Cypress Standard Secure application formats.

Otherwise, the application is stored in the Cypress Secure application format.

The procedure to launch a user application is:

1. Set CPUSS_CM0_VECTOR_TABLE_BASE to the start of the user application interrupt vector table.
2. Set CPUSS_CM4_VECTOR_TABLE_BASE to 0xFFFF_0000.
3. Perform a core reset.
4. After a core reset is performed ROM boot is launched (on CM0+).
5. ROM boot checks if CPUSS_PROTECTION ! = 0, which means ROM boot is launched on CM0+ after a core reset.
6. If (5) is true, ROM boot sets SP and PC register values from the user interrupt vector table. The address of a user application interrupt vector table is stored at step (1) to CPUSS_CM0_VECTOR_TABLE_BASE.
7. When ROM boot sets PC register value with the user reset handler address, user code starts executing.

15.3.5.18 Set Up SP (17)

The SP register value for Flash boot is at the top of user RAM.

15.3.5.19 Idle Loop (18)

Before going to an idle loop the Flash boot sets the CPUSS_CM0_VECTOR_TABLE_BASE MMIO register to 0xFFFF_0000.

In this state, a NMI interrupt can wake up the device. To prevent unwanted firmware access, device access restrictions must be set up correctly for the appropriate life cycle stage. The protection settings for each life-cycle stage are listed in Life-cycle Stages and Protection States on page 196.

15.3.5.20 Set Error Code (30)

If the user application flash or the TOC2 was determined to be invalid, an error code will be written to IPC.DATA (structure 2); this will enable to detect the cause during debug.

Table 15-12. Error Code

Error Name	Value	Description
CY_FB_STATUS_SUCCESS	0xA100_0100	Success status value
CY_FB_STATUS_BUSY_WAIT_LOOP	0xA100_0101	Debugger probe acquired the device in Test Mode. The Flash boot entered a busy wait loop.
CY_FB_ERROR_INVALID_APP_SIGN	0xF100_0100	Application signature validation failed for the device families where flash boot launches only one application from TOC2. Either the application structure or a digital signature is invalid for the device families in which Flash boot may launch either of two applications in TOC2.
CY_FB_ERROR_INVALID_TOC	0xF100_0101	Empty or invalid TOC
CY_FB_ERROR_INVALID_KEY	0xF100_0102	Invalid public key
CY_FB_ERROR_UNREACHABLE	0xF100_0103	Unreachable code
CY_FB_ERROR_TOC_DATA_CLOCK	0xF100_0104	TOC contains invalid CM0+ clock attribute
CY_FB_ERROR_TOC_DATA_DELAY	0xF100_0105	TOC contains invalid listen window delay
CY_FB_ERROR_FLL_CONFIG	0xF100_0106	FLL configuration failed
CY_FB_ERROR_INVALID_APP0_DATA	0xF100_0107	Application structure is invalid for the device families in which Flash boot may launch only one application from TOC2
CY_FB_ERROR_CRYPTO	0xF100_0108	Error in Crypto operation
CY_FB_ERROR_INVALID_PARAM	0xF100_0109	Invalid parameter value
CY_FB_ERROR_BOOT_HARD_FAULT	0xF100_010a	A hard fault exception had happened in the Flash boot
CY_FB_ERROR_UNEXPECTED_INTERRUPT	0xF100_010B	Any unexpected interrupt had happened in the Flash boot
CY_FB_ERROR_BOOTLOADER	0xF100_0140	Any bootloader error
CY_FB_ERROR_BOOT_LIN_INIT	0xF100_0141	Bootloader error, LIN initialization failed
CY_FB_ERROR_BOOT_LIN_SET_CMD	0xF100_0142	Bootloader error, LinSetCmd() failed

15.3.5.21 Protection = Virgin? (31)

The CPUSS_PROTECTION MMIO register value is compared to the desired protection mode.

15.3.5.22 Life Cycle = SECURE (32)

The life-cycle stage value is stored in eFuse. To determine this value, Flash boot reads the corresponding eFuse bits. If the SECURE life-cycle bit is set in eFuse, then the device is in the SECURE life-cycle stage. Otherwise, the device is in another life-cycle stage.

Note: Life-cycle stage is not the same as protection mode. In this case, SECURE_WITH_DEBUG is not equal to SECURE life-cycle stage; however, the protection state equals to SECURE for both these stages.

15.3.5.23 Protection = DEAD (33)

In DEAD protection state, the system will not attempt to start any flash firmware and will restrict access to system resources to shield secure/sensitive information. The intention is to allow a controlled set of failure analysis capabilities only. By default, DEAD mode is open to all debug functions and users should change to the level of security required.

15.3.5.24 Deploy Access Restrictions (34)

SECURE_DEAD access restrictions are applied for entering DEAD branch from the SECURE life-cycle stage.

Assess restrictions are applied by calling the RestrictAccess() ROM boot function.

15.3.5.25 Set Up DAP from AR (35)

ROM boot function GetAccessRestrictStruct() is called to determine which APs (access points, one of CM0+ AP, CM4, and TC) are enabled. Based on this information the proper values are written to the CPUSS_AP_CTL register.

15.3.5.26 Apply System Protection (36)

System protection settings are usually applied by ROM boot code before entering Flash boot. If the device enters the DEAD branch, the system protection settings are changed. Thus, Flash boot calls the ROM boot function ApplyProtectionSettings() to reconfigure them.

16. eFuse Memory

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The eFuse memory consists of a set of eFuse bits. When an eFuse bit is programmed, or “blown”, its value can never be changed. Some of the eFuse bits are used to store various unchanging device parameters, including critical device factory trim settings, device life cycle stages (see the Device Security chapter on page 212), DAP security settings, and encryption keys. Other eFuse bits are available for custom use.

16.1 Features

The PSoC 6 MCU eFuses have the following features:

- A total of 1024 eFuse bits. 512 of them are available for custom purposes.
- The eFuse bits are programmed one at a time, in a manufacturing environment. The eFuse bits cannot be programmed in the field.
- Multiple eFuses can be read at the bit or byte level through a PDL API function call or an SROM call. An unblown eFuse reads as logic 0 and a blown eFuse reads as logic 1. There are no hardware connections from eFuse bits to elsewhere in the device.
- SROM system calls are available to program and read eFuses. See the Nonvolatile Memory chapter on page 164. For detailed information on programming eFuses, see the *PSoC 6 MCU Programming Specifications*.

16.2 Architecture

The PSoC 6 MCU eFuses can be programmed only in a manufacturing environment. V_{DDIO0} must be set to 2.5 V, and the device must be in a specific test mode, entered through an XRES key. For more information, see the *PSoC 6 MCU Programming Specifications*.

Table 16-1 shows the usage of the PSoC 6 MCU eFuse bytes.

Table 16-1. PSoC 6 MCU eFuse Byte Assignments

Offset	No. of Bytes	Name	Description
0	20	Reserved	Reserved for PSoC 6 MCU system usage
20	16	SECURE_HASH	Secure objects 128-bit hash
36	2	Reserved	Reserved for PSoC 6 MCU system usage
38	1	SECURE_HASH_ZEROS	Number of zeros in SECURE_HASH
39	2	DEAD_ACCESS_RESTRICT	Access restrictions in DEAD life cycle stage

Table 16-1. PSoC 6 MCU eFuse Byte Assignments

Offset	No. of Bytes	Name	Description
41	2	SECURE_ACCESS_RESTRICT	Access restrictions in SECURE life cycle stage
43	1	LIFECYCLE_STAGE	Life cycle state
44	17	FACTORY_HASH, FACTORY_HASH_ZEROS	Secure objects 128-bit hash plus number of zeros in the hash
61	2	Reserved	Reserved for PSoC 6 MCU system usage
63	12	Unused	Not used
64	64	CUSTOM_DATA	Custom data

17. Device Security

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU offers several features to protect user designs from unauthorized access or copying. Selecting a secure life cycle stage, enabling flash protection, and using hardware-based encryption can provide a high level of security.

17.1 Features

The PSoC 6 MCU provides the following device security features:

- Nonvolatile and irreversible life cycle stages that can limit program and debug access.
- Shared memory protection unit (SMPU) that provides programmable flash, SRAM, and register protection.
- Cryptographic function block that provides hardware-based encryption and decryption of data and code.

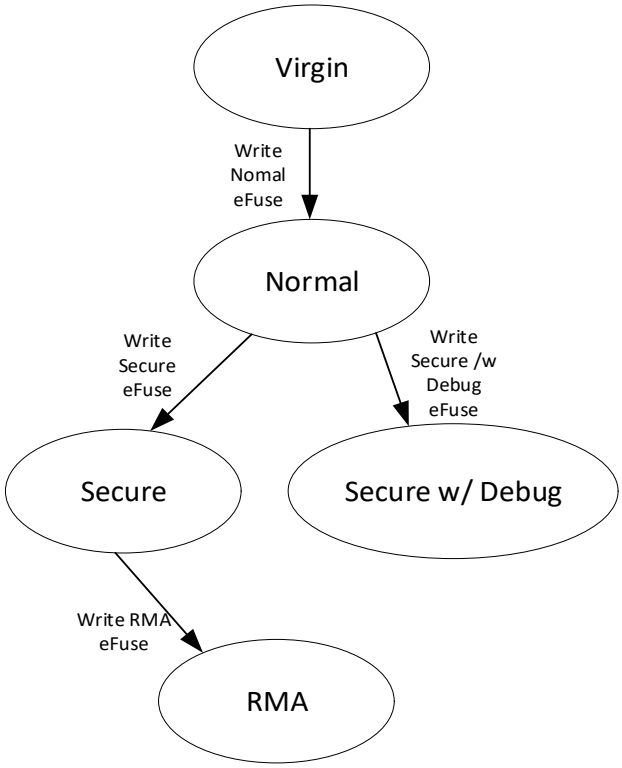
17.2 Architecture

17.2.1 Life Cycle Stages and Protection States

PSoC 6 MCUs have configurable, nonvolatile life cycle stages. Life cycle stages follow a strict, irreversible progression governed by writing to eFuse – a 1024-bit nonvolatile memory with each bit being one time programmable (OTP). The eFuse can hold unalterable keys and trim information. See the eFuse Memory chapter on page 210 for more details of the eFuse; see the Nonvolatile Memory chapter on page 164 for eFuse access system calls.

This chapter discusses hardware blocks used to implement device security. For system-level implementation of device security, see *AN221111 - PSoC 6 MCU: Creating a Secure System*.

Figure 17-1. PSoC 6 MCU Life Cycle Stage Transitions



The EFUSE_DATA_LIFECYCLE_STAGE eFuse governs the life-cycle stages.

The PSoC 6 MCU supports the following life cycle stages:

- VIRGIN – This stage is used by Cypress during assembly and testing. During this stage, trim values and flash boot are written into SFlash. Devices that are in this stage never leave the factory. In this stage, the boot ROM assumes that no other eFuse data or flash data is valid. Devices are transitioned to the Normal stage before they leave the factory.
- NORMAL – This is the life cycle stage of a device after trimming and testing is complete in the factory. All configuration and trimming information is complete. Valid flash boot code is programmed in the SFlash. To allow the OEM to check data integrity of trims, flash boot, and other objects from the factory, a hash (SHA-256 truncated to 128 bits) of these objects is stored in eFuse. This hash is referred to as Factory_HASH. In Normal mode, by default, there is full debug access to the CPUs and system. The device may be erased and programmed as needed.
- SECURE – This is the stage of a secure device. Before transitioning to this stage, ensure that the following steps are complete:
  - Secure access restrictions are programmed into the EFUSE_DATA_SECURE_ACCESS_RESTRICT0 and EFUSE_DATA_SECURE_ACCESS_RESTRICT1 registers.
  - The TOC2 should optionally be programmed with the correct address of the user key storage blocks in flash.
  - The TOC2 should be programmed with the user application format, the starting addresses of the user application, and the address of the public key in SFlash.
  - Protection units should be set up to protect sensitive code regions.
  - Public and private keys must be generated.

  In this stage, the protection state is set to Secure, and Secure DAP access restrictions are deployed. A secure device will boot only after successful authentication of its flash boot and application code.

  After an MCU is in the Secure life cycle stage, it cannot go back to the Normal stage. SFlash cannot be programmed in this stage. The debug ports may be disabled depending on user preferences, so it is not possible to reprogram or erase the device with a hardware programmer/debugger. At this point, the firmware can only be updated by invoking a bootloader, which must be provided as part of device firmware.

  Access restrictions in the SECURE state can be controlled by writing to the following eFuses: EFUSE_-

DATA_SECURE_ACCESS_RESTRICT0 and EFUSE_DATA_DEAD_ACCESS_RESTRICT1.

Code should be tested in Normal and Secure with Debug stages before advancing to this stage. This is to prevent a configuration error that can cause the device to be inaccessible for programming and therefore, become unusable.

- SECURE_WITH_DEBUG – This is similar to the Secure stage, except that Normal access restrictions are applied to enable debugging. Devices that are in the Secure with Debug stage cannot be changed back to either Secure or Normal stage.

  Secure applications may also be developed in the Normal life-cycle stage with code validation enabled. This requires the TOC2_FLAGS.APP_AUTH_DISABLE bits at offset 0x1F8 to be set. See the Boot Code chapter on page 193 for more information. This method has the advantage of being able to transition to the SECURE life-cycle stage after testing is complete.

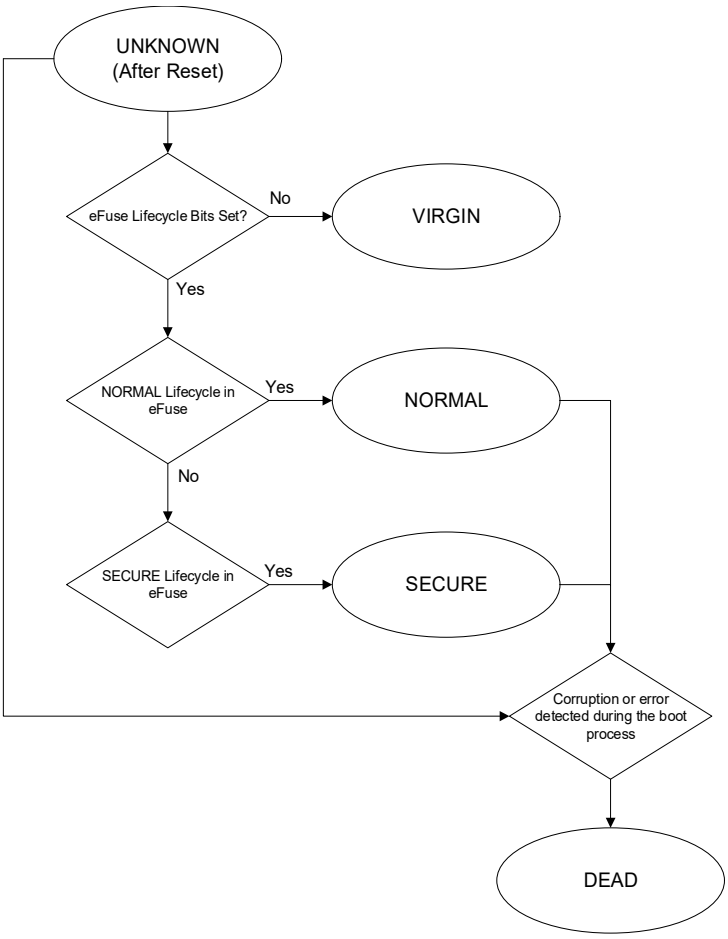
- RMA – Customers can transition the device to the RMA stage (from Secure) when they want Cypress to perform failure analysis on the device. The customer should erase all sensitive data (including firmware) before invoking the system call that transitions the device to RMA. Flash erase should be performed by calling EraseRow/EraseSubSector/EraseSector/EraseAll SROM APIs five times repeatedly on the secure FLASH region before converting the device to RMA. Erasing the device five times eliminates any possibility of determining the previous state of any flash cell. See the Nonvolatile Memory chapter on page 164 for more details on these SROM APIs.

  When invoking the system call to transition to RMA, the customer must provide a certificate that authorizes Cypress to transition the device with a specific Unique ID to the RMA life cycle stage. The certificate will be signed by the customer using the same private key that is used to sign the user application image. The verification of the signature uses the same algorithm used by flash boot to authenticate the user application. The same public key (injected by the OEM) stored in SFlash is used for the verification. Note that the signature is unique to a specific device. A signature will only work with the specific device for which it was generated because it uses the unique device ID.

  When a device is reset in the RMA life cycle stage, the boot will set access restrictions such that the DAP has access only to System AP and IPC MMIO registers for making system calls, and adequate RAM for communication. It will then wait for the “Open for RMA” system call from the DAP along with the certificate of authorization. This is the same certificate used in transitioning the device to RMA. The boot process will not initiate any firmware until it successfully executes the “Open for RMA” command. After the command is successful, the device will behave as though it were in the Virgin life cycle stage, but it cannot be transitioned to Virgin or any other stage. The life cycle stage stored in eFuse cannot be changed from RMA. Every time the device is reset, it must execute the “Open for RMA” command successfully before the device can be used

A powered PSoC 6 MCU also has a volatile protection state that reflects its life cycle stage. The protection state is determined on boot by reading the eFuse values. Figure 17-2 illustrates the protection state transitions.

Figure 17-2. Protection State Transitions



Protection state is defined by the STATE field of the CPUSS_PROTECTION register.

- STATE is 0: UNKNOWN state.
- STATE is 1: VIRGIN state.
- STATE is 2: NORMAL state.
- STATE is 3: SECURE state.
- STATE is 4: DEAD state.

The CPUSS_PROTECTION register can be written only to affect the transitions defined in Figure 17-2. Any value written to this register that does not represent a transition in Figure 17-2 is rejected in hardware. A life cycle stage change (by writing to eFuse) does not immediately affect a protection state change. After writing the NORMAL, SECURE_WITH_DEBUG, or SECURE state in the eFuse, a reboot is required for the change of life cycle stage to take effect on the protection state.

There are more protection states than there are life-cycle stages. The additional protection states are:

- UNKNOWN – A device comes out of reset in this state. The ROM boot code needs to run through its courses (reading eFuses and checking integrity) to determine the life cycle stage of the device and select the proper protection state.
- DEAD – A device will enter the DEAD protection state when a corruption/error is detected in the boot process. In DEAD protection state, the system will not attempt to start any flash firmware and may restrict access to system resources to shield secure/sensitive information. Access restrictions in DEAD mode are controlled by EFUSE_DATA_DEAD_ACCESS_RESTRICT0 and EFUSE_DATA_DEAD_ACCESS_RESTRICT1 eFuses.

17.2.2 Flash Security

PSoC 6 MCUs include a flexible flash-protection system that controls access to flash memory. This feature is designed to secure proprietary code, but it can also be used to protect against inadvertent writes to the bootloader portion of flash.

Flash memory is organized in sectors. Flash protection is provided by a shared memory protection unit (SMPU). The SMPU is intended to distinguish between different protection contexts and to distinguish secure from non-secure accesses. The system function that performs flash programming first looks at the SMPU settings and will not allow to program or erase flash blocks protected by the SMPUs.

For more details, see the Protection Units chapter on page 73.

17.2.3 Hardware-Based Encryption

The PSoC 6 MCU has a cryptographic block (Crypto) that provides hardware implementation and acceleration of cryptographic functions. It implements symmetric key encryption and decryption, hashing, message authentication, random number generation (pseudo and true), and cyclic redundancy checking. It can work with internal as well as external memory. See the Cryptographic Function Block (Crypto) chapter on page 111 and Serial Memory Interface (SMIF) chapter on page 375 for more details.

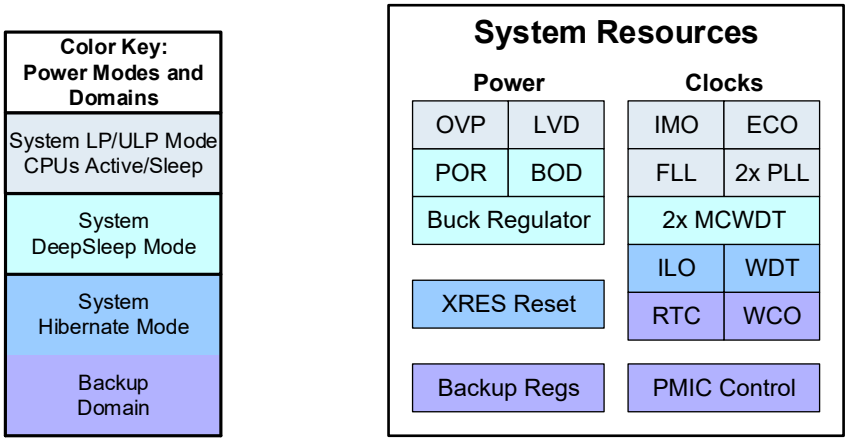
Section C: System Resources Subsystem (SRSS)

This section encompasses the following chapters:

- Power Supply and Monitoring chapter on page 218
- Device Power Modes chapter on page 225
- Backup System chapter on page 235
- Clocking System chapter on page 242
- Reset System chapter on page 257
- I/O System chapter on page 261
- Watchdog Timer chapter on page 283
- Trigger Multiplexer Block chapter on page 294
- Profiler chapter on page 299

Top Level Architecture

Figure C-1. System-Wide Resources Block Diagram



18. Power Supply and Monitoring

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU family supports an operating voltage range of 1.7 V to 3.6 V. It integrates multiple regulators including an on-chip buck converter to power the blocks within the device in various power modes. The device supports multiple power supply rails – V_{DDD}, V_{DDA}, V_{DDIO}, and V_{BACKUP} – enabling the application to use a dedicated supply for different blocks within the device. For instance, V_{DDA} is used to power analog peripherals such as ADC and opamps.

The PSoC 6 MCU family supports power-on-reset (POR), brownout detection (BOD), over-voltage protection (OVP), and low-voltage-detection (LVD) circuit for power supply monitoring and failure protection purposes.

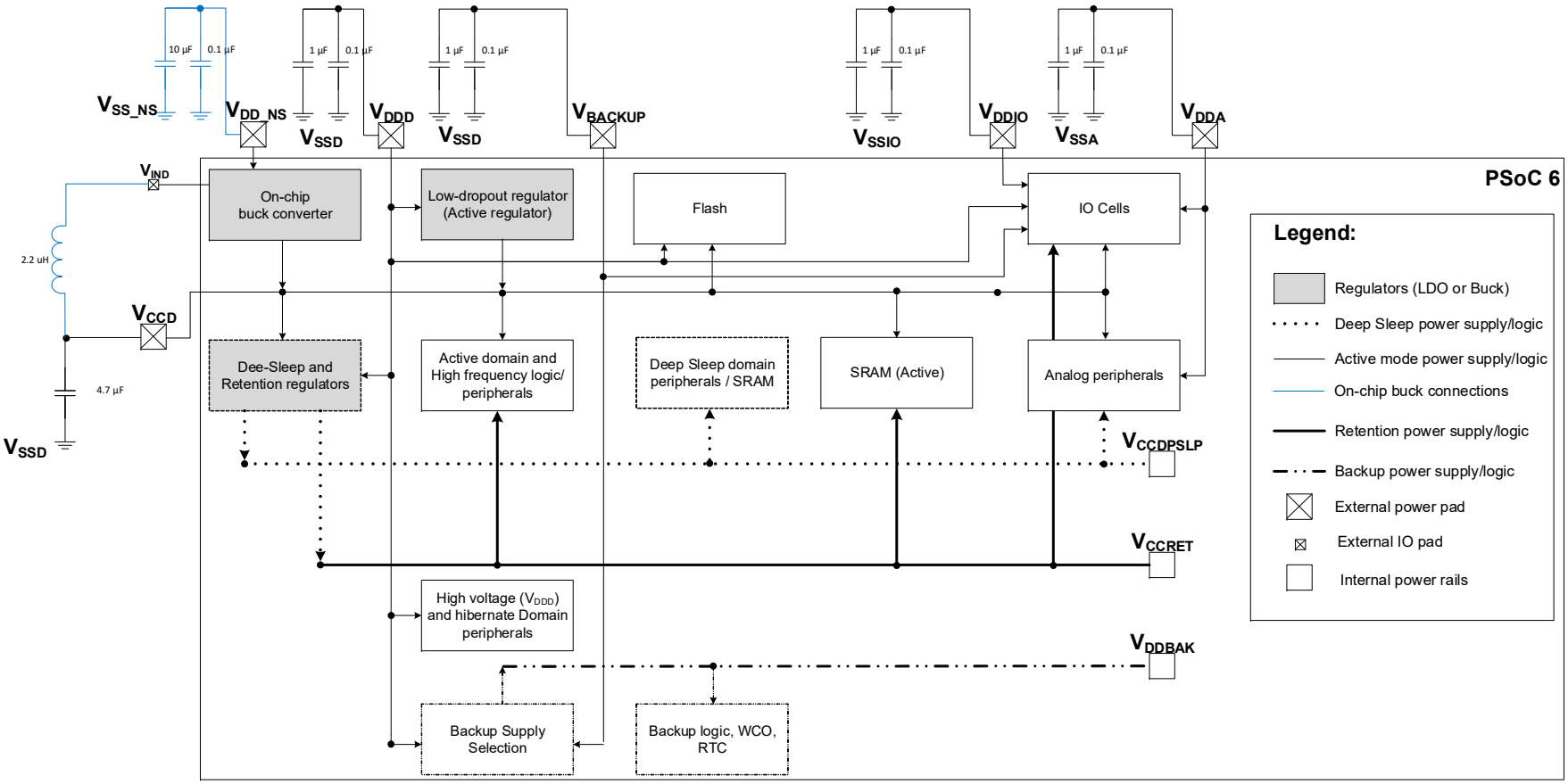
18.1 Features

The power supply subsystem of the PSoC 6 MCU supports the following features:

- Operating voltage range of 1.7 V to 3.6 V
- User-selectable core logic operation at either 1.1 V (LP) or 0.9 V (ULP)
- Three independent supply rails (V_{DDD}, V_{DDA}, and V_{DDIO}) for PSoC core peripherals and one independent supply rail (V_{BACKUP}) for backup domain
- Multiple on-chip regulators
  - One low-dropout (LDO) regulator to power peripherals in Active power mode
  - One buck converter
  - Multiple low-power regulators to power peripherals operating in different low-power modes
- Two BOD circuit (V_{DDD} and V_{CCD}) in all power modes except Hibernate mode
- LVD circuit to monitor V_{DDD}, V_{AMUXA}, V_{AMUXB}, V_{BACKUP}, or V_{DDIO}
- One OVP block monitoring V_{CCD}

18.2 Architecture

Figure 18-1. Power System Block Diagram



See the PSoC 61 datasheet/PSoC 62 datasheet for the values to be used for the capacitors and inductor shown in Figure 18-1.

The regulators and supply pins/rails shown in Figure 18-1 power various blocks inside the device. The availability of various supply rails/pins for an application will depend on the device package selected. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for details.

All the core regulators draw their input power from the V_{DDD} supply pin. The on-chip buck uses the V_{DD_NS} supply pin as its input. V_{CCD} supply is used to power all active domain and high frequency peripherals. The output of the buck is connected to the V_{CCD} pin and in firmware the V_{CCD} supply can be switched to the buck output. A dedicated Deep Sleep regulator powers all the Deep Sleep peripherals. The Deep Sleep regulator switches its output to V_{CCD} when available and to its regulated output when V_{CCD} is not present (System Deep Sleep power mode). Hibernate domain does not implement any regulators and the peripherals available in that domain such as Low-Power comparator and ILO operate directly from V_{DDD}.

When the V_{DDA} pin is not present, analog peripherals run directly from the V_{DDD}.

The I/O cells operate from various V_{DDx} (V_{DDA}, V_{DDD}, or V_{DDIO}) pins depending on the port where they are located. V_{CCD} supply is used to drive logic inside the I/O cells from core peripherals. V_{DDA} powers the analog logic such as analog mux switches inside the I/O cell. To know which I/Os operate from which supply, refer to the PSoC 61 datasheet/ PSoC 62 datasheet.

The device includes a V_{BACKUP} supply pin to power a small set of peripherals such as RTC and WCO, which run independent from other supply rails available in the device. When the V_{BACKUP} supply is not present, the device uses V_{DDD} to power these peripherals.

In addition to the power rails and regulators, the device provides options to monitor supply rails and protection against potential supply failures. These include a POR circuit, a BOD circuit, an OVP circuit, and a LVD circuit.

18.3 Power Supply

18.3.1 Regulators Summary

18.3.1.1 Core Regulators

The device includes the following core regulators to power peripherals and blocks in various power modes.

Linear Core Regulator

The device includes a linear LDO regulator to power the Active and Sleep mode peripherals. This regulator generates the core voltage (V_{CCD}) required for Active mode operation of the peripherals from V_{DDD}. The regulator is capable of providing 0.9 V and 1.1 V for core operation. See Core Operating Voltage on page 221. The regulator is available in Active and Sleep power modes. This regulator implements two sub-modes – high-current and low-current modes. LINREG_LPMODE bit [24] of the PWR_CTL register selects between the two modes of operation. The high-current mode is the normal operating mode, that is, the device operates to its full capacity in Active or Sleep power modes. In the low-current mode or the minimum regulator current mode, the current output from the regulator is limited. This mode implements the Low-Power Active and Sleep power modes. The low-current mode sets a limitation on the capabilities and availability of resources in the Low-Power Active and Sleep modes. For details, see the Device Power Modes chapter on page 225.

By default, the linear regulator is powered on reset. The regulator can be disabled by setting the LINREG_DIS bit [23] of the PWR_CTL register. Note that the linear regulator should be turned OFF only when the following conditions are satisfied:

- Switching buck regulator is ON.
- The load current requirement of the device from the V_{CCD} supply does not exceed 20 mA. This should be ensured by the firmware by disabling power consuming high-frequency peripherals and reducing the system clock frequency.

If the linear regulator is turned OFF without the above conditions satisfied, it will result in V_{CCD} brownout and the device will reset.

Switching (Buck) Core Regulator

The device includes a switching (buck) core regulator. The buck requires an inductor and a capacitor to generate the output. Note that the buck output is also available in the Deep Sleep device power mode.

The buck regulator can be enabled by setting the BUCK_EN bit [30] of the PWR_BUCK_CTL register. The buck output can be enabled/disabled using the BUCK_OUT1_EN bit [31] of the PWR_BUCK_CTL registers. The buck output supports voltages from 0.85 V to 1.20 V. Use either 0.9 V or 1.1 V for V_{CCD} operation.

The output selection can be made using BUCK_OUT_SEL bits in the PWR_BUCK_CTL registers.

When used to power the core peripherals, the buck regulator provides better power efficiency than the linear regulator, especially at higher V_{DDD}. However, the buck regulator has less load current capability (20 mA individually or 30 mA combined) than the linear regulator. Therefore, when using the buck regulator, take care not to overload the regulator by running only the necessary peripherals at a lower frequency in firmware. Overload conditions can cause the buck output to drop and result in a brownout reset.

Follow these steps in firmware when switching to the buck regulator for core (V_{CCD}) operation without causing a brownout.

1. Make sure the necessary inductor and capacitor connection for the buck as explained in Figure 18-1 is present in the hardware.
2. Change the core supply voltage to 50 mV more than the final buck voltage. For instance, if the final buck voltage is 0.9 V, then set the LDO output to 0.95 V and 1.15 V for 1.1 V buck operation. Set the ACT_REG_TRIM bits[4:0] of the PWR_TRIM_PWRSYS_CTL register to '0x0B' to switch to 0.9 V and '0x1B' to switch to 1.1 V buck operation. This is discussed in Core Operating Voltage.
3. Reduce the device current consumption by reducing clock frequency and switching off blocks to meet the buck regulator's load capacity.
4. Disable System Deep Sleep mode regulators. Because the buck regulator is available in the System Deep Sleep power mode, other deep-sleep regulators can be powered down. This is done by setting the following bits in the PWR_CTL register:
   a. DPSLP_REG_DIS bit [20] – Disables the deep-sleep core regulator.
   b. RET_REG_DIS bit [21] – Disables the logic retention regulator.
   c. NWELL_REG_DIS bit [22] – Disables the nwell regulator.
5. Set the buck output to the desired value by writing '2' (for 0.9 V) or '5' (for 1.1 V) to the BUCK_OUT1_SEL bits[2:0] of the PWR_BUCK_CTL register.
6. Enable the buck regulator and output by setting the BUCK_EN bit[30] and BUCK_OUT1_EN bit[31] of the PWR_BUCK_CTL register.
7. Wait 200 µs for the buck regulator to start up and settle.
8. Disable the linear regulator by setting the LINREG_DIS bit[23] of the PWR_CTL register.

After transitioning to the buck regulator, do not switch back to the linear regulator mode to ensure proper device operation. This should happen once during powerup.

Core Operating Voltage

PSoC 6 MCUs can operate at either 0.9 V LP mode (nominal) or 1.1 V ULP mode (nominal) core voltage. On reset, the core is configured to operate at 1.1 V by default. At 0.9 V, power consumption is less, but there are some limitations. The maximum operating frequency for all HFCLK paths should not exceed 50 MHz, whereas the peripheral and slow clock should not exceed 25 MHz.

Follow these steps to change the PSoC 6 MCU core voltage:

1. While transitioning to 0.9 V (ULP mode), reduce the operating frequency to be within the HFCLK and peri clock limits defined in the PSoC 61 datasheet/PSoC 62 datasheet for ULP mode. Turn off blocks, if required, to be within the maximum current consumption limit of the linear regulator at 0.9 V.
2. Set the ACT_REG_TRIM bits[4:0] of the PWR_TRIM_PWRSYS_CTL register to '0x07' for 0.9 V or '0x17' for 1.1 V. For the buck regulator, set the BUCK_OUT1_SEL bits[2:0] of the PWR_BUCK_CTL register to '0x02' for 0.9 V and '0x05' for 1.1 V.
3. In the case of 1.1 V to 0.9 V transition, the time it takes to discharge or settle to the new voltage may depend on the load. So the system can continue to operate while the voltage discharges.
4. In the case of 0.9 V to 1.1 V transition, wait 9 µs for the regulator to stabilize to the new voltage. The clock frequency can be increased after the settling delay.

Notes:

- When changing clock frequencies, make sure to update wait states of RAM/ROM/FLASH. Refer to the CPU Subsystem (CPUSS) chapter on page 32 for details on the wait states.
- Flash write is not allowed in 0.9 V (ULP mode) core operation.

Other Low-power Regulators

In addition to the core active regulators, the device includes multiple low-power regulators for powering Deep Sleep peripherals/logic ($V_{CCDPSLP}$), digital retention logic/SRAM (V_{CCRET}), and N-wells (V_{NWELL}) in the device. Note that V_{NWELL} is not shown in Figure 18-1 because this rail is used across the device for powering all the N-wells in the chip. These rails are shorted to V_{CCD} in Active and Sleep power modes. V_{CCRET} powers all the Active mode logic that needs to be in retention in Deep Sleep mode.

Note that none of these power rails are available in Hibernate mode. In Hibernate mode, all the hibernate logic and peripherals operate from V_{DDD} directly and a Hibernate wakeup resets the device. For details, refer to the Device Power Modes chapter on page 225.

18.3.2 Power Pins and Rails

Table 18-1 lists all the power supply pins available in the device. The supply rails running inside the device ($V_{CCDPSLP}$, V_{CCRET}, V_{DDBAK}, and V_{NWELL}) are derived from these external supply pins/rails.

Table 18-1. Supply Pins

Supply Pin	Ground Pin	Voltage Range Supported[a]	Description
V_{DD} or V_{DDD}	V_{SS} or V_{SSD}	1.7 V to 3.6 V	V_{DD} is a single supply input for multiple supplies and V_{DDD} is a digital supply input. Either of the pins will be present in a given package.
V_{DD_NS}	V_{SS_NS}	1.7 V to 3.6 V	Supply input to the buck regulator on-chip.
V_{CCD}	V_{SS} or V_{SSD}	Capacitor (0.9 V or 1.1 V)	Core supply or bypass capacitor for the internal core regulator (LDO).
V_{DDA}	V_{SS} or V_{SSA} (if available)	1.7 V to 3.6 V	Analog supply voltage.
V_{DDIO}	V_{SS} or V_{SSIO} (if available)	1.7 V to 3.6 V	Additional I/O supply voltage.
V_{BACKUP}	V_{SS} or V_{SSD} (if available)	1.4 V to 3.6 V	Backup domain supply voltage.
V_{IND}	–	Inductor	Inductor connection for the internal buck regulator.

a. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for exact range and recommended connections.

18.3.3 Power Sequencing Requirements

V_{DDD}, V_{BACKUP}, V_{DDIO}, and V_{DDA} do not have any sequencing limitation and can establish in any order. The presence of V_{DDA} without V_{DD} or V_{DDD} can cause some leakage from V_{DDA}. However, it will not drive any analog or digital output. All the V_{DDA} pins in packages that offer multiple V_{DDA} supply pins, must be shorted externally (on the PCB). Note that the system will not exit reset until both V_{DDD} and V_{DDA} are established. However, it will not wait for other supplies to establish.

18.3.4 Backup Domain

The PSoC 6 MCU offers an independent backup supply option (V_{BACKUP}). This rail powers a small set of peripherals that includes an RTC, WCO, and a small number of retention registers. This rail is independent of all other rails and can exist even when other rails are absent. As Figure 18-1 shows, this pin sources the V_{DDBAK} rail in the device. The V_{DDBAK} rail is connected to V_{DDD} when no V_{BACKUP} supply exists. For details on the backup domain, refer to the Backup System chapter on page 235.

18.3.5 Power Supply Sources

The PSoC 6 MCU offers power supply options that support a wide range of application voltages and requirements. V_{DDD} input supports a voltage range of 1.7 V to 3.6 V. If the application voltage is in this range, then the PSoC 6 MCU (V_{DDD}) can be interfaced directly to the application voltage. In applications that have voltage beyond this range, a suitable PMIC (Buck or Boost or Buck-Boost) should be used to bring the voltage to this range.

Other supply rails and pins such as V_{DDA}, V_{DDIO}, and V_{BACKUP} exist independent of V_{DDD} and V_{CCD}.

18.4 Voltage Monitoring

The PSoC 6 MCU offers multiple voltage monitoring and supply failure protection options. This includes POR, BOD, LVD, and OVP.

18.4.1 Power-On-Reset (POR)

POR circuits provide a reset pulse during the initial power ramp. POR circuits monitor V_{DDD} voltage. Typically, the POR circuits are not very accurate about the trip-point.

POR circuits are used during initial chip power-up and then disabled. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for details on the POR trip-point levels.

18.4.2 Brownout-Detect (BOD)

The BOD circuit protects the operating or retaining logic from possibly unsafe supply conditions by applying reset to the device. The PSoC 6 MCU offers two BOD circuits – high-voltage BOD (HVBOD) and low-voltage BOD (LVBOD). The HVBOD monitors the V_{DDD} voltage and LVBOD monitors the V_{CCD} voltage. Both BOD circuits generate a reset if a voltage excursion dips below the minimum V_{DDD}/V_{CCD} voltage required for safe operation (see the PSoC 61 datasheet/PSoC 62 datasheet for details). The system will not come out of RESET until the supply is detected to be valid again.

The HVBOD circuit guarantees a reset in System LP, ULP, and Deep Sleep power modes before the system crashes,

provided the V_{DDD} supply ramp satisfies the datasheet maximum supply ramp limits in that mode. There is no BOD support in Hibernate mode. Applications that require BOD support should not use Hibernate mode and should disable it. Refer to the Device Power Modes chapter on page 225 for details.

The LVBOD, operating on V_{CCD}, is not as robust as the HVBOD. The limitation is because of the small voltage detection range available for LVBOD on the minimum allowed V_{CCD}.

For details on the BOD trip-points, supported supply ramp rate, and BOD detector response time, refer to the PSoC 61 datasheet/PSoC 62 datasheet.

18.4.3 Low-Voltage-Detect (LVD)

An LVD circuit monitors external supply voltage and accurately detects depletion of the energy source. The LVD generates an interrupt to cause the system to take preventive measures.

The LVD can be configured to monitor V_{DDD}, V_{AMUXA}, V_{AMUXB}, or V_{DDIO}. The HVLVD1_SRCSEL bits [6:4] of the PWR_LVD_CTL register selects the source of the LVD. The LVD support up to 15 voltage levels (thresholds) to monitor between 1.2 V to 3.1 V. The HVLVD1_TRIPSEL bits [3:0] of the PWR_LVD_CTL register select the threshold levels of the LVD. LVD should be disabled before selecting the threshold. The HVLVD1_EN bit [7] of the PWR_LVD_CTL register can be used to enable or disable the LVD.

Whenever the voltage level of the supply being monitored drops below the threshold, the LVD generates an interrupt. This interrupt status is available in the SRSS_INTR register. HVLVD1 bit [1] of the SRSS_INTR register indicates a pending LVD interrupt. The SRSS_INTR_MASK register decides whether LVD interrupts are forwarded to the CPU or not.

Note that the LVD circuit is available only in Active, LPACTIVE, Sleep, and LPSLEEP power modes. If an LVD is required in Deep Sleep mode, then the device should be configured to periodically wake up from Deep Sleep mode using a Deep Sleep wakeup source. This makes sure an LVD check is performed during Active/LPACTIVE mode.

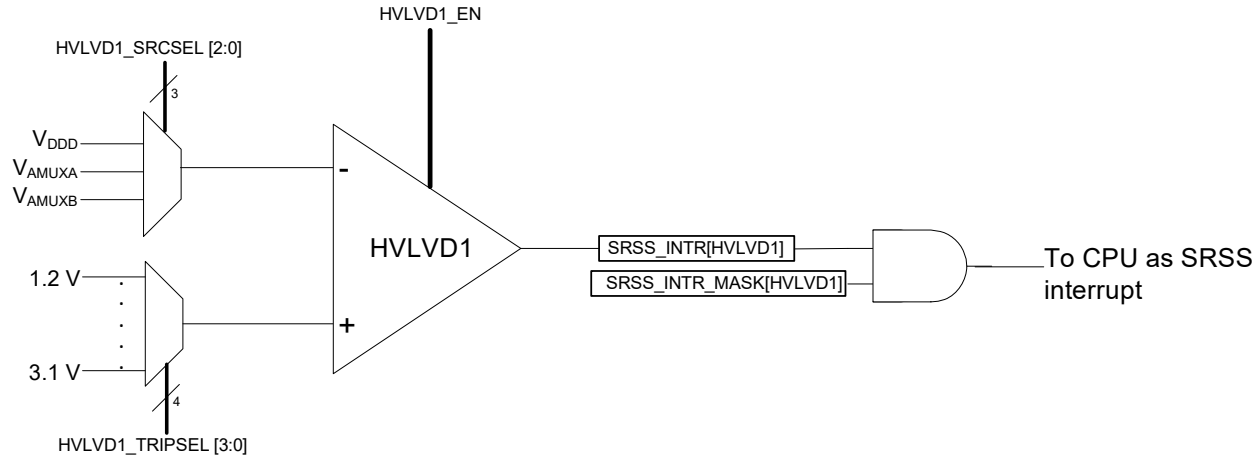
When enabling the LVD circuit, it is possible to receive a false interrupt during the initial settling time. Firmware can mask this by waiting for 8 µs after setting the HVLVD1_EN bit in the PWR_LVD_CTL register. The recommended firmware procedure to enable the LVD function is:

1. Ensure that the HVLVD1 bit in the SRSS_INTR_MASK register is 0 to prevent propagating a false interrupt.
2. Set the required trip-point in the HVLVD1_TRIPSEL field of the PWR_LVD_CTL register.
3. Configure the LVD edge (falling/rising/both) that triggers the interrupt by configuring HVLVD1_EDGE_SEL bits[1:0] of SRSS_INTR_CFG register. By default, the configuration disables the interrupt. **Note:** LVD logic may falsely detect a falling edge during Deep Sleep entry. This applies only when HVLVD1_EDGE_SEL is set to FALLING(2) or BOTH(3). Firmware can workaround this condition by disabling falling edge detection before entering Deep Sleep, and re-enabling it after exiting Deep Sleep.
4. Enable the LVD by setting the HVLVD1_EN bit in the PWR_LVD_CTL register. This may cause a false LVD event.
5. Wait at least 8 µs for the circuit to stabilize.
6. Clear any false event by setting the HVLVD1 bit in the SRSS_INTR register. The bit will not clear if the LVD condition is truly present.
7. Unmask the interrupt by setting the HVLVD1 bit in SRSS_INTR_MASK.

LVD is used to detect potential brownouts. When the supply being monitored drops below the threshold, the interrupt generated can be used to save necessary data to flash, dump logs, trigger external circuits, and so on.

For details on supported LVD thresholds, refer to the PSoC 61 datasheet/PSoC 62 datasheet and the PWR_LVD_CTL register definition in the registers TRM.

Figure 18-2. PSoC 6 MCU LVD Block



18.4.4 Over-Voltage Protection (OVP)

The PSoC 6 MCU offers an over-voltage protection circuit that monitors the V_{CCD} supply. Similar to the BOD circuit, the OVP circuit protects the device from unsafe supply conditions by applying a reset. As the name suggests, the OVP circuit applies a device reset, when the V_{CCD} supply goes above the maximum allowed voltage. The OVP circuit can generate a reset in all device power modes except the Hibernate mode.

18.5 Register List

Name	Description
PWR_CTL	Power Mode Control register - controls the device power mode options and allows observation of current state
PWR_BUCK_CTL	Buck Control register - controls the buck output and master buck enable
PWR_LVD_CTL	LVD Configuration register
SRSS_INTR	SRSS Interrupt register - shows interrupt requests from the SRSS peripheral
SRSS_INTR_MASK	SRSS Interrupt Mask register - controls forwarding of the interrupt to CPU
SRSS_INTR_SET	SRSS Interrupt Set register - sets interrupts; this register is used for firmware testing
SRSS_INTR_MASKED	SRSS Interrupt Masked register - logical AND of corresponding SRSS interrupt request (SRSS Interrupt register) and mask bits (SRSS Interrupt Mask register)

19. Device Power Modes

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU can operate in four system and three CPU power modes. These modes are intended to minimize the average power consumption in an application. The power modes supported by PSoC 6 MCUs, in the order of decreasing power consumption, are:

- System Low Power (LP) – All peripherals and CPU power modes are available at maximum speed and current
- System Ultra Low Power (ULP) – All peripherals and CPU power modes are available, but with limited speed and current
- CPU Active – CPU is executing code in system LP or ULP mode
- CPU Sleep – CPU code execution is halted in system LP or ULP mode
- CPU Deep Sleep – CPU code execution is halted and system deep sleep is requested while in system LP or ULP mode
- System Deep Sleep – Entered only after both CPUs enter CPU Deep Sleep mode. Only low-frequency peripherals are available
- System Hibernate – Device and I/O states are frozen and the device resets on wakeup

CPU Active, Sleep, and Deep Sleep are standard Arm-defined power modes supported by the Arm CPU instruction set architecture (ISA). System LP, ULP, Deep Sleep and Hibernate modes are additional low-power modes supported by PSoC 6. Hibernate mode is the lowest power mode in the PSoC 6 MCU and on wakeup, the CPU and all peripherals go through a reset.

19.1 Features

The PSoC 6 MCU power modes have the following features:

- Four system and three CPU power modes aimed at optimizing power consumption in an application
- System ULP mode with reduced operating current and clock frequency while supporting full device functionality
- System Deep Sleep mode with support for multiple wakeup sources and configurable amount of SRAM retention
- System Hibernate mode with wakeup from I/O, comparator, WDT, RTC, and timer alarms

The power consumption in different power modes is further controlled by using the following methods:

- Enabling and disabling clocks to peripherals
- Powering on/off clock sources
- Powering on/off peripherals and resources inside the PSoC 6 device

19.2 Architecture

The PSoC 6 device supports multiple power modes. Some modes only affect the CPUs (CPU power modes) and others affect the whole system (system power modes). The system and CPU power modes are used in combination to control the total system performance and power. CPU power modes are entered separately for each CPU on the device.

The SysPm Peripheral Driver Library (PDL) driver supports all device power mode transitions and is the recommended method of transition and configuration of PSoC 6 MCU power resources.

Table 19-1 summarizes the power modes available in PSoC 6 MCUs, their description, and details on their entry and exit conditions.

Table 19-1. PSoC 6 MCU Power Modes

System Power Mode	MCU Power Mode	Description	Entry Conditions	Wakeup Sources	Wakeup Action
LP	Active	System – Primary mode of operation. 1.1 V core voltage. All peripherals are available (programmable). Maximum clock frequencies CPU – Active mode	Reset from external reset, brownout, power on reset system and Hibernate mode. Manual register write from system ULP mode. Wakeup from CPU Sleep or CPU Deep Sleep while in system LP mode. Wakeup from system Deep Sleep after entered from LP mode.	Not applicable	N/A
	Sleep	1.1 V core voltage. One or more CPUs in Sleep mode (execution halted). All peripherals are available (programmable). Maximum clock frequencies	In system LP mode, CPU executes WFI/WFE instruction with Deep Sleep disabled	Any interrupt to CPU	Interrupt
	Deep Sleep	1.1 V core voltage. One CPU in Deep Sleep mode (execution halted). Other CPU in Active or Sleep mode. All peripherals are available (programmable). Maximum clock frequencies	In system LP mode, CPU executes WFI/WFE instruction with Deep Sleep enabled	Any interrupt to CPU	Interrupt
ULP	Active	0.9 V core voltage. All peripherals are available (programmable). Limited clock frequencies. No Flash write.	Manual register write from system LP mode. Wakeup from CPU Sleep or CPU Deep Sleep while in system ULP mode. Wakeup from system Deep Sleep after entered from ULP mode.	Not applicable	N/A
	Sleep	0.9 V core voltage. One or more CPUs in Sleep mode (execution halted). All peripherals are available (programmable). Limited clock frequencies. No Flash write.	In system ULP mode, CPU executes WFI/WFE instruction with Deep Sleep disabled	Any interrupt to CPU	Interrupt
	Deep Sleep	0.9 V core voltage. One CPU in Deep Sleep mode (execution halted). Other CPU in Active or Sleep mode. All peripherals are available (programmable). Limited clock frequencies. No Flash write.	In system ULP mode, CPU executes WFI/WFE instruction with Deep Sleep enabled	Any interrupt to CPU	Interrupt
Deep Sleep	Deep Sleep	All high-frequency clocks and peripherals are turned off. Low-frequency clock (32 kHz) and low-power analog and digital peripherals are available for operation and as wakeup sources. SRAM is retained (programmable).	Both CPUs simultaneously in CPU Deep Sleep mode.	GPIO interrupt, Low-Power comparator, SCB, watchdog timer, and RTC alarms	Interrupt
Hibernate	N/A	GPIO states are frozen. All peripherals and clocks in the device are completely turned off except optional low-power comparators and backup domain. Wakeup is possible through WAKEUP pins, XRES, low-power comparator (programmable), WDT, and RTC alarms (programmable). Device resets on wakeup.	Manual register write from LP or ULP modes.	WAKEUP pin, low- power comparator, watchdog timer[a], and RTC[b] alarms	Reset

a. Watchdog timer is capable of generating a hibernate wakeup. See the Watchdog Timer chapter on page 283 for details.
b. RTC (along with WCO) is part of the backup domain and is available irrespective of the device power mode. RTC alarms are capable of waking up the device from any power mode.

19.2.1 CPU Power Modes

The CPU Active, Sleep, and Deep Sleep modes are the standard Arm-defined power modes supported by both Cortex-M4 and Cortex-M0+ CPUs. All Arm CPU power modes are available in both system LP and ULP power modes. CPU power modes affect each CPU independently.

19.2.1.1 CPU Active Mode

In CPU Active mode, the CPU executes code and all logic and memory is powered. The firmware may decide to enable or disable specific peripherals and power domains depending on the application and power requirements. All the peripherals are available for use in Active mode. The device enters CPU Active mode upon any device reset or wakeup.

19.2.1.2 CPU Sleep Mode

In CPU Sleep mode, the CPU clock is turned off and the CPU halts code execution. Note that in the PSoC 6 MCU, Cortex-M4 and Cortex-M0+ both support their own CPU Sleep modes and each CPU can be in sleep independent of the other CPU state. All peripherals available in Active mode are also available in Sleep mode. Any peripheral interrupt, masked to the CPU, will wake the CPU to Active mode. Only the CPU(s) with the interrupt masked will wake.

19.2.1.3 CPU Deep Sleep Mode

In CPU Deep Sleep mode, the CPU requests the device to go into system Deep Sleep mode. When the device is ready, it enters Deep Sleep mode as detailed in 19.2.3 System Deep Sleep Mode.

Because PSoC 6 has more than one CPU, both CPUs must independently enter CPU Deep Sleep before the system will transition to system Deep Sleep.

19.2.2 System Power Modes

System power modes affect the whole device and may be combined with CPU power modes.

19.2.2.1 System Low Power Mode

System Low Power (LP) mode is the default operating mode of the device after reset and provides maximum system performance. In LP mode all resources are available for operation at their maximum power level and speed.

While in system LP mode the CPUs may operate in any of the standard Arm defined CPU modes detailed in 19.2.1 CPU Power Modes.

19.2.2.2 System Ultra Low Power Mode

System Ultra Low Power (ULP) mode is identical to LP mode with a performance tradeoff made to achieve lower system current. This tradeoff lowers the core operating voltage, which then requires reduced operating clock frequency and limited high-frequency clock sources. Flash write operations are not available in ULP mode. Table 19-5 provides the list of resources available in ULP mode along with limitations.

While in system ULP mode, the CPUs may operate in any of the standard Arm-defined CPU modes detailed in 19.2.1 CPU Power Modes.

Transitioning between LP and ULP modes is performed by reducing the core regulator voltage from the LP mode voltage to the ULP mode voltage. The lower voltage reduces system operating current and slows down signal speeds requiring a lower maximum operating frequency. Refer to Core Operating Voltage section in Power Supply and Monitoring chapter on page 218 for details on how to switch between LP and ULP modes.

19.2.3 System Deep Sleep Mode

In system Deep Sleep mode, all the high-speed clock sources are off. This in turn makes high-speed peripherals unusable in system Deep Sleep mode. However, low-speed clock sources and peripherals may continue to operate, if configured and enabled by the firmware. In addition, the peripherals that do not need a clock or receive a clock from their external interface (I^2C or SPI slave) may continue to operate, if configured for system Deep Sleep operation. The PSoC 6 MCU provides an option to configure the amount of SRAM, in blocks of 32 KB, that are retained during Deep Sleep mode.

Both Cortex-M0+ and Cortex-M4 can enter CPU Deep Sleep mode independently. However, the entire device enters system Deep Sleep mode only when both the CPUs are in CPU Deep Sleep. On wakeup, the CPU that woke up enters CPU Active mode and the other CPU remains in CPU Deep Sleep mode. On wakeup, the system will return to LP or ULP mode based on what mode was Active before entering system Deep Sleep. Both CPUs may wake up to CPU Active simultaneously from the same wakeup source if so configured.

The device enters system Deep Sleep mode after the following conditions are met.

- The LPM_READY bit of the PWR_CTL register should read ‘1’. This ensures the system is ready to enter Deep Sleep. If the bit reads ‘0’, then the device will wait in system LP or ULP mode instead of system Deep Sleep until the bit is set, at which instant the device automatically enters Deep Sleep mode, if requested.
- Both Cortex-M0+ and Cortex-M4 must be in CPU Deep Sleep. This is achieved by setting the SLEEPDEEP bit [2] of the SCR register of both Cortex-M0+ and Cortex-M4 and then executing WFI or WFE instruction.
- The HIBERNATE bit [31] of the PWR_HIBERNATE register should be cleared; otherwise, the device will enter system Hibernate mode.

In system Deep Sleep mode, the LP and ULP mode regulator is turned off and a lower power, Deep Sleep regulator sources all the peripherals enabled in system Deep Sleep mode. Alternatively, the buck regulator can be used to power the Deep Sleep peripherals. See the Power Supply and Monitoring chapter on page 218 for details. Table 19-5 provides the list of resources available in system Deep Sleep mode.

Interrupts from low-speed, asynchronous, or low-power analog peripherals can cause a CPU wakeup from system Deep Sleep mode. Note that when a debugger is running on either core, the device stays in system LP or ULP mode and the CPUs enter CPU Sleep mode instead of CPU Deep Sleep mode. PSoC 6 uses buffered writes. Therefore, writes to an MMIO register or memory can take few a cycles from the write instruction execution. The only way to ensure that the write operation is complete is by reading the same location after the write. It is required to follow a write by a read to the same location before executing a WFI/WFE instruction to enter CPU Deep Sleep mode.

19.2.4 System Hibernate Mode

System Hibernate mode is the lowest power mode of the device when external supplies are still present and XRES is deasserted. It is intended for applications in a dormant state. In this mode, both the Active LP/ULP mode regulator and Deep Sleep regulator are turned off and GPIO states are automatically frozen. Wakeup is facilitated through dedicated wakeup pins and a Low-Power comparator output. Low-Power comparator operation in Hibernate mode requires externally generated voltages for wakeup comparison. Internal references are not available in Hibernate mode. Optionally, an RTC alarm from the backup domain or a watchdog timer (16-bit free-running WDT) interrupt can generate a Hibernate wakeup signal. Set the MASK_HIBALARM bit [18] of the PWR_HIBERNATE register to enable the RTC alarm wakeup from Hibernate mode.

The device goes through a reset on wakeup from Hibernate. I/O pins remain in the configuration they were frozen before entering Hibernate mode. To differentiate between other system resets and a Hibernate mode wakeup, the TOKEN bits [7:0] of the PWR_HIBERNATE register can be used as described in the Power Mode Transitions on page 230. The PWR_HIBERNATE (except the HIBERNATE bit [31]) register along with the PWR_HIB_DATA register are retained through the Hibernate wakeup sequence and can be used by the application for retaining some content. Note that these registers are reset by other reset events. On a Hibernate wakeup event, the HIBERNATE bit [31] of the PWR_HIBERNATE register is cleared.

The brownout detect (BOD) block is not available in Hibernate mode. As a result, the device does not recover from a brownout event in Hibernate mode. Do not enter Hibernate mode in applications that require brownout detection, that is, applications where the supply is not stable. In addition, make sure the supply is stable for at least 250 µs before the device enters Hibernate mode. To prevent accidental entry into Hibernate mode in applications that cannot meet these requirements, an option to disable the Hibernate mode is provided. Set the HIBERNATE_DISABLE bit [30] of the PWR_HIBERNATE register to disable Hibernate mode in the device. Note that this bit is a write-once bit during execution and is cleared only on reset. Debug functionality will be lost and the debugger will disconnect on entering Hibernate mode.

19.2.5 Other Operation Modes

In addition to the power modes discussed in the previous sections, there are three other states the device can be in – Reset, Off, and Backup states. These states are determined by the external power supply and XRES connections. No firmware action is required to enter these modes nor an interrupt or wakeup event to exit them.

19.2.5.1 Backup Domain

PSoC 6 offers an independent backup supply option that can be supplied through a separate Vbackup pin. For details on the backup domain and the powering options, refer to the Backup System chapter on page 235. This domain powers a real-time clock (RTC) block, WCO, and a small set of backup registers. Because the power supply to these blocks come from a dedicated Vbackup pin, these blocks continue to operate in all CPU and system power modes, and even when the device power is disconnected or held in reset as long as a Vbackup supply is provided. The RTC present in the backup domain provides an option to wake up the device from any CPU or system power mode. The RTC can be clocked by an external crystal (WCO) or the internal low-speed oscillator (ILO). However, the ILO is available only if the device is powered – the device should not be in the off or reset state. Using ILO is not recommended for timekeeping purpose; however, it can be used for wakeup from Hibernate power mode.

19.2.5.2 Reset State

Reset is the device state when an external reset (XRES pin pulled low) is applied or when POR/BOD is asserted. Reset is not a power mode. During the reset state, all the components in the device are powered down and I/Os are tristated, keeping the power consumption to a minimum.

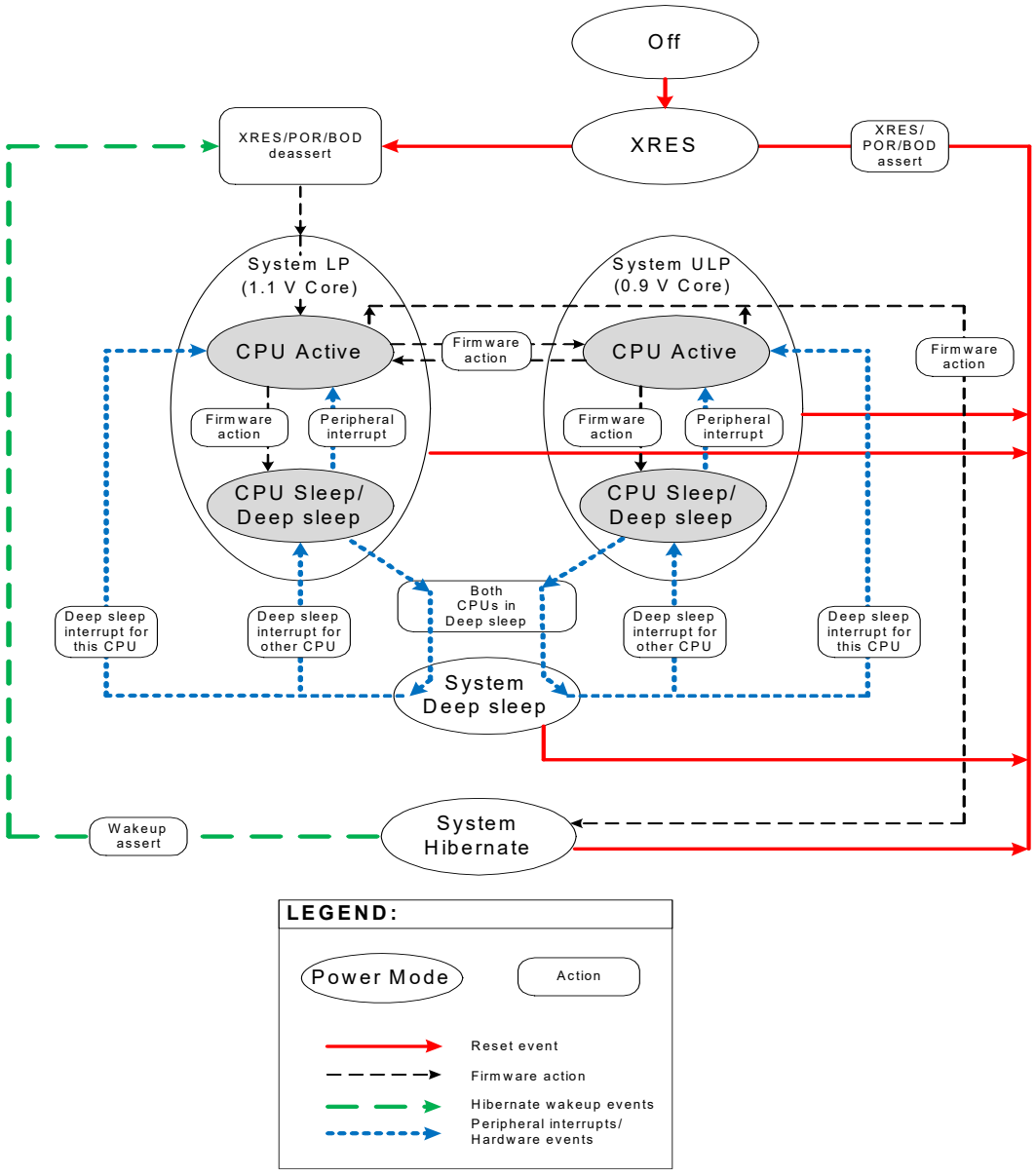
19.2.5.3 Off State

The off state simply represents the device state with no power applied. Even in the device off state, the backup domain can continue to receive power (Vbackup pin) and run the peripherals present in that domain. The reset and off states are discussed for completeness of all possible modes and states the device can be in. These states can be used in a system to further optimize power consumption. For instance, the system can control the supply of the PSoC 6 MCU by enabling or disabling the regulator output powering the device using the PMIC interface.

19.3 Power Mode Transitions

Figure 19-1 shows various states the device can be in along with possible power mode transition paths.

Figure 19-1. Power Mode Transitions in PSoC 6 MCU



19.3.1 Power-up Transitions

Table 19-2 summarizes various power-up transitions, their type, triggers, and actions.

Table 19-2. Power Mode Transitions

Initial State	Final State	Type	Trigger	Actions
Off	XRES	External	Power rail (V_{DDD}) ramps up above POR voltage level with XRES pin asserted.	1. All high-voltage logic is reset
Off	Reset	External	Power rail (V_{DDD}) ramps up above POR voltage level with XRES pin de-asserted.	1. All high-voltage logic is reset 2. Low-voltage (internal core and Deep Sleep mode) regulators and references are ramped up 3. All low-voltage logic (logic operating from internal regulators) is reset 4. IMO clock is started
XRES	Reset	External	XRES is de-asserted with V_{DDD} present and above POR level.	1. Low-voltage regulators and references are ramped up 2. All low-voltage logic is reset 3. IMO clock is started
Reset	Active	Internal	Reset sequence completes. This transition can also be caused by internal resets.	1. Clock is released to the system 2. System reset is de-asserted 3. CPU starts execution

19.3.2 Power Mode Transitions

Table 19-3. Power Mode Transitions

Initial State	Final State	Type	Trigger	Actions
System LP	System ULP	Internal	Firmware action 1. Ensure the Clk_HF paths, peripheral, and slow clocks are less than the ULP clock speed limitations. 2. Flash/SRAM/ROM wait states values are increased to ULP values as detailed in the Nonvolatile Memory chapter on page 164. 3. Configure the core regulator to 0.9 V.	1. Device is put into ULP mode with all peripherals available with limited speed. Flash write operation are not supported.
System ULP	System LP	Internal	Firmware action 1. Configure the core regulator to 1.1 V. 2. Wait 9 µs to allow the core voltage to stabilize at the new value. 3. Flash/SRAM/ROM wait states values are decreased to LP values as detailed in the Nonvolatile Memory chapter on page 164. 4. Increase the Clk_HF paths, peripheral, and slow clocks as desired up to the maximum LP clock speed specifications.	1. Device is put into LP mode with all peripherals available with maximum speed. Flash write operations are supported.

Table 19-3. Power Mode Transitions

Initial State	Final State	Type	Trigger	Actions
System LP/ ULP and CPU Active	System LP/ ULP and CPU Sleep	Internal	Firmware action 1. Clear the SLEEPDEEP bit [2] of the SCR register for both Cortex-M0+ and Cortex-M4. 2. Optionally, set the SLEEPONEXIT bit [1] of the SCR register, if the CPU runs only on interrupts. When this bit is set, the CPU does not return to application code after the WFI/WFE instruction is executed. The CPU wakes up on any enabled (masked to CPU) interrupt or event and enters CPU Sleep mode as soon as it exits the interrupt or services the event. 3. Optionally, set the SEVONPEND bit [4] of the SCR register if the application must wake up the CPU from any pending interrupt. If this bit is set, any interrupt that enters a pending state wakes up the CPU. This includes all the disabled (unmasked) interrupts to CPU. 4. Execute WFI/WFE instruction on both CPUs.	1. CPU clocks are gated off 2. CPU waits for an interrupt or event to wake it up.
System LP/ ULP and CPU Active	System LP/ UP and CPU Deep Sleep	Internal	Firmware action Perform these steps to enter Deep Sleep mode (LPM_READY bit [5] of the PWR_CTL register should read '1' before performing these steps): 1. Clear the HIBERNATE bit [31] of the PWR_HIBERNATE register. 2. Set the SLEEPDEEP bit [2] of the SCR register for one or both Cortex-M0+ and Cortex-M4. 3. Optionally, set the SLEEPONEXIT bit [1] of the SCR register if the CPU runs only on interrupts. When this bit is set, the CPU does not return to application code after the WFI/WFE instruction is executed. The CPU wakes up on any enabled (masked to CPU) interrupt or event and enters CPU Deep Sleep mode as soon as it exits the interrupt or services the event. 4. Optionally, set the SEVONPEND bit [4] of the SCR register if the application needs to wake up the CPU from any pending interrupt. If this bit is set, any interrupt that enters a pending state wakes up the CPU. This includes all the disabled (unmasked) interrupts to CPU. 5. Read the SCR register before executing a WFI/WFE instruction to ensure the write operation is complete. PSoC 6 uses buffered writes and any write transfer just before executing WFI/WFE instruction should be followed by a read to the same memory location. This ensures that the write operation has taken effect before entering Deep Sleep mode. Execute WFI/WFE instruction on both CPUs. 6. CPU in Deep Sleep mode generates a hardware request for the whole device to enter system Deep Sleep. A CPU waiting in CPU Deep Sleep state is functionally identical to CPU Sleep mode with the exception of the hardware request. 7. If only one CPU is in Deep Sleep mode the system will remain in system LP or ULP mode until the other CPU also enters CPU Deep Sleep. While waiting, a masked interrupt can wake the CPU to Active mode.	1. CPU clocks are gated off 2. CPU waits for a Deep Sleep interrupt to wake it up.
System LP/ ULP and CPU Deep Sleep	System Deep Sleep	Internal	Hardware action 1. When both CPUs enter CPU Deep Sleep mode and the LPM_READY bit [5] of the PWR_CTL register reads '1', the device will automatically transition to system Deep Sleep power mode.	1. High-frequency clocks are shut down. 2. Retention is enabled and non-retention logic is reset. 3. Active regulator is disabled and Deep Sleep regulator takes over.

Table 19-3. Power Mode Transitions

Initial State	Final State	Type	Trigger	Actions
System LP/ ULP and CPU Active	System Hibernate	Internal	Firmware action Perform these steps to enter Hibernate mode (LPM_READY bit [5] of the PWR_CTL register should read '1' before performing these steps): 1. Set the TOKEN bits [7:0] of the PWR_HIBERNATE register (optional) and PWR_HIB_DATA register to some application-specific branching data that can be used on a wakeup event from Hibernate mode. 2. Set the UNLOCK bits [8:15] of the PWR_HIBERNATE register to 0x3A, this ungates writes to FREEZE and HIBERNATE bits of the PWR_HIBERNATE register. 3. Configure wakeup pins polarity (POLARITY_HIBPIN bits [23:20]), wakeup pins mask (MASK_HIBPIN bits [27:24]) and wakeup alarm mask (MASK_HIBALARM bit [18]) in the PWR_HIBERNATE register based on the application requirement. 4. Optionally, set the FREEZE bit [17] of the PWR_HIBERNATE register to freeze the I/O pins. 5. Set the HIBERNATE bit [31] of the PWR_HIBERNATE register to enter Hibernate mode. 6. Ensure that the write operation to the PWR_HIBERNATE register is complete by reading the PWR_HIBERNATE register. Otherwise, instead of entering Hibernate mode, the CPU may start executing instructions after the write instruction. When the write is followed by a WFI/WFE instruction, the WFI/WFE can prevent the write from taking effect. On completion of the write operation, the device automatically enters Hibernate mode.	1. CPU enters low-power mode. 2. Both high-frequency and low-frequency clocks are shut down. 3. Retention is enabled and non-retention logic is reset. 4. Both Active and Deep Sleep regulators are powered down. The peripherals that are active in the Hibernate domain operate directly out of V_{DDD}.

19.3.3 Wakeup Transitions

Table 19-4. Wakeup Transitions

Initial State	Final State	Type	Trigger	Actions
CPU Sleep	CPU Active	Internal/External	Any peripheral interrupt masked to CPU	1. Clock to CPU is ungated. 2. Peripheral interrupt is serviced by CPU. 3. Device remains in current system LP or ULP power mode.
System Deep Sleep	System LP/ ULP Active and CPU Active	Internal/External	Any Deep Sleep interrupt	1. Active regulator and references are enabled. 2. Retention is disabled and non-retention reset is de-asserted. 3. High-frequency clocks are turned on. 4. CPU exits low-power mode and services the interrupt. 5. Returns to previous system LP or ULP power mode. **Note:** If only one CPU wakes up from the system Deep Sleep interrupt, then the other CPU remains in the CPU Deep Sleep state until its Deep Sleep interrupt wakes it up. However, the system will wake up to LP or ULP mode. Note that a Deep Sleep interrupt can wake up either one or both CPUs depending on the WIC configuration for the CPU. See the Interrupts chapter on page 56.
Hibernate	System LP and CPU Active	External	Wakeup pin, RTC alarm, WDT interrupt, or Low-Power comparator output asserts	1. Device is reset and goes through a reset to active power-up transition. 2. Optionally, read the TOKEN bits [7:0] of the PWR_HIBERNATE and PWR_HIB_DATA registers for application-specific branching from hibernate wakeup. 3. Optionally, set the I/O drive modes by reading the I/O frozen output and setting the I/O output to the read value. 4. Unfreeze the I/O cells by clearing the FREEZE bit [17] of the PWR_HIBERNATE register.

19.4 Summary

Table 19-5. Resources Available in Different Power Modes

Component	Power Modes							
	LP		ULP		Deep Sleep	Hibernate	XRES	Power Off with Backup
	CPU Active	CPU Sleep/ Deep Sleep	CPU Active	CPU Sleep/ Deep Sleep				
Core functions								
CPU	On	Sleep	On	Sleep	Retention	Off	Off	Off
SRAM	On	On	On	On	Retention	Off	Off	Off
Flash	Read/Write	Read/Write	Read Only	Read Only	Off	Off	Off	Off
High-Speed Clock (IMO, ECO, PLL, FLL)	On	On	On	On	Off	Off	Off	Off
LVD	On	On	On	On	Off	Off	Off	Off
ILO	On	On	On	On	On	On	Off	Off
Peripherals								
SMIF/SHDC	On	On	On	On	Retention	Off	Off	Off
SAR ADC	On	On	On	On	On	Off	Off	Off
LPCMP	On	On	On	On	On[a]	On[a]	Off	Off
TCPWM	On	On	On	On	Off	Off	Off	Off
CSD	On	On	On	On	Retention	Off	Off	Off
LCD	On	On	On	On	On	Off	Off	Off
SCB	On	On	On	On	Retention (I^2C/SPI wakeup available)[b]	Off	Off	Off
GPIO	On	On	On	On	On	Freeze	Off	Off
Watchdog timer	On	On	On	On	On	On	Off	Off
Multi-Counter WDT	On	On	On	On	On	Off	Off	Off
Resets								
XRES	On	On	On	On	On	On	On	Off
POR	On	On	On	On	On	On	Off	Off
BOD	On	On	On	On	On	Off	Off	Off
Watchdog reset	On	On	On	On	On	On[c]	Off	Off
Backup domain								
WCO, RTC, alarms	On	On	On	On	On	On	On	On

a. Low-Power comparator may be optionally enabled in the Hibernate mode to generate wakeup.
b. Only the SCB with system Deep Sleep support is available in the Deep Sleep power mode; other SCBs are not available in the Deep Sleep power mode.
c. Watchdog interrupt can generate a Hibernate wakeup. See the Watchdog Timer chapter on page 283 for details.

19.5 Register List

Name	Description
PWR_CTL	Power Mode Control register – controls the device power mode options and allows observation of current state
PWR_HIBERNATE	Hibernate Mode register – controls various Hibernate mode entry/exit related options
PWR_HIB_DATA	Hibernate Mode Data register – data register that is retained through a hibernate wakeup sequence
CM4_SCS_SCR	Cortex-M4 System Control register – controls the CM4 CPU sleep and deep sleep decisions on the WFI/WFE instruction execution. This register is detailed in the ARMv7-M Architecture Reference Manual available from Arm

Name	Description
CM0P_SCS_SCR	Cortex-M0+ System Control register – controls the CM0+ CPU sleep and deep sleep decisions on the WFI/ WFE instruction execution. This register is detailed in the ARMv7-M Architecture Reference Manual available from Arm
CPUSS_CM0_CTL	Controls the CM0+ power state. Note that this register may only be modified by the CM4 while the CM0+ is in CPU Deep Sleep mode
CPUSS_CM0_STATUS	Specifies if the CM0+ is in CPU Active, Sleep, or Deep Sleep power mode
CPUSS_CM4_PWR_CTL	Controls the CM4 power state. Note that this register may only be modified by the CM0+ while the CM4 is in CPU Deep Sleep mode.
CPUSS_CM4_STATUS	Specifies if the CM4 is in CPU Active, Sleep, or Deep Sleep power mode

20. Backup System

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The Backup domain adds an “always on” functionality to PSoC 6 MCUs using a separate power domain supplied by a backup supply (V_{BACKUP}) such as a battery or supercapacitor. It contains a real-time clock (RTC) with alarm feature, supported by a 32768-Hz watch crystal oscillator (WCO), and power-management IC (PMIC) control.

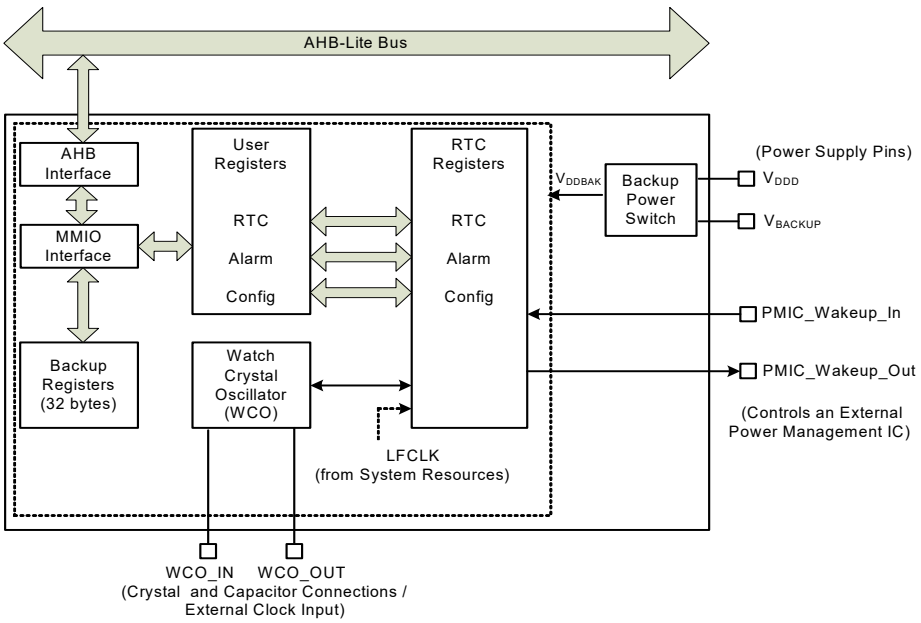
Backup is not a power mode; it is a power domain with its own power supply, which can be active during any of the device power modes. For more details, see the Power Supply and Monitoring chapter on page 218 and Device Power Modes chapter on page 225.

20.1 Features

- Fully-featured RTC
  - Year/Month/Date, Day-of-Week, Hour : Minute : Second fields
  - All fields binary coded decimal (BCD)
  - Supports both 12-hour and 24-hour formats
  - Automatic leap year correction
- Configurable alarm function
  - Alarm on Month/Date, Day-of-Week, Hour : Minute : Second fields
  - Two independent alarms
- 32768-Hz WCO with calibration
- Automatic backup power switching
- Built-in supercapacitor charger
- External PMIC control
- 32-byte backup registers

20.2 Architecture

Figure 20-1. Block Diagram



The Backup system includes an accurate WCO that can generate the clock required for the RTC with the help of an external crystal or external clock inputs. The RTC has a programmable alarm feature, which can generate interrupts to the CPU. An AHB-Lite interface provides firmware access to MMIO registers in the Backup domain.

An automatic backup power switch selects the V_{DDBAK} supply required to run the blocks in the Backup domain – either V_{DDD} (main power) or V_{BACKUP} (backup battery/ supercapacitor power).

The domain also has backup registers that can store 32 bytes of data and retain its contents even when the main supply (V_{DDD}) is OFF as long as the backup supply (V_{BACKUP}) is present. The Backup system can also control an external PMIC that supplies V_{DDD}.

20.3 Power Supply

Power to the backup system (V_{DDBAK}) is automatically switched between V_{DDD} (main supply) and V_{BACKUP} (Backup domain supply). V_{BACKUP} is typically connected to an independent supply derived from a battery or supercapacitor (see the Power Supply and Monitoring chapter on page 218 for more details).

There are no V_{BACKUP} versus V_{DDD} sequencing restrictions for the power selector switch. Either V_{BACKUP} or V_{DDD} may be removed during normal operation, and the Backup system will remain powered.

The VDDBAK_CTL bitfield in the BACKUP_CTL register controls the behavior of the power selector switch. See the registers TRM for details of this register. Possible options are:

- VDDBAK_CTL = 0 (Default mode): Selects V_{DDD} when the brownout detector in the system resources is enabled and no brownout situation is detected (see the Power Supply and Monitoring chapter on page 218 for more details). Otherwise, it selects the highest supply among V_{DDD} and V_{BACKUP}.
- VDDBAK_CTL = 1, 2, or 3: Always selects V_{BACKUP} for debug purposes.

If a supercapacitor is connected to V_{BACKUP}, the PSoC 6 MCU can charge the supercapacitor while V_{DDD} is available. Supercapacitor charging can be enabled by writing “3C” to the EN_CHARGE_KEY bitfield in the BACKUP_CTL register. Note that this feature is for charging supercapacitors only and cannot safely charge a battery. Do not write this key when V_{BACKUP} is connected to a battery. Battery charging must be handled at the board level using external circuitry.

Note: If V_{DDD} and V_{BACKUP} are connected on the PCB, the Backup domain may require an explicit reset triggered by firmware using the RESET bitfield in the BACKUP_RESET register. This firmware reset is required if the V_{BACKUP} supply was invalid during a previous power supply ramp-up or brownout event. It is not necessary to reset the Backup domain if the RES_CAUSE register indicates a non-power-related reset as the reset cause, or if the PSoC 6 MCU just

woke from the Hibernate power mode and the supply is assumed to have been valid the entire time.

It is possible to monitor the V_{BACKUP} supply using an ADC attached to AMUXBUS-A by setting the VBACKUP_MEAS bit in the BACKUP_CTL register. Note that the V_{BACKUP} signal is scaled by 10 percent so it is within the supply range of the ADC. See the SAR ADC chapter on page 506 for more details on how to connect the ADC to AMUXBUS-A.

20.4 Clocking

The RTC primarily runs from a 32768-Hz clock, after it is scaled down to one-second ticks. This clock signal can come from either of these internal sources:

- Watch-crystal oscillator (WCO). This is a high-accuracy clock generator that is suitable for RTC applications and requires a 32768-Hz external crystal populated on the application board. WCO can also operate without crystal, using external clock/sine wave inputs. These additional operating modes are explained later in this section. WCO is supplied by the Backup domain and can therefore run without V_{DDD} present.
- Alternate Backup Clock (ALTBAK): This option allows the use of LFCLK generated by the System Resources Subsystem (SRSS) as the Backup domain clock. Note that LFCLK is not available in all device power modes or when the V_{DDD} is removed. (See the Device Power Modes chapter on page 225 for more detail.)

  Clock glitches can propagate into the Backup system when LFCLK is enabled or disabled by the SRSS. In addition, LFCLK may not be as accurate as WCO depending on the actual source of LFCLK. Because of these reasons, LFCLK is not recommend for RTC applications. Also, if the WCO is intended as the clock source then choose it directly instead of routing through LFCLK.

For more details on these clocks and calibration, see the Clocking System chapter on page 242.

The RTC clock source can be selected using the CLK_SEL bitfield in the BACKUP_CTL register. The WCO_EN bit in the BACKUP_CTL register can be used to enable or disable the WCO. If the WCO operates with an external crystal, make sure the WCO_BYPASS bit in the BACKUP_CTL register is cleared before enabling the WCO. In addition, the PRESCALER bitfield in BACKUP_CTL must be configured for a prescaler value of 32768.

Note: External crystal and bypass capacitors of proper values must be connected to WCO_IN and WCO_OUT and pins. See the PSoC 61 datasheet/PSoC 62 datasheet for details of component values and electrical connections. In addition, GPIOs must be configured for WCO_OUT and WCO_IN signals. See the I/O System chapter on page 261 to know how to configure the GPIOs.

20.4.1 WCO with External Clock/Sine Wave Input

The WCO can also operate from external clock/sine wave inputs. In these modes, WCO must be bypassed by setting the WCO_BYPASS bit in the BACKUP_CTL register before enabling the WCO. Also, GPIOs must be configured for WCO_OUT and WCO_IN signals (in Analog mode). The external clock/sine wave input modes, prescaler settings, and electrical connections are as follows:

- 32768-Hz external clock mode: In this mode, WCO_IN is floating and WCO_OUT is externally driven by a 32768-Hz square wave clock toggling between ground and V_{DDD} supply levels. In this configuration, the WCO_OUT pin functions as a digital input pin for the external clock. The PRESCALER bitfield in BACKUP_CTL must be configured for a prescaler value of 32768.
- 60-Hz external clock mode: This mode can be used for deriving a clock from the 60-Hz AC mains supply. In this mode, WCO_OUT is floating and WCO_IN is driven with an external sine wave with zero DC offset, derived from the 60-Hz/120-V mains through a 100:1 capacitive divider. For example, a suitable capacitive divider can be formed by connecting a 220-pF/6-V capacitor between WCO_IN and ground, and a 2.2-pF/ 200-V capacitor between WCO_IN and the 60-Hz/120-V mains input. The PRESCALER bitfield in BACKUP_CTL must be configured for a prescaler value of 60.
- 50-Hz external clock mode: This mode is similar to the 60-Hz mode, and can be used for 50-Hz/220-V mains standard. The capacitive divider explained previously can be modified to fit this type of supply by having a 1-pF /250-V capacitor between WCO_IN and the mains input. The PRESCALER bitfield in BACKUP_CTL must be configured for a prescaler value of 50.

20.4.2 Calibration

The absolute frequency of the clock input can be calibrated using the BACKUP_CAL_CTL register. Calibration only works when the PRESCALER bitfield in BACKUP_CTL is set to 32768.

CALIB_VAL is a 6-bit field in the BACKUP_CAL_CTL register that holds the calibration value for absolute frequency (at a fixed temperature). One bit of this field translates into 128 ticks to be added or removed from the clock count. Therefore, each bit translates to a change of 1.085 ppm (= 128/(60*60*32768)).

The CALIB_SIGN field in the BACKUP_CAL_CTL register controls whether the ticks are added (it takes fewer clock ticks to count one second) or subtracted (it takes more clock ticks to count one second).

For more information, see the BACKUP_CAL_CTL register in the registers TRM.

20.5 Reset

The PSoC 6 MCU reset sources that monitor device power supply such as power-on reset (POR) and brownout reset (BOD) cannot reset the backup system as long as the backup supply (V_{DDBAK}) is present. Moreover, internal and external resets such as watchdog timer (WDT) reset and XRES cannot reset the backup system. The backup system is reset only when:

- all the power supplies are removed from the Backup domain, also known as a “cold-start”.
- the firmware triggers a Backup domain reset using the RESET bitfield in the BACKUP_RESET register.

20.6 Real-Time Clock

The RTC consists of seven binary coded decimal (BCD) fields and one control bit as follows:

Table 20-1. RTC Fields

Bitfield Name	Number of Bits	Description
RTC_SEC	7	Calendar seconds in BCD, 0-59
RTC_MIN	7	Calendar minutes in BCD, 0-59
RTC_HOUR	6	Calendar hours in BCD; value depends on 12-hour or 24-hour format set in the BACKUP_RTC_TIME register. In 24-hour mode, bits RTC_HOUR[5:0] = 0–23 In 12-hour mode, bit RTC_HOUR[5] = 0 for AM and 1 for PM Bits RTC_HOUR[4:0] = 1–12
CTRL_12HR	1	Select the 12-hour or 24-hour mode: 1=12HR, 0=24HR
RTC_DAY	3	Calendar day of the week in BCD, 1-7 You should define the meaning of the values
RTC_DATE	6	Calendar day of the month in BCD, 1-31 Automatic leap year correction
RTC_MON	4	Calendar month in BCD, 1-12
RTC_YEAR	8	Calendar year in BCD, 0-99 (Can be used to represent years 2000 -2099)

BCD encoding indicates that each four-bit nibble represents one decimal digit. Constant bits are omitted in the RTC implementation. For example, the maximum RTC_SEC is 59, which can be represented as two binary nibbles 0101b 1001b. However, the most significant bit is always zero and is therefore omitted, making the RTC_SEC a 7-bit field.

The RTC supports both 12-hour format with AM/PM flag, and 24-hour format for “hours” field. The RTC also includes a “day of the week” field, which counts from 1 to 7. You should define which weekday is represented by a value of ‘1’.

The RTC implements automatic leap year correction for the Date field (day of the month). If the Year is divisible by 4, the month of February (Month=2) will have 29 days instead of 28. When the year reaches 2100 - the Year field rolls over from 99 to 00 - the leap year correction will be wrong (2100 is flagged as a leap year which it is not); therefore, an interrupt is raised to allow the firmware to take appropriate actions. This interrupt is called the century interrupt.

User registers containing these bitfields are BACKUP_RTC_TIME and BACKUP_RTC_DATE. See the corresponding register descriptions in the registers TRM for details. As the user registers are in the high-frequency bus-clock domain and the actual RTC registers run from the low-frequency 32768-Hz clock, reading and writing RTC registers require special care. These processes are explained in the following section.

20.6.1 Reading RTC User Registers

To start a read transaction, the firmware should set the READ bit in the BACKUP_RTC_RW register. When this bit is set, the RTC registers will be copied to user registers and frozen so that a coherent RTC value can safely be read by the firmware. The read transaction is completed by clearing the READ bit.

The READ bit cannot be set if:

- RTC is still busy with a previous operation (that is, the RTC_BUSY bit in the BACKUP_STATUS register is set)
- WRITE bit in the BACKUP_RTC_RW register is set

The firmware should verify that the above bits are not set before setting the READ bit.

20.6.2 Writing to RTC User Registers

When the WRITE bit in the BACKUP_RTC_RW register is set, data can be written into the RTC user registers; otherwise, writes to the RTC user registers are ignored. When all the RTC writes are done, the firmware must clear the WRITE bit for the RTC update to take effect. After the

WRITE bit is cleared, the hardware will copy all the new data on one single WCO clock edge to ensure coherency to the actual RTC registers.

The WRITE bit cannot be set if:

- RTC is still busy with a previous operation (that is, the RTC_BUSY bit in the BACKUP_STATUS register is set)
- READ bit in the BACKUP_RTC_RW register is set

The firmware should make sure that the values written to the RTC fields form a coherent legal set. The hardware does not check the validity of the written values. Writing illegal values results in undefined behavior of the RTC.

When in the middle of an RTC update with the WRITE bit set, and a brownout, reset, or entry to Deep Sleep or Hibernate mode occurs, the write operation will not be complete. This is because the WRITE bit will be cleared by a reset, and the RTC update is triggered only when this bit is cleared by a WRITE transaction. If the write operation is in progress (RTC_BUSY), data corruption can occur if the system is reset or enters Deep Sleep or Hibernate mode.

20.7 Alarm Feature

The Alarm feature allows the RTC to be used to generate an interrupt, which may be used to wake up the system from Sleep, Deep Sleep, and Hibernate power modes.

The Alarm feature consists of six fields corresponding to the fields of the RTC: Month/Date, Day-of-Week, and Hour : Minute : Second. Each Alarm field has an enable bit that needs to be set to enable matching; if the bit is cleared, then the field will be ignored for matching.

The Alarm bitfields are as follows:

Table 20-2. Alarm Bitfields

Bitfield Name	Number of Bits	Description
ALARM_SEC	7	Alarm seconds in BCD, 0-59
ALARM_SEC_EN	1	Alarm second enable: 0=disable, 1=enable
ALARM _MIN	7	Alarm minutes in BCD, 0-59
ALARM _MIN_EN	1	Alarm minutes enable: 0=disable, 1=enable
ALARM _HOUR	6	Alarm hours in BCD, value depending on the 12-hour or 24-hour mode In 12-hour mode, bit ALARM _HOUR[5] = 0 for AM and 1 for PM, bits ALARM_HOUR[4:0] = 1–12 In 24-hour mode, bits ALARM_HOUR[5:0] = 0–23
ALARM _HOUR_EN	1	Alarm hour enable: 0=disable, 1=enable
ALARM_DAY	3	Calendar day of the week in BCD, 1-7 You should define the meaning of the values
ALARM_DAY_EN	1	Alarm day of the week enable: 0=disable, 1=enable
ALARM _DATE	6	Alarm day of the month in BCD, 1-31 Leap year corrected
ALARM _DATE_EN	1	Alarm day of the month enable: 0=disable, 1=enable
ALARM _MON	4	Alarm month in BCD, 1-12
ALARM _MON_EN	1	Alarm month enable: 0=disable, 1=enable
ALM_EN	1	Master enable for alarm. 0: Alarm is disabled. Fields for date and time are ignored. 1: Alarm is enabled. If none of the date and time fields are enabled, then this alarm triggers once every second.

If the master enable (ALM_EN) is set, but all alarm fields for date and time are disabled, an alarm interrupt will be generated once every second. Note that there is no alarm field for Year because the life expectancy of a chip is about 20 years and thus setting an alarm for a certain year means that the alarm matches either once or never in the lifetime of the chip.

The PSoC 6 MCU has two independent alarms. See the BACKUP_ALM1_TIME, BACKUP_ALM1_DATE, BACKUP_ALM2_TIME, and BACKUP_ALM2_DATE registers in the registers TRM for details.

Note that the alarm user registers, similar to RTC user registers, require certain special steps before read/write operations, as explained in sections Reading RTC User Registers on page 238 and Writing to RTC User Registers on page 238.

Interrupts must be properly configured for the RTC to generate interrupts/wake up events. Also, to enable RTC interrupts to wake up the device from Hibernate mode, the MASK_HIBALARM bit in the PWR_HIBERNATE register must be set. See the Device Power Modes chapter on page 225 and Interrupts chapter on page 56 for details.

The BACKUP_INTR_MASK register can be used to disable certain interrupts from the backup system.

Table 20-3. Interrupt Mask Bits

Bit Name	Description
ALARM1	Mask bit for interrupt generated by ALARM1
ALARM2	Mask bit for interrupt generated by ALARM2
CENTURY	Mask bit for century interrupt (interrupt generated when the Year field rolls over from 99 to 00)

The RTC alarm can also control an external PMIC as explained in the following section.

20.8 PMIC Control

The backup system can control an external PMIC that supplies V_{DDD}. PMIC enable is an active-high output that is available at a certain pin PMIC_Wakeup_Out. See the PSoC 61 datasheet/PSoC 62 datasheet for the location of this pin. This pin can be connected to the “enable” input of a PMIC. PMIC_Wakeup_Out will retain its state until the backup domain is powered. If there is a brownout and backup domain looses power, the pin is reset to its default high value.

Table 20-4 shows the bitfields in BACKUP_PMIC_CTL.

Table 20-4. PMIC Control Bits

Bitfield Name	Number of Bits	Description
UNLOCK	8	This byte must be set to 0x3A for PMIC to be disabled. Any other value in this field will cause writes to PMIC_EN to be ignored. Do not change PMIC_EN in the same write cycle as setting/clearing the UNLOCK code; do these in separate write cycles.
POLARITY	1	Reserved for future use. Keep this bit at ‘1’.
PMIC_EN_OUTEN	1	Output enable of the PMIC_EN pin. 0: PMIC_EN pin output is HI-Z. This allows the PMIC_EN pin to be used as a GPIO. The HI-Z condition is kept only if the UNLOCK key (0x3A) is present. 1: PMIC_EN pin output is enabled.
PMIC_ALWAYSEN	1	Override normal PMIC controls to prevent errant firmware from accidentally turning off the PMIC. 0: Normal operation; PMIC_EN and PMIC_OUTEN work as explained in their bitfield descriptions. 1: PMIC_EN is forced set; PMIC_EN and PMIC_OUTEN are ignored. This bit is a write-once bit that cannot be cleared until the next Backup domain reset.
PMIC_EN	1	Enable external PMIC (hardware output available at pin PMIC_Wakeup_Out). This bit is enabled by default. This bit will only clear if the UNLOCK field was written correctly in a previous write operation and PMIC_ALWAYSEN = 0.

Note that two writes to the BACKUP_PMIC_CTL register are required to change the PMIC_EN setting. The first write should update the desired settings (including the UNLOCK code) but should not change PMIC_EN or PMIC_EN_OUTEN. The second write must use the same bit values as the first one except desired PMIC_EN/ PMIC_EN_OUTEN settings.

When the PMIC_EN bit is cleared by firmware, the external PMIC is disabled and the system functions normally until V_{DDD} is no longer present (OFF with Backup mode). The firmware can set this bit if it does so before V_{DDD} is actually removed. The time between firmware disabling the PMIC_EN bit and the actual removal of V_{DDD} depends on the external PMIC and supply-capacitor characteristics.

Additionally, PMIC can be turned on by one of these events:

- An RTC Alarm/Century Interrupt
- A logic high input at the PMIC_Wakeup_In pin. See the PSoC 61 datasheet/PSoC 62 datasheet for the location of this pin. This allows a mechanical button or an external input from another device to wake up the system and enable the PMIC. The POLARITY bit in the

BACKUP_PMIC_CTL register must be set to '1' to use this feature. The same wakeup pin PMIC_Wakeup_In can wake up the device from Hibernate mode. See the Device Power Modes chapter on page 225 for details.

Make sure that one or more of these events are configured properly, and a battery or supercapacitor is connected to V_{BACKUP} with sufficient charge to power the backup system until one of these events occur. Otherwise, the PMIC may continue to be in the disabled state with the PSoC 6 MCU unable to enable it again.

20.9 Backup Registers

The Backup domain has sixteen registers (BACKUP_BREG0 to BACKUP_BREG15), which can be used to store 32 bytes of important information/flags. These registers retain their contents even when the main supply (V_{DDD}) is off as long as backup supply (V_{BACKUP}) is present. These registers can also be used to store information that must be retained when the device enters Hibernate mode.

20.10 Register List

Table 20-5. Backup Registers

Register Name	Description
BACKUP_CTL	Main control register (including power and clock)
BACKUP_RTC_RW	RTC read/write control register
BACKUP_STATUS	Status register
BACKUP_RTC_TIME	Calendar seconds, minutes, hours, and day of week
BACKUP_RTC_DATE	Calendar day of month, month, and year
BACKUP_ALM1_TIME	Alarm 1 seconds, minute, hours, and day of week
BACKUP_ALM1_DATE	Alarm 1 day of month, and month
BACKUP_ALM2_TIME	Alarm 2 seconds, minute, hours, and day of week
BACKUP_ALM2_DATE	Alarm 2 day of month, and month
BACKUP_INTR	Interrupt request register
BACKUP_INTR_MASK	Interrupt mask register
BACKUP_INTR_MASKED	Interrupt masked request register
BACKUP_PMIC_CTL	PMIC control register
BACKUP_BREG0 to BACKUP_BREG15	Backup registers
BACKUP_RESET	Reset register for the backup domain

21. Clocking System

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

PSoC 6 MCU provides flexible clocking options with on-chip crystal oscillators, phase lock loop, frequency lock loop, and supports multiple external clock sources.

21.1 Features

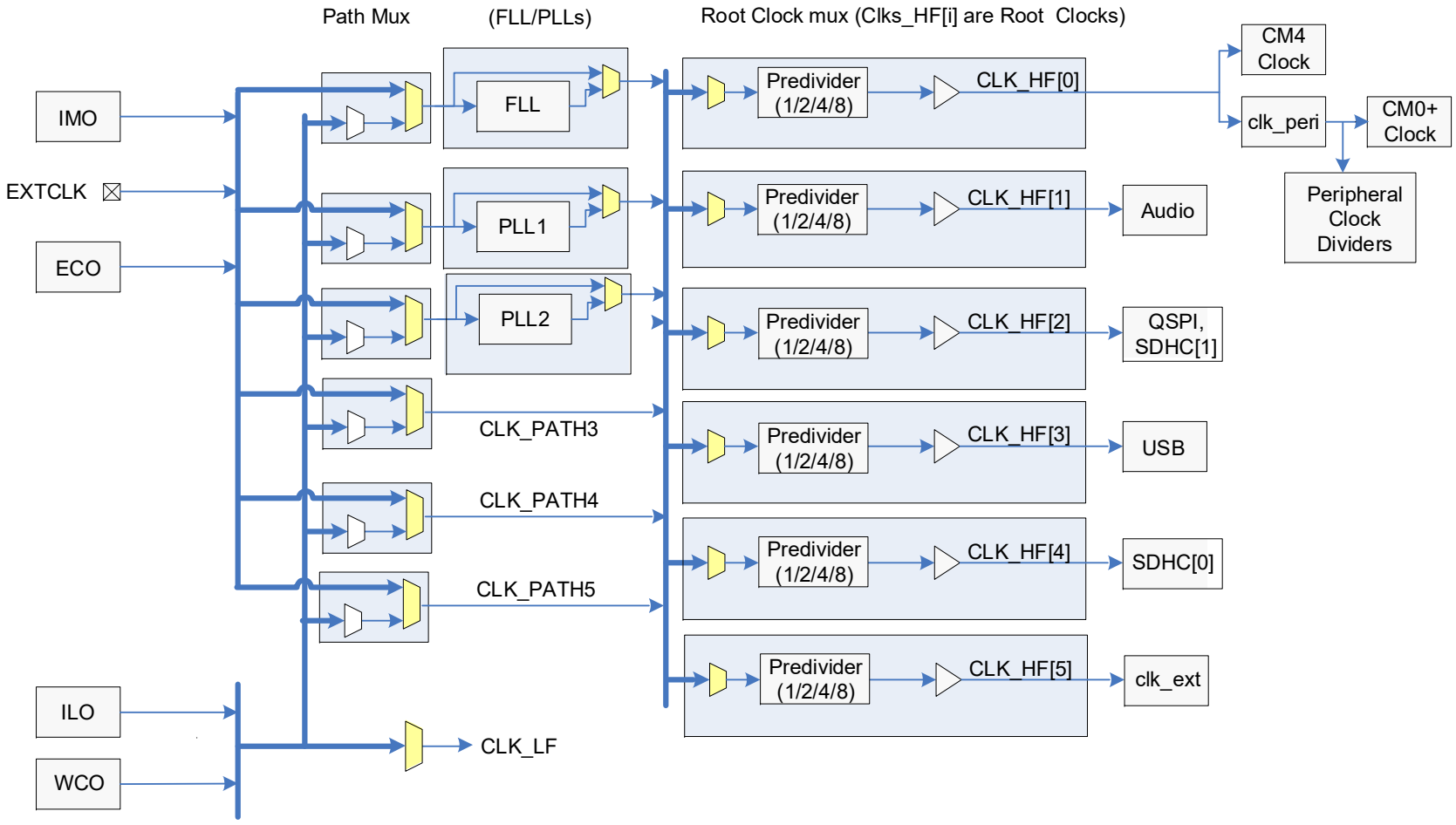
The PSoC 6 MCU clock system includes these resources:

- Two internal clock sources:
  - 8-MHz internal main oscillator (IMO)
  - 32-kHz internal low-speed oscillator (ILO)
- Three external clock sources
  - External clock (EXTCLK) generated using a signal from an I/O pin
  - External 16–35 MHz crystal oscillator (ECO)
  - External 32-kHz watch crystal oscillator (WCO)
- One frequency lock loop (FLL) with 24–100 MHz output range
- Two phase-locked loop (PLL) with 10.625–150 MHz output range

21.2 Architecture

Figure 21-1 gives a generic view of the clocking system in PSoC 6 MCUs.

Figure 21-1. Clocking System Block Diagram



21.3 Clock Sources

21.3.1 Internal Main Oscillator (IMO)

The IMO is an accurate, high-speed internal (crystal-less) oscillator that produces a fixed frequency of 8 MHz. The IMO output can be used by the PLL or FLL to generate a wide range of higher frequency clocks, or it can be used directly by the high-frequency root clocks.

When USB is present the USB Start-of-Frame (SOF) signal is used to trim the IMO to ensure that the IMO matches the accuracy of the USB SOF. The ENABLE_LOCK bitfield in the USBFS0_USBDEV_CR register of the USB block needs to be set for this feature to work. The driver for the USB block in the PDL does this automatically.

The IMO is available only in the system LP and ULP power modes.

21.3.2 External Crystal Oscillator (ECO)

The PSoC 6 MCU contains an oscillator to drive an external 16-MHz to 35-MHz crystal. This clock source is built using an oscillator circuit in PSoC. The circuit employs an external crystal that needs to be populated on the external crystal pins of the PSoC 6 MCU. See *AN218241 - PSoC 6 MCU Hardware Design Considerations* for more details.

The ECO can be enabled by using the CLK_ECO_CONFIG ECO_EN register bitfields.

21.3.2.1 ECO Trimming

The ECO supports a wide variety of crystals and ceramic resonators with the nominal frequency range specification of f = 16 MHz – 35 MHz. The crystal manufacturer typically provides numerical values for parameters, namely the maximum drive level (DL), the maximum equivalent series resistance (ESR), shunt capacitance of the crystal (C_0), and the parallel load capacitance (C_L). These parameters can be used to calculate the transconductance (g_m) and the maximum peak to peak (V_{PP}).

Max peak to peak value:

$$V_{PP} = 2 \times V_P$$

$$V_P = 2 \times \frac{\sqrt{\frac{2 \times DL}{ESR}}}{4\pi \times f \times (C_L + C_0)}$$

Transconductance:

$$g_m = 4 \times 5 \times (2\pi \times f \times (C_L + C_0))^2 \times ESR$$

ECO does not support V_{PP} less than 1.3 V.

The following fields are found in the CLK_TRIM_ECO_CTL register. The Amplitude trim (ATRIM) and WDTRIM settings control the trim for amplitude of the oscillator output. ATRIM sets the crystal drive level when automatic gain control (AGC) is enabled (CLK_ECO_CONFIG.AGC_EN = 1). AGC must be enabled all the time.

ATRIM and WDTRIM values are set at 15 and 7, respectively.

The GTRIM sets up the trim for amplifier gain based on the calculated g_m, as shown in Table 21-1.

Table 21-1. GTRIM Settings

g_m	GTRIM
g_m < 9 mA/V	0x01
9 mA/V < g_m < 18 mA/V	0x00
g_m > 18 mA/V	INT (g_m / 9 mA/V)

RTRIM and FTRIM are set to 0 and 3, respectively.

First, set up the trim values based on Table 21-1 and other listed values; then enable the ECO. After the ECO is enabled, the CLK_ECO_STATUS register can be checked to ensure it is ready.

21.3.3 External Clock (EXTCLK)

The external clock is a 0- to 100-MHz range clock that can be sourced from a signal on a designated I/O pin. This clock can be used as the source clock for either the PLL or FLL, or can be used directly by the high-frequency clocks.

When manually configuring a pin as an input to EXTCLK, the drive mode of the pin must be set to high-impedance digital to enable the digital input buffer. See the I/O System chapter on page 261 for more details. Consult the PSoC 61 datasheet/PSoC 62 datasheet to determine the specific pin used for EXTCLK. See KBA224493 for more details.

21.3.4 Internal Low-speed Oscillator (ILO)

The ILO operates with no external components and outputs a stable clock at 32.768 kHz nominal. The ILO is relatively low power and low accuracy. It is available in all power modes. If the ILO is to remain active in Hibernate mode, and across power-on-reset (POR) or brownout detect (BOD), the ILO_BACKUP bit must be set in the CLK_ILO_CONFIG register.

The ILO can be used as the clock source for CLK_LF, which in turn can be used as a source for the backup domain (CLK_BAK). CLK_BAK runs the real-time clock (RTC). This can be useful if you do not wish to populate a WCO. Although the ILO is not suitable as an RTC due to its poor accuracy, it can be used as a HIBERNATE wakeup source using the wakeup alarm facility in the RTC. In this case, the V_{DDD} rail must be supplied during HIBERNATE for the ILO to run, and the ILO_BACKUP bit must be set in the CLK_ILO_CONFIG register. CLK_LF is also the source of

the MCWDT timers; see the Watchdog Timer chapter on page 283 for details.

The ILO is always the source of the watchdog timer (WDT).

The ILO is enabled and disabled with the ENABLE bit of the CLK_ILO_CONFIG register. Always leave the ILO enabled as it is the source of the WDT.

If the WDT is enabled, the only way to disable the ILO is to first clear the WDT_LOCK bit in the WDT_CTL register and then clear the ENABLE bit in the CLK_ILO_CONFIG register. If the WDT_LOCK bit is set, any register write to disable the ILO will be ignored. Enabling the WDT will automatically enable the ILO.

The calibration counters described in Clock Calibration Counters on page 256 can be used to measure the ILO against a high-accuracy clock such as the ECO. This result can then be used to determine how the ILO must be adjusted. The ILO can be trimmed using the CLK_TRIM_ILO_CTL register.

21.3.5 Watch Crystal Oscillator (WCO)

The WCO is a highly accurate 32.768-kHz clock source. It is the primary clock source for the RTC. The WCO can also be used as a source for CLK_LF.

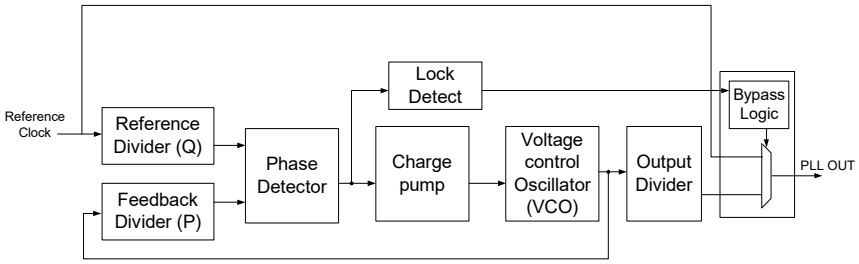
The WCO can be enabled and disabled by setting the WCO_EN bit in the CTL register for the backup domain. The WCO can also be bypassed and an external 32.768-kHz clock can be routed on a WCO output pin. This is done by setting the WCO_BYPASS bit in the CTL register for the backup domain. See WCO with External Clock/Sine Wave Input on page 237 for more details.

21.4 Clock Generation

21.4.1 Phase-Locked Loop (PLL)

The PSoC 6 MCU contains two PLLs, which reside on CLK_PATH1 and CLK_PATH2. They are capable of generating a clock output in the range 10.625–150 MHz; the input frequency must be between 4 and 64 MHz. This makes it possible to use the IMO to generate much higher clock frequencies for the rest of the system.

Figure 21-2. PLL Block Diagram



The PLL is configured following these steps:

Note: f_{ref} is the input frequency of the PLL, that is, the frequency of input clock, such as 8 MHz for the IMO.

1. Determine the desired output frequency (f_{out}). Calculate the reference (REFERENCE_DIV), feedback (FEEDBACK_DIV), and output (OUTPUT_DIV) dividers subject to the following constraints:
   a. PFD frequency (phase detector frequency). $f_{pfd} = f_{ref}$ / REFERENCE_DIV. It must be in the range 4 MHz to 8 MHz. There may be multiple reference divider values that meet this constraint.
   b. VCO frequency. $f_{vco} = f_{pfd}$ * FEEDBACK_DIV. It must be in the range 170 MHz to 400 MHz. There may be multiple feedback divider values that meet this constraint with different REFERENCE_DIV choices.
   c. Output frequency. $f_{out} = f_{vco}$ / OUTPUT_DIV. It must be in the range 10.625 MHz to 150 MHz. Note that your device may not be capable of operating at this frequency; check the device datasheeet. It may not be possible to get the desired frequency due to granularity; therefore, consider the frequency error of the two closest choices.
   d. Choose the best combination of divider parameters depending on the application. Some possible decision-making factors are: minimum output frequency error, lowest power consumption (lowest f_{vco}), or lowest jitter (highest f_{vco}).
2. Program the divider settings in the appropriate CLK_PLL_CONFIG register. Do not enable the PLL on the same cycle as configuring the dividers. Do not change the divider settings while the PLL is enabled.
3. Enable the PLL (CLK_PLL_CONFIG.ENABLE = 1). Wait at least 1 µs for PLL circuits to start.
4. Wait until the PLL is locked before using the output. By default, the PLL output is bypassed to its reference clock and will automatically switch to the PLL output when it is locked. This behavior can be changed using PLL_CONFIG.BYPASS_SEL. The status of the PLL can be checked by reading CLK_PLL_STATUS. This register contains a bit indicating the PLL has locked. It also contains a bit indicating if the PLL lost the lock status.

To disable the PLL, first set the PLL_CONFIG.BYPASS_SEL to PLL_REF. Then wait at least six PLL output clock cycles before disabling the PLL by setting PLL_CONFIG.ENABLE to '0'.

21.4.2 Frequency Lock Loop (FLL)

The PSoC 6 MCU contains one frequency lock loop (FLL), which resides on Clock Path 0. The FLL is capable of generating a clock output in the range 24 MHz to 100 MHz; the input frequency must be between 0.001 and 100 MHz, and must be at least 2.5 times less than the CCO frequency. This makes it possible to use the IMO to generate much higher clock frequencies for the rest of the system.

The FLL is similar in purpose to a PLL but is not equivalent:

- FLL can start up (lock) much faster than the PLL.
- It consumes less current than the PLL.
- FLL does not lock the phase. At the heart of the FLL is a current-controlled oscillator (CCO). The output frequency of this CCO is controlled by adjusting the trim of the CCO; this is done in hardware and is explained in detail later in this section.
- FLL can produce up to 100-MHz clock with good duty cycle through its divided clock output.
- FLL reference clock can be the WCO (32 kHz), IMO (8 MHz), or any other periodic clock source.

Note: The CCO frequency must be at least 2.5 times greater than the reference frequency.

The CCO can output a stable frequency in the 48 MHz to 200 MHz range. This range is divided into five sub-ranges as shown by Table 21-2.

Table 21-2. CCO Frequency Ranges

CCO Range	0	1	2	3	4
f_{min}	48 MHz	64 MHz	85 MHz	113 MHz	150 MHz
f_{max}	64 MHz	85 MHz	113 MHz	150 MHz	200 MHz

Note: The output of the CCO has an option to enable a divide by two or not. For this device, the divide by two must always be enabled.

Within each range, the CCO output is controlled via a 9-bit trim field. This trim field is updated via hardware based on the control algorithm described below.

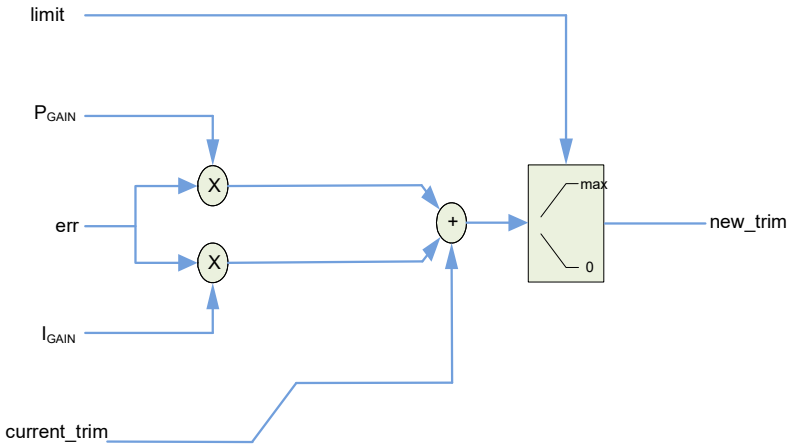
A reference clock must be provided to the FLL. This reference clock is typically the IMO, but could be many different clock sources. The FLL compares the reference clock against the CCO clock to determine how to adjust the CCO trim. Specifically, the FLL will count the number of CCO clock cycles inside a specified window of reference clock cycles. The number of reference clock cycles to count is set by the FLL_REF_DIV field in the CLK_FLL_CONFIG2 register.

After the CCO clocks are counted, they are compared against an ideal value and an error is calculated. The ideal value is programmed into the FLL_MULT field of the CLK_FLL_CONFIG register.

As an example, the reference clock is the IMO (8 MHz), the desired CCO frequency is 100 MHz, the value for FLL_REF_DIV is set to 146. This means that the FLL will count the number of CCO clocks within 146 clock periods of the reference clock. In one clock cycle of the reference clock (IMO), there should be 100 / 8 = 12.5 clock cycles of the CCO. Multiply this number by 146 and the value of FLL_MULT should be 1825.

If the FLL counts a value different from 1825, it attempts to adjust the CCO such that it achieves 1825 the next time it counts. This is done by scaling the error term with FLL_LF_IGAIN and FLL_LF_PGAIN found in CLK_FLL_CONFIG3. Figure 21-3 shows how the error (err) term is multiplied by FLL_LF_IGAIN and FLL_LF_PGAIN and then summed with the current trim to produce a new trim value for the CCO. The CCO_LIMIT field in the CLK_FLL_CONFIG4 can be used to put an upper limit on the trim adjustment; this is not needed for most situations.

Figure 21-3. FLL Error Correction Diagram



The FLL determines whether it is “locked” by comparing the error term with the LOCK_TOL field in the CLK_FLL_CONFIG2 register. When the error is less than LOCK_TOL the FLL is considered locked.

After each adjustment to the trim the FLL can be programmed to wait a certain number of reference clocks before doing a new measurement. The number of reference clocks to wait is set in the SETTLING_COUNT field of CLK_FLL_CONFIG3. Set this such that the FLL waits ~1 µs before a new count. Therefore, if the 8-MHz IMO is used as the reference this field should be programmed to ‘8’.

When configuring the FLL there are two important factors that must be considered: lock time and accuracy. Accuracy is the closeness to the intended output frequency. These two numbers are inversely related to each other via the value of REF_DIV. Higher REF_DIV values lead to higher accuracy, whereas lower REF_DIV values lead to faster lock times.

In the example used previously the 8-MHz IMO was used as the reference, and the desired FLL output was 100 MHz. For that example, there are 12.5 CCO clocks in one reference clock. If the value for REF_DIV is set to ‘1’ then FLL_MULT must be set to either ‘13’ or ‘12’. This will result in a CCO output of either 96 MHz or 104 MHz, and an error of 4 percent from the desired 100 MHz. Therefore, the best way to improve this is to increase REF_DIV. However, the larger REF_DIV is, the longer each measurement cycle takes, thus increasing the lock time. In this example, REF_DIV was set to 146. This means each measurement cycle takes 146 * (1/8 MHz) = 18.25 µs, whereas when REF_DIV is set to 1, each measurement cycle takes 1 * (1/ 8 MHz) 0.125 µs.

Another issue with lower REF_DIV values is that the minimum LOCK_TOL is 1, so the output of the CCO can have an error of ±1. In the example where REF_DIV = 1 and FLL_MULT = 13, the MULT value can really be 12, 13, or 14 and still be locked. This means the output of the FLL may vary between 96 and 112 MHz, which may not be desirable.

Thus, a choice must be made between faster lock times and more accurate FLL outputs. The biggest change to make for this is the value of REF_DIV. The following section describes how to configure all of the FLL registers and gives some example equations to set REF_DIV N for best accuracy.

21.4.2.1 Configuring the FLL

This section provides a guide to calculate FLL parameters. ModusToolbox and the PDL provide drivers and calculators that automatically populate the FLL registers based on FLL input and output frequency requirements.

The following equations are tailored to achieve the best accuracy.

In these equations:

N = REF_DIV

M = FLL _MULT

1. Set CCO_RANGE in the CLK_FLL_CONFIG4 register.
   a. Determine the output frequency of the FLL.
   b. Calculate the CCO frequency. The CCO frequency = 2 * FLL output frequency.

c. Use Table 21-3 to determine the CCO range.
d. Write CCO range into CCO_RANGE in the CLK_FLL_CONFIG4 register.

Table 21-3. CCO Frequency Ranges

CCO Range	0	1	2	3	4
f_{min}	48 MHz	64 MHz	85 MHz	113 MHz	150 MHz
f_{max}	64 MHz	85 MHz	113 MHz	150 MHz	200 MHz

2. Set the FLL_OUTPUT_DIV in CLK_FLL_CONFIG. Set the output divider to '1'.
3. Set FLL_REF_DIV in CLK_FLL_CONFIG2.

   FLL_REF_DIV divides the FLL input clock. The FLL counts the number of CCO clocks within one period of the divided reference clock. A general equation to calculate the reference divider is as follows:

$$N = ROUNDUP\left(\frac{2 \times f_{ref}}{CCO_{Trim_{Step}} \times f_{CCO_{Target}}}\right)$$

Equation 21-1

   The $CCO_{TrimStep}$ is found in Table 21-4.

Table 21-4. CCO Trim Steps

CCO Range	0	1	2	3	4
CCO_Trim_Steps	0.0011	0.0011	0.0011	0.0011	0.00117

   A larger N results in better precision on the FLL output, but longer lock times; a smaller N will result in faster lock times, but less precision.

   Note: When the WCO is used as the reference clock, N must be set to 19.
4. Set FLL_MULT in CLK_FLL_CONFIG.

   FLL_MULT is the ratio between the desired CCO frequency and the divided input frequency. This is the ideal value for the counter that counts the number of CCO clocks in one period of the divided input frequency.

$$M = f_{CCO_{Target}} \times \frac{N}{f_{ref}}$$

Equation 21-2

5. Set the FLL_LF_IGAIN and FLL_LF_PGAIN in CLK_FLL_CONFIG3.

   Within each range of the CCO there are 512 steps of adjustment for the CCO frequency. These steps are controlled by CCO_FREQ in the CLK_FLL_CONFIG4 register. The FLL automatically adjusts CCO_FREQ based on the output of the FLL counter.

   The output of the counter gives the number of CCO clocks, over one period of the divided reference clock, by which the FLL is off. This value is then multiplied by the sum of FLL_LF_IGAIN and FLL_LF_PGAIN. The result of this multiplication is then summed with the value currently in the CCO_FREQ register.

   To determine the values for I_{GAIN} and P_{GAIN} use the following equation:

$$I_{GAIN} < \left(\frac{0.85}{K_{CCO} \times \frac{N}{f_{ref}}}\right)$$

Equation 21-3

   Find the value of I_{GAIN} closest but not over the values in the gain row in Table 21-5.

Table 21-5. I_{GAIN} and P_{GAIN} Register Values

Register Value	0	1	2	3	4	5	6	7	8	9	10	11
Gain Value	1/256	1/128	1/64	1/32	1/16	1/8	1/4	1/2	1	2	4	8

Program FLL_LF_IGAIN with the register value that corresponds to the chosen gain value.

Take the I_{GAIN} value from the register and use it in the following equation:

$$P_{GAIN} < I_{GAINreg} - \frac{0.85}{K_{CCO} \times \frac{N}{f_{ref}}}$$

Equation 21-4

Find the value of P_{GAIN} closest but not over the values in the gain row in Table 21-5. Program FLL_PF_IGAIN with the register value that corresponds to the chosen gain value.

For best performance Pgain_reg + Igain_reg should be as close as possible to calculated I_{GAIN} without exceeding it. kCCO is the gain of the CCO; Table 21-6 lists the kCCO for each CCO range.

Table 21-6. kCCO Values

CCO Range	0	1	2	3	4
kCCO	48109.38	64025.65	84920	113300	154521.85

6. Set SETTLING_COUNT in CLK_FLL_CONFIG3.

   SETTLING_COUNT is the number of reference clocks to wait for the CCO to settle after it has changed. It is best to set this such that the settling time is around 1 µs.

   $$SETTLING\_COUNT = 1\ \mu s * fref$$

   Do not set the settling time to anything less than 1 µs, greater will lead to longer lock times.

1. Set LOCK_TOL in CLK_FLL_CONFIG2.

   LOCK_TOL determines how much error the FLL can tolerate at the output of the counter that counts the number of CCO clocks in one period of the divided reference clock. A higher tolerance can be used to lock more quickly or track a less accurate source. The tolerance should be set such that the FLL does not unlock under normal conditions. A lower tolerance means a more accurate output, but if the input reference is unstable then the FLL may unlock.

   The following equation can be used to help determine the value:

$$LOCK\_TOL = M \times \left[\left(\frac{1 + CCO_{accuracy}}{1 - F_{refaccuracy}} \right) - 1 \right]$$

Equation 21-5

   CCO (accuracy) = 0.25% or 0.0025

   ref (accuracy) is the accuracy of the reference clock

2. Set CCO_FREQ in CLK_FLL_CONFIG4.

   This field determines the frequency at which the FLL starts before any measurement. The nearer the FLL is to the desired frequency, the faster it will lock.

$$CCO_{FREQ} = ROUPNDUP \left(\frac{LOG\left(\frac{f_{CCO_{Target}}}{f_{CCO_{Base}}} \right)}{LOG(1 + CCO_{Trim_{Step}})} \right)$$

Equation 21-6

   $fCCO_{Base}$ can be found in Table 21-7.

Table 21-7. CCO Base Frequency

CCO Range	0	1	2	3	4
CCO_Base	43600	58100	77200	103000	132000

Table 21-8. CCO Trim Steps

CCO Range	0	1	2	3	4
CCO_Trim_Steps	0.0011	0.0011	0.0011	0.0011	0.00117

9. Calculating Precision, Accuracy, and Lock Time of FLL

   To calculate the precision and accuracy of the FLL, the accuracy of the input source must be considered.

   The precision is the larger of:

$$\text{Precision}_{FLL} = f_{ref} \times \left(\frac{1 + f_{ref}(\text{accuracy})}{N \times f_{CCO_{Target}}} \right)$$

Equation 21-7

   Or (CCO_Trim_Steps / 2)

   Example: The desired FLL output is 100 MHz, thus CCO target is 200 MHz. The 2 percent accurate 8 MHz IMO is used as the reference input. N is calculated to be 69.

   Precision_{FLL} = ((8 MHz * 1.02)/(69 * 200 MHz)) = 0.059%, which is greater than the (CCO_Trim_Steps / 2)

   The accuracy of the FLL output is the precision multiplied by the lock tolerance. If the CCO goes beyond this range, then the FLL will unlock.

   Accuracy_{FLL} = Precision_{FLL} * LOCK_TOL

   The value for the reference divider N should be tuned such that it achieves the best precision/accuracy versus lock time.

   The lock time depends on the time for each adjustment step in the locking process;

   Step_Time = (N / f_{ref}) + (SETTLING_COUNT / f_{ref})

   Multiply this number by the number of steps it takes to lock, to determine lock time. Typically, the FLL locks within the first ~10 steps.

21.4.2.2 Enabling and Disabling the FLL

The FLL requires firmware sequencing when enabling, disabling, and entering/exiting DEEPSLEEP.

To enable the FLL, follow these steps:

1. Enable the CCO by writing CLK_FLL_CONFIG4.CCO_ENABLE = 1 and wait until CLK_FLL_STATUS.CCO_READY==1.
2. Ensure the reference clock has stabilized and CLK_FLL_CONFIG3.BYPASS_SEL = FLL_REF.
3. Write FLL_ENABLE = 1 and wait until CLK_FLL_STATUS.LOCKED==1.
4. Write CLK_FLL_CONFIG3.BYPASS_SEL = FLL_OUT to switch to the FLL output.

To disable the FLL, follow these steps:

1. Ensure the processor is operating from a different clock than clk_path0. If the muxes are changed, wait four FLL output clock cycles for it to complete.
2. Write CLK_FLL_CONFIG3.BYPASS_SEL = FLL_REF and read the same register to ensure the write completes.
3. Wait at least six FLL reference clock cycles and disable it with FLL_ENABLE = 0.
4. Disable the CCO by writing CLK_FLL_CONFIG4.CCO_ENABLE = 0.

Before entering DEEPSLEEP, either disable the FLL using the above sequence or use the following procedure to deselect/select it before/after DEEPSLEEP. Before entering DEEPSLEEP, write CLK_FLL_CONFIG3.BYPASS_SEL = FLL_REF to change the FLL to use its reference clock. After DEEPSLEEP wakeup, wait until CLK_FLL_STATUS.LOCKED==1 and then write CLK_FLL_CONFIG3.BYPASS_SEL = FLL_OUT to switch to the FLL output.

Note: It is not recommended to use the FLL_AUTO option in the BYPASS_SEL field.

21.5 Clock Trees

The PSoC 6 MCU clocks are distributed throughout the device, as shown in Figure 21-1. The clock trees are described in this section:

- Path Clocks
- High-Frequency Root Clocks (CLK_HF[i])
- Low-Frequency Clock (CLK_LF)
- Timer Clock (CLK_TIMER)
- Analog Pump Clock (CLK_PUMP)

21.5.1 Path Clocks

The PSoC 6 MCU has six clock paths: CLK_PATH0 contains the FLL, CLK_PATH1 and CLK_PATH2 contain the PLLs, and CLK_PATH3, CLK_PATH4, and CLK_PATH5 are a direct connection to the high-frequency root clocks. Note that the FLL and PLL(s) can be bypassed if they are not needed. These paths are the input sources for the high-frequency clock roots.

Each clock path has a mux to determine which source clock will clock that path. This configuration is done in CLK_PATH_SELECT[i] register.

Table 21-9. Clock Path Source Selections

Name	Description
PATH_MUX[2:0]	Selects the source for clk_path[i] 0: IMO 1: EXTCLK 2: ECO 4: DSI_MUX 3: Reserved

The DSI mux is configured through the CLK_DSI_SELECT[i] register.

Table 21-10. DSI Mux Source Selection

Name	Description
DSI_MUX[4:0]	Selects the source for the DSI_MUX[i] 0-15: Reserved 16: ILO 17: WCO 18: Reserved 19: Reserved 20-31: Invalid

21.5.2 High-Frequency Root Clocks

The PSoC 6 MCU has six high-frequency root clocks (CLK_HF[i]). Each CLK_HF has a particular destination on the device, as shown in Table 21-11.

Table 21-11. CLK_HF Destinations

Name	Description
CLK_HF[0]	Root clock for both CPUs, PERI, and AHB infrastructure
CLK_HF[1]	Root clock for the PDM/PCM and I^2S audio subsystem
CLK_HF[2]	Root clock for the Serial Memory Interface subsystem and SDHC[1] block
CLK_HF[3]	Root clock for USB communications
CLK_HF[4]	Root clock for SDHC[0] block
CLK_HF[5]	Clock output on clk_ext pin (when used as an output)

Each high-frequency root clock has a mux to determine its source. This configuration is done in the CLK_ROOT_SELECT[i] register.

Table 21-12. HFCLK Input Selection Bits

Name	Description
ROOT_MUX[3:0]	HFCLK input clock selection 0: Select CLK_PATH0 1: Select CLK_PATH1 2: Select CLK_PATH2 3: Select CLK_PATH3 4: Select CLK_PATH4 5: Select CLK_PATH5

Each CLK_HF has a pre-divider, which is set in the CLK_ROOT_SELECT register.

Table 21-13. HFCLK Divider Selection

Name	Description
ROOT_DVI[5:4]	Selects predivider value for the clock root 0: No Divider 1: Divide clock by 2 2: Divide clock by 4 3: Divide clock by 8

CLK_HF[1-5] can be enabled and disabled. CLK_HF[0] is always enabled as it is the source of the CPU. To enable and disable CLK_HF[1-5] set the ENABLE bit in the CLK_ROOT_SELECT register.

21.5.3 Low-Frequency Clock

The low-frequency clock (CLK_LF) in the PSoC 6 MCU has two input options: ILO and WCO.

CLK_LF is the source for the multi-counter watchdog timers (MCWDT) and the RTC.

The source of CLK_LF is set in the LFCLK_SEL bits of the CLK_SELECT register.

Table 21-14. LFCLK Input Selection Bits LFCLK_SEL

Name	Description
LFCLK_SEL[1:0]	LFCLK input clock selection 0: ILO. Uses the internal local oscillator as the source of the LFCLK 1: WCO. Uses the watch crystal oscillator as the source of the LFCLK 2: Reserved 3: Reserved

21.5.4 Timer Clock

The timer clock (CLK_TIMER) can be used as a clock source for the profiler or the CPU SYSTICK timer. The source for CLK_TIMER can either be the IMO or CLK_HF[0]. This selection is made in the TIMER_SEL bitfield of the CLK_TIMER_CTL register. Several dividers can be applied to this clock, which are found in the CLK_TIMER_CTL register.

21.5.5 Group Clocks (clk_sys)

On the PSoC 6 platform, peripherals are grouped. Each group has a dedicated group clock (also referred to as clk_sys). The group clock sets the clock rate for the AHB interface on the peripheral; it also sets the clock rate for the trigger outputs and trigger input synchronization. Each group clock has an eight-bit divider located in the CLOCK_CTL register in the PERI_GROUP_STRUCT in the PERI register set. For a majority of applications these dividers should be left at default (divide by 1).

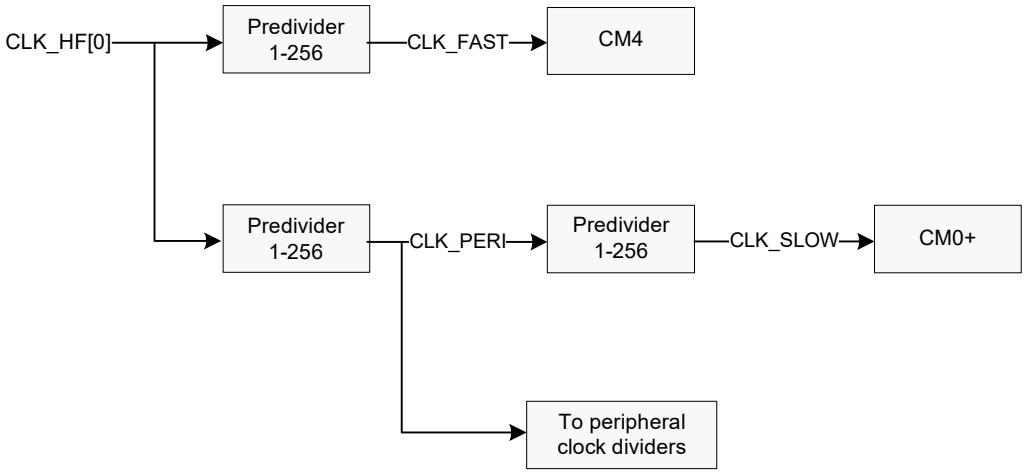
21.5.6 Backup Clock (clk_bak)

The backup clock is used to clock the backup domain, specifically the RTC. For more information see WCO with External Clock/Sine Wave Input on page 237.

21.6 CLK_HF[0] Distribution

clk_hf[0] is the root clock for the CPU subsystem and for the peripheral clock dividers.

Figure 21-4. CLK_HF[0] Distribution



21.6.1 CLK_FAST

CLK_FAST clocks the Cortex-M4 processor. This clock is a divided version of CLK_HF[0]. The divider for this clock is set in the CM4_CLOCK_CTL register of the CPU subsystem.

21.6.2 CLK_PERI

CLK_PERI is the source clock for all programmable peripheral clock dividers and for the Cortex-M0+ processor. It is a divided version of CLK_HF[0]. The divider for this clock is set in the PERI_INT_DIV bitfields of the CM0_CLOCK_CTL register.

21.6.3 CLK_SLOW

CLK_SLOW is the source clock for the Cortex-M0+. This clock is a divided version of CLK_PERI. The divider for this clock is set in the SLOW_INT_DIV bitfields of the CM0_CLOCK_CTL register.

21.7 Peripheral Clock Dividers

The PSoC 6 MCU peripherals such as SCBs and TCPWMs require a clock. These peripherals can be clocked only by a peripheral clock divider.

The PSoC 6 MCU has 29 peripheral clock dividers (PCLK). It has eight 8-bit dividers, sixteen 16-bit dividers, four fractional 16.5-bit dividers (16 integer bits, five fractional bits), and one 24.5-bit divider (24 integer bits, five fractional bits). The output of any of these dividers can be routed to any peripheral.

21.7.1 Fractional Clock Dividers

Fractional clock dividers allow the clock divisor to include a fraction of 0..31/32. For example, a 16.5-bit divider with an integer divide value of 3 generates a 16-MHz clock from a 48-MHz CLK_PERI. A 16.5-bit divider with an integer divide value of 4 generates a 12-MHz clock from a 48-MHz CLK_PERI. A 16.5-bit divider with an integer divide value of 3 and a fractional divider of 16 generates a 48 / (3 + 16/32) = 48 / 3.5 = 13.7-MHz clock from a 48-MHz CLK_PERI. Not all 13.7-MHz clock periods are equal in size; some will have a 16-MHz period and others will have a 12-MHz period, such that the average is 13.7 MHz.

Fractional dividers are useful when a high-precision clock is required (for example, for a UART/SPI serial interface). Fractional dividers are not used when a low jitter clock is required, because the clock periods have a jitter of one CLK_PERI cycle.

21.7.2 Peripheral Clock Divider Configuration

The peripheral clock dividers are configured using registers from the peripheral block; specifically DIV_CMD, DIV_8_CTL, DIV_16_CTL, DIV_16_5_CTL, DIV_24_5_CTL, and CLOCK_CTL registers.

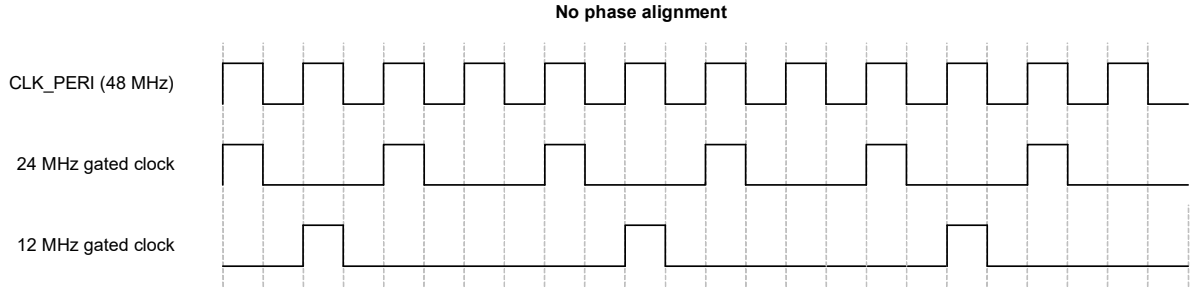
First the clock divider needs to be configured. This is done via the DIV_8_CTL, DIV_16_CTL, DIV_16_5_CTL, and DIV_24_5_CTL registers. There is one register for each divider; for example, there are eight DIV_8_CTL registers as there are eight 8-bit dividers. In these registers, set the value of the integer divider; if it is a fractional divider then set the fraction portion as well.

After the divider is configured use the DIV_CMD register to enable the divider. This is done by setting the DIV_SEL to the divider number you want to enable, and setting the TYPE_SEL to the divider type. For example, if you wanted to enable the 0th 16.5-bit divider, write ‘0’ to DIV_SEL and ‘2’ to TYPE_SEL. If you wanted to enable the tenth 16-bit divider, write ‘10’ to DIV_SEL and ‘1’ to TYPE_SEL. See the registers TRM for more details.

21.7.2.1 Phase Aligning Dividers

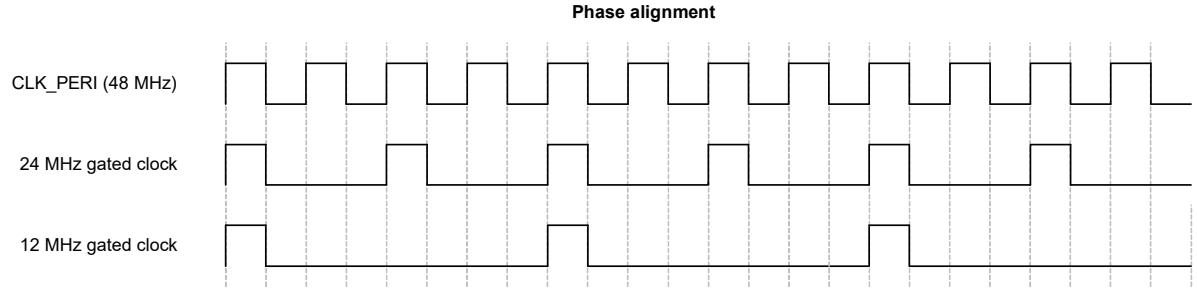
For specific use cases, you must generate clocks that are phase-aligned. For example, consider the generation of two gated clocks at 24 and 12 MHz, both of which are derived from a 48-MHz CLK_PERI. If phase alignment is not considered, the generated gated clocks appear as follows.

Figure 21-5. Non Phase-Aligned Clock Dividers



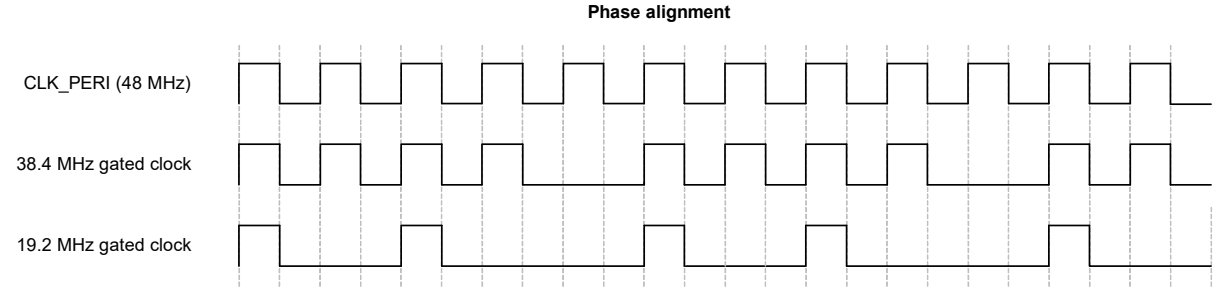
These clock signals may or may not be acceptable, depending on the logic functionality implemented on these two clocks. If the two clock domains communicate with each other, and the slower clock domain (12 MHz) assumes that each high/‘1’ pulse on its clock coincides with a high/‘1’ phase pulse in the higher clock domain (24 MHz), the phase misalignment is not acceptable. To address this, it is possible to have dividers produce clock signals that are phase-aligned with any of the other (enabled) clock dividers. Therefore, if (enabled) divider x is used to generate the 24-MHz clock, divider y can be phase-aligned to divider x and used to generate the 12-MHz clock. The aligned clocks appear as follows.

Figure 21-6. Phase-Aligned Clock Dividers



Phase alignment also works for fractional divider values. If (enabled) divider x is used to generate the 38.4-MHz clock (divide by 1 8/32), divider y can be phase-aligned to divider x and used to generate the 19.2-MHz clock (divide by 2 16/32). The generated gated clocks appear as follows.

Figure 21-7. Phase-Aligned Fractional Dividers



Divider phase alignment requires that the divider to which it is phase-aligned is already enabled. This requires the dividers to be enabled in a specific order.

Phase alignment is implemented by controlling the start moment of the divider counters in hardware. When a divider is enabled, the divider counters are set to '0'. The divider counters will only start incrementing from '0' to the programmed integer and fractional divider values when the divider to which it is phase-aligned has an integer counter value of '0'.

Note that the divider and clock multiplexer control register fields are all retained during the Deep Sleep power mode. However, the divider counters that are used to implement the integer and fractional clock dividers are not. These counters are set to '0' during the Deep Sleep mode. Therefore, when transitioning from Deep Sleep to Active mode, all dividers (and clock signals) are enabled and phase-aligned by design.

Phase alignment is accomplished by setting the PA_DIV_SEL and PA_DIV_TYPE bits in the DIV_CMD register before enabling the clock. For example, to align the fourth 8-bit divider to the third 16-bit divider, set DIV_SEL to '4', TYPE_SEL to '0', PA_DIV_SEL to '3', and PA_TYPE_SEL to '1'.

21.7.2.2 Connecting Dividers to Peripheral

The PSoC 6 MCU has 54 peripherals, which can connect to one of the programmable dividers. Table 21-15 lists those peripherals.

Table 21-15. Clock Dividers to Peripherals

Clock Number	Destination
0	scb[0].clock
1	scb[1].clock
2	scb[2].clock
3	scb[3].clock
4	scb[4].clock
5	scb[5].clock
6	scb[6].clock
7	scb[7].clock
8	scb[8].clock
9	scb[9].clock
10	scb[10].clock
11	scb[11].clock
12	scb[12].clock
13	smartio[8].clock
14	smartio[9].clock
15	tcpwm[0].clocks[0]
16	tcpwm[0].clocks[1]
17	tcpwm[0].clocks[2]
18	tcpwm[0].clocks[3]
19	tcpwm[0].clocks[4]
20	tcpwm[0].clocks[5]
21	tcpwm[0].clocks[6]
22	tcpwm[0].clocks[7]
23	tcpwm[1].clocks[0]
24	tcpwm[1].clocks[1]
25	tcpwm[1].clocks[2]
26	tcpwm[1].clocks[3]
27	tcpwm[1].clocks[4]
28	tcpwm[1].clocks[5]
29	tcpwm[1].clocks[6]
30	tcpwm[1].clocks[7]
31	tcpwm[1].clocks[8]
32	tcpwm[1].clocks[9]
33	tcpwm[1].clocks[10]
34	tcpwm[1].clocks[11]
35	tcpwm[1].clocks[12]
36	tcpwm[1].clocks[13]
37	tcpwm[1].clocks[14]
38	tcpwm[1].clocks[15]
39	tcpwm[1].clocks[16]
40	tcpwm[1].clocks[17]
41	tcpwm[1].clocks[18]
42	tcpwm[1].clocks[19]
43	tcpwm[1].clocks[20]
44	tcpwm[1].clocks[21]
45	tcpwm[1].clocks[22]
46	tcpwm[1].clocks[23]
47	csd.clock
48	lcd.clock
49	profile.clock_profile
50	cpuss.clock_trace_in
51	pass.clock_pump_peri
52	pass.clock_sar
53	usb.clock_dev_brs

To connect a peripheral to a specific divider, the PERI_CLOCK_CTL register is used. There is one PERI_CLOCK_CTL register for each entry in Table 21-15. For example, to select the twelfth 16-bit divider for tcpwm[0].clocks[2] write to the twenty-fifth CLOCK_CTL register, set the DIV_SEL to '12', and the TYPE_SEL to '1'.

21.8 Clock Calibration Counters

A feature of the clocking system in PSoC 6 MCUs is built-in hardware calibration counters. These counters can be used to compare the frequency of two clock sources against one another. The primary use case is to take a higher accuracy clock such as the ECO and use it to measure a lower accuracy clock such as the ILO. The result of this measurement can then be used to trim the ILO.

There are two counters: Calibration Counter 1 is clocked off of Calibration Clock 1 (generally the high-accuracy clock) and it counts down; Calibration Counter 2 is clocked off of Calibration Clock 2 and it counts up. When Calibration Counter 1 reaches 0, Calibration Counter 2 stops counting up and its value can be read. From that value the frequency of Calibration Clock 2 can be determined with the following equation.

$$\text{Calibration Clock 2 Frequency} = \frac{\text{Counter 2 Final Value}}{\text{Counter 1 Initial Value}} \times \text{Calibration Clock 1 Frequency}$$

For example, if Calibration Clock 1 = 8 MHz, Counter 1 = 1000, and Counter 2 = 5

Calibration Clock 1 Frequency = (5/1000) * 8 MHz = 40 kHz.

Calibration Clock 1 and Calibration Clock 2 are selected with the CLK_OUTPUT_FAST register. All clock sources are available as a source for these two clocks. CLK_OUTPUT_SLOW is also used to select the clock source.

Calibration Counter 1 is programmed in CLK_CAL_CNT1. Calibration Counter 2 can be read in CLK_CAL_CNT2.

When Calibration Counter 1 reaches 0, the CAL_COUNTER_DONE bit is set in the CLK_CAL_CNT1 register.

22. Reset System

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU family supports several types of resets that guarantee error-free operation during power up and allow the device to reset based on user-supplied external hardware or internal software reset signals. The PSoC 6 MCU also contains hardware to enable the detection of certain resets.

22.1 Features

The PSoC 6 MCU has these reset sources:

- Power-on reset (POR) to hold the device in reset while the power supply ramps up to the level required for the device to function properly
- Brownout reset (BOD) to reset the device if the power supply falls below the device specifications during normal operation
- External reset (XRES) to reset the device using an external input
- Watchdog timer (WDT) reset to reset the device if the firmware execution fails to periodically service the watchdog timer
- Software initiated reset to reset the device on demand using firmware
- Logic-protection fault resets to reset the device if unauthorized operating conditions occur
- Clock-supervision logic resets to reset the device when clock-related errors occur
- Hibernate wakeup reset to bring the device out of the Hibernate low-power mode

22.2 Architecture

The following sections provide a description of the reset sources available in the PSoC 6 MCU family.

Note: None of these sources can reset the Backup system. The Backup domain is reset only when all the power supplies are removed from it, also known as a “cold start” or if the firmware triggers a reset using the BACKUP_RESET register. Firmware reset is required if the VBACKUP supply was invalid during a previous power supply ramp-up or brownout event. For more details, see the Backup System chapter on page 235.

Table 22-1 lists all the reset sources in the PSoC 6 MCU.

Table 22-1. PSoC 6 MCU Reset Sources

Reset Source	Reset Condition	Availability in System Power Modes	Cause Detection
Power-on Reset	This reset condition occurs during device power-up. POR holds the device under reset until the V_{DDD} reaches the threshold voltage as specified in the PSoC 61 datasheet/PSoC 62 datasheet.	All	Not inferable using the reset cause registers.
Brownout Reset	V_{DDD} falls below the minimum logic operating voltage specified in the PSoC 61 datasheet/PSoC 62 datasheet.	All	Not inferable using the reset cause registers.
Watchdog Timer Reset	WDT is not reset by firmware within the configured reset time period of the WDT.	All	RESET_HWWDT bit or RESET_SWWDT0 to RESET_SWWDT3 status bits of the RES_CAUSE register is set when a watchdog reset occurs. This bit remains set until cleared by the firmware or until a POR, XRES, or BOD reset occurs. All other resets leave this bit unaltered.
Hibernate Wakeup	System wakes up from Hibernate power mode.	Hibernate	Value written to the TOKEN bit field of the PWR_HIBERNATE register is retained after reset.
Software Reset	SYSRESETREQ bit is set by the firmware in the CM0_AIRCR/CM4_AIRCR register.	Active	RESET_SOFT bit of the RES_CAUSE register is set
External Reset	Logic LOW input to XRES pin	All	Not inferable using the reset cause registers.
Logic Protection Fault Reset	Unauthorized protection violation	Active, Deep Sleep modes	RESET_ACT_FAULT or RESET_DPSLP_FAULT bits of the RES_CAUSE register is set.
Clock-Supervision Logic Reset	Loss of a high-frequency clock or watch-crystal clock, or due to a high-frequency clock error.	Active, Deep Sleep modes	RESET_CSV_WCO_LOSS bit of the RES_CAUSE register is set when WCO clock is lost. The RESET_CSV_HF_LOSS of RES_CAUSE2 register can be used to identify resets caused by the loss of a high-frequency clock. RESET_CSV_HF_FREQ field can be used to identify resets caused by the frequency error of a high-frequency clock.

22.2.1 Power-on Reset

Power-on reset is provided to keep the system in a reset state during power-up. POR holds the device in reset until the supply voltage, V_{DDD} reaches the datasheet specification. The POR activates automatically at power-up. Refer to thePSoC 61 datasheet/PSoC 62 datasheet for details on the POR trip-point levels.

POR events do not set a reset cause status bit, but can be partially inferred by the absence of any other reset source. If no other reset event is detected, then the reset is caused by POR, BOD, or XRES.

22.2.2 Brownout Reset

Brownout reset monitors the chip digital voltage supply V_{DDD} and generates a reset if V_{DDD} falls below the minimum logic operating voltage specified in the PSoC 61 datasheet/PSoC 62 datasheet. See the Power Supply and Monitoring chapter on page 218 for more details.

BOD events do not set a reset cause status bit, but in some cases they can be detected. In some BOD events, V_{DDD} will fall below the minimum logic operating voltage specified by the datasheet, but remain above the minimum logic retention voltage.

22.2.3 Watchdog Timer Reset

Watchdog timer reset causes a reset if the WDT is not serviced by the firmware within a specified time limit. See the Watchdog Timer chapter on page 283 for more details.

The RESET_WDT bit or RESET_MCWDT0 to RESET_MCWDT3 status bits of the RES_CAUSE register is set when a watchdog reset occurs. This bit remains set

until cleared by the firmware or until a POR, XRES, or BOD reset occurs. All other resets leave this bit unaltered.

For more details, see the Watchdog Timer chapter on page 283.

22.2.4 Software Initiated Reset

Software initiated reset is a mechanism that allows the CPU to request a reset. The Cortex-M0+ and Cortex-M4 Application Interrupt and Reset Control registers (CM0_AIRCR and CM4_AIRCR, respectively) can request a reset by writing a '1' to the SYSRESETREQ bit of the respective registers.

Note that a value of 0x5FA should be written to the VECTKEY field of the AIRCR register before setting the SYSRESETREQ bit; otherwise, the processor ignores the write. See the CPU Subsystem (CPUSS) chapter on page 32 and Arm documentation on AIRCR for more details.

The RESET_SOFT status bit of the RES_CAUSE register is set when a software reset occurs. This bit remains set until cleared by firmware or until a POR, XRES, or BOD reset occurs. All other resets leave this bit unaltered.

22.2.5 External Reset

External reset (XRES) is a reset triggered by an external signal that causes immediate system reset when asserted. The XRES pin is active low – a logic '1' on the pin has no effect and a logic '0' causes reset. The pin is pulled to logic '1' inside the device. XRES is available as a dedicated pin. For detailed pinout, refer to the pinout section of the PSoC 61 datasheet/PSoC 62 datasheet.

The XRES pin holds the device in reset as long as the pin input is '0'. When the pin is released (changed to logic '1'), the device goes through a normal boot sequence. The logical thresholds for XRES and other electrical characteristics are listed in the Electrical Specifications section of the PSoC 61 datasheet/PSoC 62 datasheet. XRES is available in all power modes, but cannot reset the Backup system.

An XRES event does not set a reset cause status bit, but can be partially inferred by the absence of any other reset source. If no other reset event is detected, then the reset is caused by POR, BOD, or XRES.

22.2.6 Logic Protection Fault Reset

Logic protection fault reset detects any unauthorized protection violations and causes the device to reset if they occur. One example of a protection fault is reaching a debug breakpoint while executing privileged code.

The RESET_ACT_FAULT or RESET_DPSLP_FAULT bits of the RES_CAUSE register is set when a protection fault occurs in Active or Deep Sleep modes, respectively. These bits remain set until cleared or until a POR, XRES, or BOD reset. All other resets leave this bit unaltered.

22.2.7 Clock-Supervision Logic Reset

Clock-supervision logic initiates a reset due to the loss of a high-frequency clock or watch-crystal clock, or due to a high-frequency clock error.

The RESET_CSV_WCO_LOSS bit of the RES_CAUSE register is set when the clock supervision logic requests a reset due to the loss of a watch-crystal clock (if enabled).

The RESET_CSV_HF_LOSS is a 16-bit field in the RES_CAUSE2 register that can be used to identify resets caused by the loss of a high-frequency clock. Similarly, the RESET_CSV_HF_FREQ field can be used to identify resets caused by the frequency error of a high-frequency clock.

For more information on clocks, see the Clocking System chapter on page 242.

22.2.8 Hibernate Wakeup Reset

Hibernate wakeup reset occurs when one of the Hibernate wakeup sources performs a device reset to return to the Active power mode. See the Device Power Modes chapter on page 225 for details on Hibernate mode and available wakeup sources.

TOKEN is an 8-bit field in the PWR_HIBERNATE register that is retained through a Hibernate wakeup sequence. The firmware can use this bitfield to differentiate hibernate wakeup from a general reset event. Similarly, the PWR_HIB_DATA register can retain its contents through a Hibernate wakeup reset, but is cleared when XRES is asserted.

22.3 Identifying Reset Sources

When the device comes out of reset, it is often useful to know the cause of the most recent or even older resets. This is achieved through the RES_CAUSE and RES_CAUSE2 registers. These registers have specific status bits allocated for some of the reset sources. These registers record the occurrences of WDT reset, software reset, logic-protection fault, and clock-supervision resets. However, these registers do not record the occurrences of POR, BOD, XRES, or Hibernate wakeup resets. The bits in these registers are set on the occurrence of the corresponding reset and remain

set after the reset, until cleared by the firmware or a loss of retention, such as a POR, XRES, or BOD.

Hibernate wakeup resets can be detected by examining the TOKEN field in the PWR_HIBERNATE register as described previously. Hibernate wakeup resets that occur as a result of an XRES cannot be detected. The other reset sources can be inferred to some extent by the status of the RES_CAUSE and RES_CAUSE2 registers, as shown in Table 22-2.

Table 22-2. Reset Cause Bits to Detect Reset Source

Register	Bitfield	Number of Bits	Description
RES_CAUSE	RESET_WDT	1	A hardware WDT reset has occurred since the last power cycle.
RES_CAUSE	RESET_ACT_FAULT	1	Fault logging system requested a reset from its Active logic.
RES_CAUSE	RESET_DPSLP_FAULT	1	Fault logging system requested a reset from its Deep Sleep logic.
RES_CAUSE	RESET_CSV_WCO_LOSS	1	Clock supervision logic requested a reset due to loss of a watch-crystal clock.
RES_CAUSE	RESET_SOFT	1	A CPU requested a system reset through its SYSRESETREQ. This can be done via a debugger probe or in firmware.
RES_CAUSE	RESET_MCWDT0	1	Multi-counter WDT reset #0 has occurred since the last power cycle.
RES_CAUSE	RESET_MCWDT1	1	Multi-counter WDT reset #1 has occurred since the last power cycle.
RES_CAUSE	RESET_MCWDT2	1	Multi-counter WDT reset #2 has occurred since the last power cycle.
RES_CAUSE	RESET_MCWDT3	1	Multi-counter WDT reset #3 has occurred since the last power cycle.
RES_CAUSE2	RESET_CSV_HF_LOSS	16	Clock supervision logic requested a reset due to loss of a high-frequency clock. Each bit index K corresponds to a clk_hf<K>. Unimplemented clock bits return zero.
RES_CAUSE2	RESET_CSV_HF_FREQ	16	Clock supervision logic requested a reset due to frequency error of a high-frequency clock. Each bit index K corresponds to a clk_hf<K>. Unimplemented clock bits return zero.

For more information, see the RES_CAUSE and RES_CAUSE2 registers in the registers TRM.

If these methods cannot detect the cause of the reset, then it can be one of the non-recorded and non-retention resets: BOD, POR, or XRES. These resets cannot be distinguished using on-chip resources.

22.4 Register List

Table 22-3. Reset System Register List

Register	Description
RES_CAUSE	Reset cause observation register
RES_CAUSE2	Reset cause observation register 2
PWR_HIBERNATE	Hibernate power mode control register. Contains a TOKEN field that can be used to detect the Hibernate wakeup reset
PWR_HIB_DATA	Retains its contents through Hibernate wakeup reset
CM4_AIRCR	Application interrupt and reset control register of Cortex-M4
CM0_AIRCR	Application interrupt and reset control register of Cortex-M0+

23. I/O System

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

This chapter explains the PSoC 6 MCU I/O system, its features, architecture, operating modes, and interrupts. The I/O system provides the interface between the CPU core and peripheral components to the outside world. The flexibility of PSoC 6 MCUs and the capability of its I/O to route most signals to most pins greatly simplifies circuit design and board layout. The GPIO pins in the PSoC 6 MCU family are grouped into ports; a port can have a maximum of eight GPIO pins.

23.1 Features

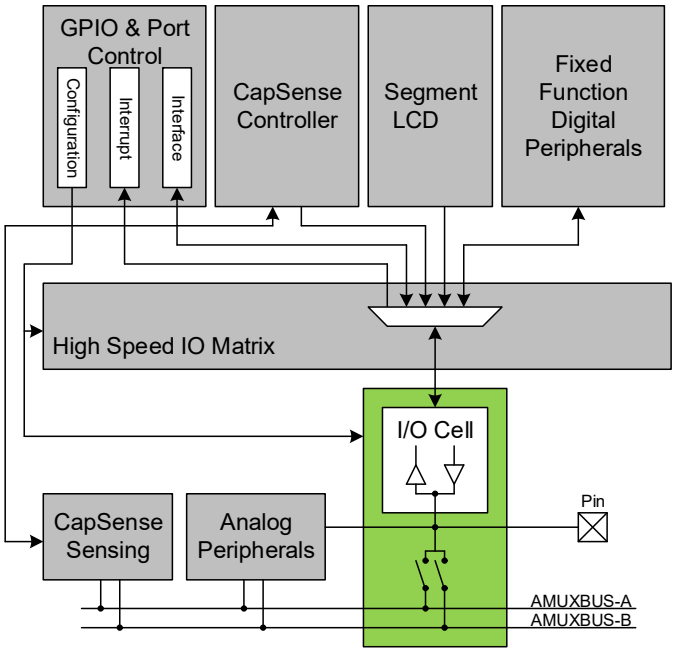
The PSoC 6 MCU GPIOs have these features:

- Analog and digital input and output capabilities
- Eight drive strength modes
- Separate port read and write registers
- Overvoltage tolerant (OVT-GPIO) pins
- Separate I/O supplies and voltages for up to six groups of I/O
- Edge-triggered interrupts on rising edge, falling edge, or on both edges, on all GPIO
- Slew rate control
- Frozen mode for latching previous state (used to retain the I/O state in System Hibernate Power mode)
- Selectable CMOS and low-voltage LVTTL input buffer mode
- CapSense support
- Smart I/O provides the ability to perform Boolean functions in the I/O signal path
- Segment LCD drive support

23.2 Architecture

The PSoC 6 MCU is equipped with analog and digital peripherals. Figure 23-1 shows an overview of the routing between the peripherals and pins.

Figure 23-1. GPIO Interface Overview

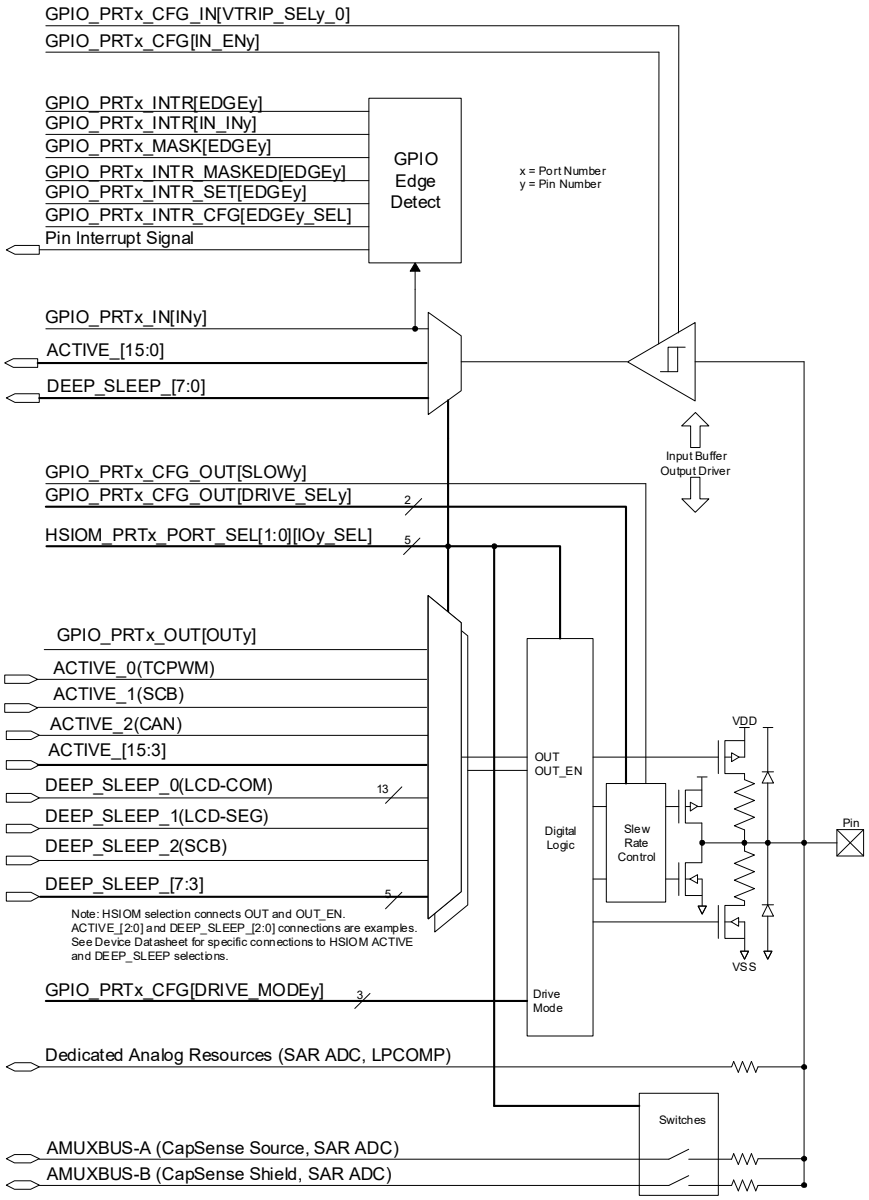


GPIO pins are connected to I/O cells. These cells are equipped with an input buffer for the digital input, providing high input impedance and a driver for the digital output signals. The digital peripherals connect to the I/O cells via the high-speed I/O matrix (HSIOM). The HSIOM for each pin contains multiplexers to connect between the selected peripheral and the pin. Analog peripherals such as SAR ADC,Low-Power comparator (LPCOMP), and CapSense are either connected to the GPIO pins directly or through the AMUXBUS.

23.2.1 I/O Cell Architecture

Figure 23-2 shows the I/O cell architecture present in every GPIO cell. It comprises an input buffer and an output driver that connect to the HSIOM multiplexers for digital input and output signals. Analog peripherals connect directly to the pin for point to point connections or use the AMUXBUS.

Figure 23-2. GPIO and GPIO_OVT Cell Architecture



23.2.2 Digital Input Buffer

The digital input buffer provides a high-impedance buffer for the external digital input. The buffer is enabled or disabled by the IN_EN[7:0] bit of the Port Configuration Register (GPIO_PRTx_CFG, where x is the port number).

The input buffer is connected to the HSIOM for routing to the CPU port registers and selected peripherals. Writing to the HSIOM port select register (HSIOM_PORT_SELx) selects the pin connection. See the PSoC 61 datasheet/PSoC 62 datasheet for the specific connections available for each pin.

If a pin is connected only to an analog signal, the input buffer should be disabled to avoid crowbar currents.

Each pin's input buffer trip point and hysteresis are configurable for the following modes:

- CMOS + I^2C
- TTL

These buffer modes are selected by the VTRIP_SEL[7:0]_0 bit of the Port Input Buffer Configuration register. (GPIO_PRTx_CFG_IN).

23.2.3 Digital Output Driver

Pins are driven by the digital output driver. It consists of circuitry to implement different drive modes and slew rate control for the digital output signals. The HSIOM selects the control source for the output driver. The two primary types of control sources are CPU registers and fixed-function digital peripherals. A particular HSIOM connection is selected by writing to the HSIOM port select register (HSIOM_PORT_SELx).

I/O ports are powered by different sources. The specific allocation of ports to supply sources can be found in the Pinout section of the PSoC 61 datasheet/PSoC 62 datasheet.

Each GPIO pin has ESD diodes to clamp the pin voltage to the I/O supply source. Ensure that the voltage at the pin does not exceed the I/O supply voltage $V_{DDIO}/V_{DDD}/V_{DDA}$ or drop below $V_{SSIO}/V_{SSD}/V_{SSA}$. For the absolute maximum and minimum GPIO voltage, see the PSoC 61 datasheet/ PSoC 62 datasheet.

23.2.3.1 Drive Modes

Each I/O is individually configurable to one of eight drive modes by the DRIVE_MODE[7:0] field of the Port Configuration register, GPIO_PRTx_CFG. Table 23-1 lists the drive modes. Drive mode '1' is reserved and should not be used in most designs. CPU register and AMUXBUS connections support seven discrete drive modes to maximize design flexibility. Fixed-function digital peripherals, such as SCB and TCPWM blocks, support modified functionality for the same seven drive modes compatible with fixed peripheral signaling. Figure 23-3 shows simplified output driver diagrams of the pin view for CPU register control on each of the eight drive modes. Figure 23-4 is a simplified output driver diagram that shows the pin view for fixed-function-based peripherals for each of the eight drive modes.

Table 23-1. Drive Mode Settings

Drive Mode	Value	CPU Register, AMUXBUS				Fixed-Function Digital Peripheral			
		OUT_EN = 1		OUT_EN = 0		OUT_EN = 1		OUT_EN = 0	
		OUT = 1	OUT = 0	OUT = 1	OUT = 0	OUT = 1	OUT = 0	OUT = 1	OUT = 0
High Impedance	0	HI-Z	HI-Z	HI-Z	HI-Z	HI-Z	HI-Z	HI-Z	HI-Z
Reserved	1	Strong 1	Strong 0	HI-Z	HI-Z	Strong 1	Strong 0	Weak 1 & 0	Weak 1 & 0
Resistive Pull Up	2	Weak 1	Strong 0	HI-Z	HI-Z	Strong 1	Strong 0	Weak 1	Weak 1
Resistive Pull Down	3	Strong 1	Weak 0	HI-Z	HI-Z	Strong 1	Strong 0	Weak 0	Weak 0
Open Drain, Drives Low	4	HI-Z	Strong 0	HI-Z	HI-Z	Strong 1	Strong 0	HI-Z	HI-Z
Open Drain, Drives High	5	Strong 1	HI-Z	HI-Z	HI-Z	Strong 1	Strong 0	HI-Z	HI-Z
Strong	6	Strong 1	Strong 0	HI-Z	HI-Z	Strong 1	Strong 0	HI-Z	HI-Z
Resistive Pull Up and Down	7	Weak 1	Weak 0	HI-Z	HI-Z	Strong 1	Strong 0	Weak 1	Weak 0

Figure 23-3. CPU I/O Drive Mode Block Diagram

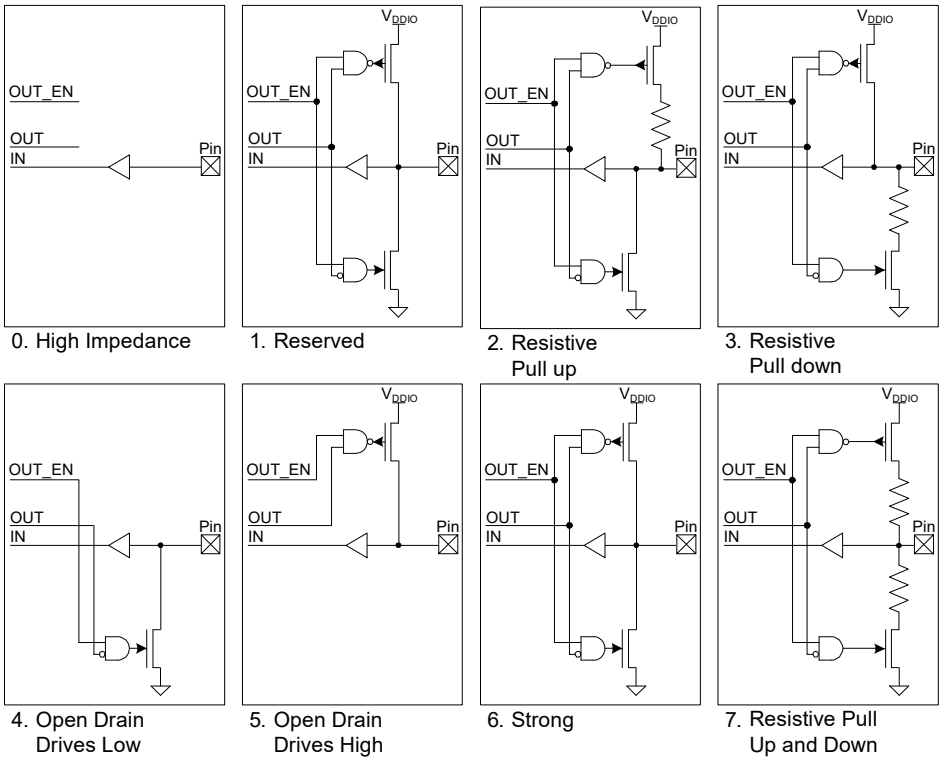
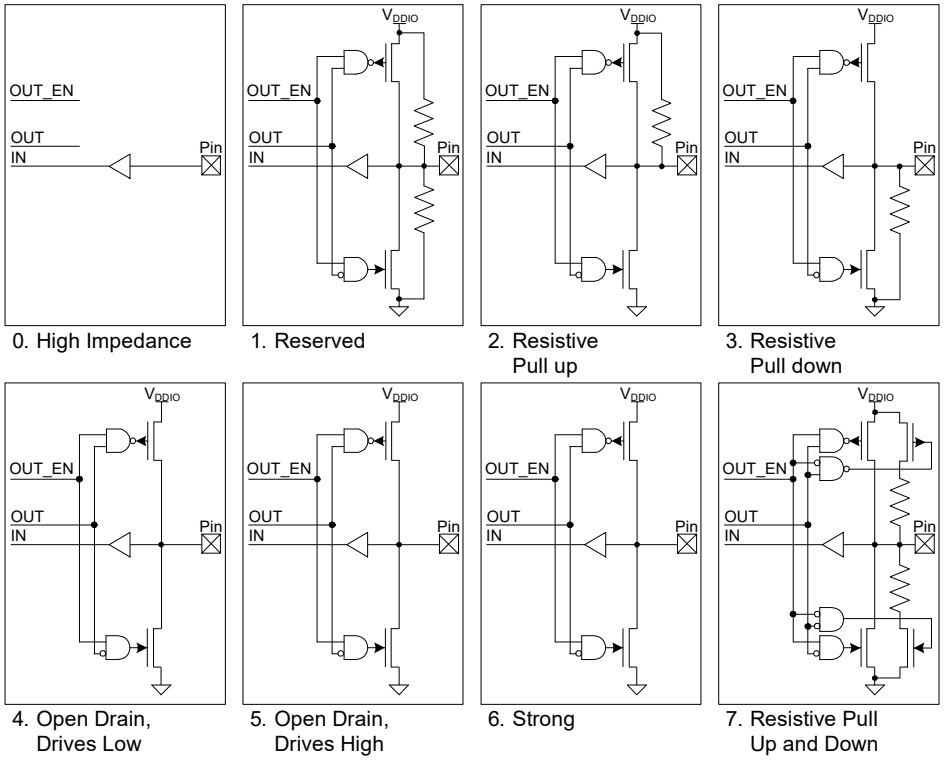


Figure 23-4. Peripheral I/O Drive Mode Block Diagrams



- High-Impedance

This is the standard high-impedance (HI-Z) state recommended for analog and digital inputs. For digital signals, the input buffer is enabled; for analog signals, the input buffer is typically disabled to reduce crowbar current and leakage in low-power designs. To achieve the lowest device current, unused GPIOs must be configured to the high-impedance drive mode with input buffer disabled. High-impedance drive mode with input buffer disabled is also the default pin reset state.

- Resistive Pull-Up or Resistive Pull-Down

Resistive modes provide a series resistance in one of the data states and strong drive in the other. Pins can be used for either digital input or digital output in these modes. If resistive pull-up is required, a '1' must be written to that pin's Data Register bit. If resistive pull-down is required, a '0' must be written to that pin's Data Register. Interfacing mechanical switches is a common application of these drive modes. The resistive modes are also used to interface PSoC with open drain drive lines. Resistive pull-up is used when the input is open drain low and resistive pull-down is used when the input is open drain high.

- Open Drain Drives High and Open Drain Drives Low

Open drain modes provide high impedance in one of the data states and strong drive in the other. Pins are useful as digital inputs or outputs in these modes. Therefore, these modes are widely used in bidirectional digital communication. Open drain drive high mode is used when the signal is externally pulled down and open drain drive low is used when the signal is externally pulled high. A common application for the open drain drives low mode is driving I^2C bus signal lines.

- Strong Drive

The strong drive mode is the standard digital output mode for pins; it provides a strong CMOS output drive in both high and low states. Strong drive mode pins should not be used as inputs under normal circumstances. This mode is often used for digital output signals or to drive external devices.

- Resistive Pull-Up and Resistive Pull-Down

In the resistive pull-up and pull-down mode, the GPIO will have a series resistance in both logic 1 and logic 0 output states. The high data state is pulled up while the low data state is pulled down. This mode is useful when the pin is driven by other signals that may cause shorts.

23.2.3.2 Slew Rate Control

GPIO pins have fast and slow output slew rate options for the strong drivers configured using the SLOW bit of the port output configuration register (GPIO_PRTx_CFG_OUT). By default, this bit is cleared and the port works in fast slew mode. This bit can be set if a slow slew rate is required. Slower slew rate results in reduced EMI and crosstalk and are recommended for low-frequency signals or signals without strict timing constraints.

When configured for fast slew rate, the drive strength can be set to one of four levels using the DRIVE_SEL field of the port output configuration register (GPIO_PRTx_CFG_OUT). The drive strength field determines the active portion of the output drivers used and can affect the slew rate of output signals. Drive strength options are full drive strength (default), one-half strength, one-quarter strength, and one-eighth strength. Drive strength must be set to full drive strength when the slow slew rate bit (SLOW) is set.

Note: For some devices in the PSoC 6 MCU family, simultaneous GPIO switching with unrestricted drive strengths and frequency can induce noise in on-chip subsystems affecting CapSense and ADC results. Refer to the Errata section in the respective device datasheet for details.

23.2.3.3 GPIO-OVT Pins

Select device pins are overvoltage tolerant (OVT) and are useful for interfacing to busses or other signals that may exceed the pin's V_{DDIO} supply, or where the whole device supply or pin V_{DDIO} may not be always present. They are identical to regular GPIOs with the additional feature of being overvoltage tolerant. GPIO-OVT pins have hardware to compare V_{DDIO} to the pin voltage. If the pin voltage exceeds V_{DDIO}, the output driver is disabled and the pin driver is tristated. This results in negligible current sink at the pin.

Note that in overvoltage conditions, the input buffer data will not be valid if the external source's specification of V_{OH} and V_{OL} do not match the trip points of the input buffer defined by the current V_{DDIO} voltage.

23.3 High-Speed I/O Matrix

The high-speed I/O matrix (HSIOM) is a set of high-speed multiplexers that route internal CPU and peripheral signals to and from GPIOs. HSIOM allows GPIOs to be shared with multiple functions and multiplexes the pin connection to a user-selected peripheral. The HSIOM_PRTx_PORT_SEL[1:0] registers allow a single selection from up to 32 different connections to each pin as listed in Table 23-2.

Table 23-2. HSIOM Connections

SELy_SEL	Name	Digital Driver Signal Source		Digital Input Signal Destination	Analog Switches		Description
		OUT	OUT_EN		AMUXA	AMUXB	
0	GPIO	OUT Register	1	IN Register	0	0	GPIO_PRTx_OUT register controls OUT
1	Reserved	–	–	–	–	–	–
2	Reserved	–	–	–	–	–	–
3	Reserved	–	–	–	–	–	–
4	AMUXA	OUT Register	1	IN Register	1	0	Analog mux bus A connected to pin
5	AMUXB	OUT Register	1	IN Register	0	1	Analog mux bus B connected to pin
6	Reserved	–	–	–	–	–	–
7	Reserved	–	–	–	–	–	–
8	ACT_0	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 0 - See the datasheet for specific pin connectivity
9	ACT_1	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 1 - See the datasheet for specific pin connectivity
10	ACT_2	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 2 - See the datasheet for specific pin connectivity
11	ACT_3	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 3 - See the datasheet for specific pin connectivity
12	DS_0	Deep Sleep Source OUT	Deep Sleep Source OUT-_EN	Deep Sleep IN	0	0	Deep Sleep functionality 0 - See the datasheet for specific pin connectivity
13	DS_1	Deep Sleep Source OUT	Deep Sleep Source OUT-_EN	Deep Sleep IN	0	0	Deep Sleep functionality 1 - See the datasheet for specific pin connectivity
14	DS_2	Deep Sleep Source OUT	Deep Sleep Source OUT-_EN	Deep Sleep IN	0	0	Deep Sleep functionality 2 - See the datasheet for specific pin connectivity
15	DS_3	Deep Sleep Source OUT	Deep Sleep Source OUT-_EN	Deep Sleep IN	0	0	Deep Sleep functionality 3 - See the datasheet for specific pin connectivity
16	ACT_4	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 4 - See the datasheet for specific pin connectivity
17	ACT_5	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 5 - See the datasheet for specific pin connectivity
18	ACT_6	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 6 - See the datasheet for specific pin connectivity
19	ACT_7	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 7 - See the datasheet for specific pin connectivity
20	ACT_8	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 8 - See the datasheet for specific pin connectivity
21	ACT_9	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 9 - See the datasheet for specific pin connectivity

Table 23-2. HSIOM Connections

SELy_SEL	Name	Digital Driver Signal Source		Digital Input Signal Destination	Analog Switches		Description
		OUT	OUT_EN		AMUXA	AMUXB	
22	ACT_10	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 10 - See the datasheet for specific pin connectivity
23	ACT_11	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 11 - See the datasheet for specific pin connectivity
24	ACT_12	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 12 - See the datasheet for specific pin connectivity
25	ACT_13	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 13 - See the datasheet for specific pin connectivity
26	ACT_14	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 14 - See the datasheet for specific pin connectivity
27	ACT_15	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 15 - See the datasheet for specific pin connectivity
28	DS_4	Deep Sleep Source OUT	Deep Sleep Source OUT-_EN	Deep Sleep IN	0	0	Deep Sleep functionality 4 - See the datasheet for specific pin connectivity
29	DS_5	Deep Sleep Source OUT	Deep Sleep Source OUT-_EN	Deep Sleep IN	0	0	Deep Sleep functionality 5 - See the datasheet for specific pin connectivity
30	DS_6	Deep Sleep Source OUT	Deep Sleep Source OUT-_EN	Deep Sleep IN	0	0	Deep Sleep functionality 6 - See the datasheet for specific pin connectivity
31	DS_7	Deep Sleep Source OUT	Deep Sleep Source OUT-_EN	Deep Sleep IN	0	0	Deep Sleep functionality 7 - See the datasheet for specific pin connectivity

Note: The Active and Deep Sleep sources are pin dependent. See the “Pinouts” section of the PSoC 61 datasheet/PSoC 62 datasheet for more details on the features supported by each pin.

23.4 I/O State on Power Up

During power up, all the GPIOs are in high-impedance analog state and the input buffers are disabled. During runtime, GPIOs can be configured by writing to the associated registers. Note that the pins supporting debug access port (DAP) connections (SWD lines) are always enabled as SWD lines during power up. The DAP connection does not provide pull-up or pull-down resistors; therefore, if left floating some crowbar current is possible. The DAP connection can be disabled or reconfigured for general-purpose use through the HSIOM only after the device boots and starts executing code.

23.5 Behavior in Low-Power Modes

Table 23-3 shows the status of GPIOs in low-power modes.

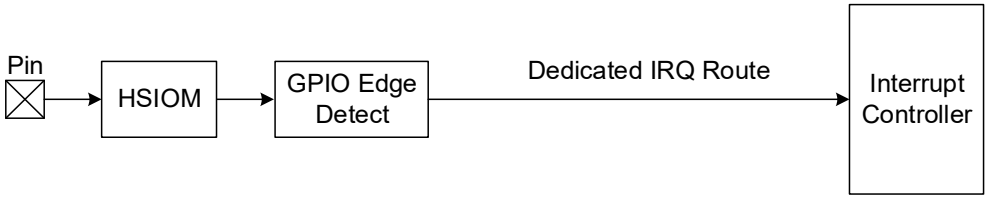
Table 23-3. GPIO in Low-Power Modes

Low-Power Mode	Status
CPU Sleep	■ Standard GPIO, GPIO-OVT, and SIO pins are active and can be driven by most peripherals such as CapSense, TCPWMs, and SCBs, which can operate in CPU Sleep mode. ■ Inputs buffers are active; thus an interrupt on any I/O can be used to wake the CPU.
System Deep Sleep	■ GPIO, GPIO-OVT, and SIO pins, connected to System Deep Sleep domain peripherals, are functional. All other pins maintain the last output driver state and configuration. ■ Pin interrupts are functional on all I/Os and can be used to wake the device.
System Hibernate	■ Pin output states and configuration are latched and remain in the frozen state. ■ Pin interrupts are functional only on select IOs and can be used to wake the device. See the PSoC 61 datasheet/PSoC 62 datasheet for specific hibernate pin connectivity.

23.6 Interrupt

All port pins have the capability to generate interrupts. Figure 23-5 shows the routing of pin signals to generate interrupts.

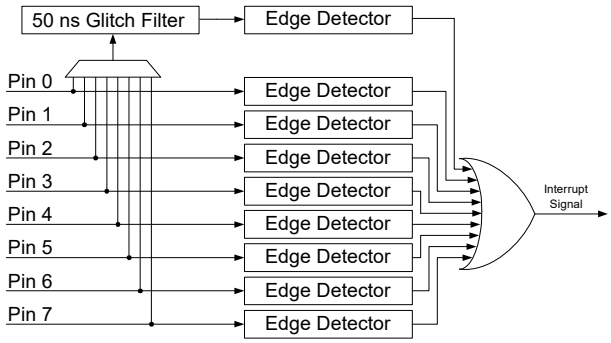
Figure 23-5. Interrupt Signal Routing



■ Pin signal through the “GPIO Edge Detect” block with direct connection to the CPU interrupt controller

Figure 23-6 shows the GPIO Edge Detect block architecture.

Figure 23-6. GPIO Edge Detect Block Architecture



An edge detector is present at each pin. It is capable of detecting rising edge, falling edge, and both edges without any reconfiguration. The edge detector is configured by writing into the EDGEv_SEL bits of the Port Interrupt Configuration register, GPIO_PRTx_INTR_CFG, as shown in Table 23-4.

Table 23-4. Edge Detector Configuration

EDGE_SEL	Configuration
00	Interrupt is disabled
01	Interrupt on Rising Edge
10	Interrupt on Falling Edge
11	Interrupt on Both Edges

Writing '1' to the corresponding status bit clears the pin edge state. Clearing the edge state status bit is important; otherwise, an interrupt can occur repeatedly for a single trigger or respond only once for multiple triggers, which is explained later in this section. When the Port Interrupt Control Status register is read at the same time an edge is occurring on the corresponding port, it can result in the edge not being properly detected. Therefore, when using GPIO interrupts, read the status register only inside the corresponding interrupt service routine and not in any other part of the code.

Firmware and the debug interface are able to trigger a hardware interrupt from any pin by setting the corresponding bit in the GPIO_PRTx_INTR_SET register.

In addition to the pins, each port provides a glitch filter connected to its own edge detector. This filter can be driven by one of the pins of a port. The selection of the driving pin is done by writing to the FLT_SEL field of the GPIO_PRTx_INTR_CFG register as shown in Table 23-5.

Table 23-5. Glitch Filter Input Selection

FLT_SEL	Selected Pin
000	Pin 0 is selected
001	Pin 1 is selected
010	Pin 2 is selected
011	Pin 3 is selected
100	Pin 4 is selected
101	Pin 5 is selected
110	Pin 6 is selected
111	Pin 7 is selected

When a port pin edge occurs, you can read the Port Interrupt Status register, GPIO_PRTx_INTR, to know which pin caused the edge. This register includes both the latched information on which pin detected an edge and the current pin status. This allows the CPU to read both information in a single read operation. This register has an additional use – to clear the latched edge state.

The GPIO_PRTx_INTR_MASK register enables forwarding of the GPIO_PRTx_INTR edge detect signal to the interrupt controller when a '1' is written to a pin's corresponding bitfield. The GPIO_PRTx_INTR_MASKED register can then be read to determine the specific pin that generated the interrupt signal forwarded to the interrupt controller. The masked edge detector outputs of a port are then ORed together and routed to the interrupt controller (NVIC in the CPU subsystem). Thus, there is only one interrupt vector per port.

The masked and ORed edge detector block output is routed to the Interrupt Source Multiplexer shown in Figure 8-3 on page 59, which gives an option of Level and Rising Edge detection. If the Level option is selected, an interrupt is triggered repeatedly as long as the Port Interrupt Status register bit is set. If the Rising Edge detect option is selected, an interrupt is triggered only once if the Port Interrupt Status register is not cleared. Thus, the interrupt status bit must be cleared if the Edge Detect block is used.

All of the port interrupt vectors are also ORed together into a single interrupt vector for use on devices with more ports than there are interrupt vectors available. To determine the port that triggered the interrupt, the GPIO_INTR_CAUSEx registers can be read. A '1' present in a bit location indicates that the corresponding port has a pending interrupt. The indicated GPIO_PRTx_INTR register can then be read to determine the pin source.

23.7 Peripheral Connections

23.7.1 Firmware-Controlled GPIO

For standard firmware-controlled GPIO using registers, the GPIO mode must be selected in the HSIOM_PORT_SELx register.

The GPIO_PRTx_OUT register is used to read and write the output buffer state for GPIOs. A write operation to this register changes the GPIO's output driver state to the written value. A read operation reflects the output data written to this register and the resulting output driver state. It does not return the current logic level present on GPIO pins, which may be different. Using the GPIO_PRTx_OUT register, read-modify-write sequences can be safely performed on a port that has both input and output GPIOs.

In addition to the data register, three other registers – GPIO_PRTx_SET, GPIO_PRTx_CLR, and GPIO_PRTx_INV – are provided to set, clear, and invert the output data respectively on specific pins in a port without affecting other pins. This avoids the need for read-modify-write operations in most use cases. Writing '1' to these register bitfields will set, clear, or invert the respective pin; writing '0' will have no affect on the pin state.

GPIO_PRTx_IN is the port I/O pad register, which provides the actual logic level present on the GPIO pin when read. Writes to this register have no effect.

23.7.2 Analog I/O

Analog resources, such as LPCOMP and SAR ADC, which require low-impedance routing paths have dedicated pins. Dedicated analog pins provide direct connections to specific analog blocks. They help improve performance and should be given priority over other pins when using these analog resources. See the PSoC 61 datasheet/PSoC 62 datasheet for details on these dedicated pins of the PSoC 6 MCU.

To configure a GPIO as a dedicated analog I/O, it should be configured in high-impedance analog mode (see Table 23-1) with input buffer disabled. The respective connection should be enabled via registers in the specific analog resource.

To configure a GPIO as an analog pin connecting to AMUXBUS, it should be configured in high-impedance analog mode with the input buffer disabled and then routed to the correct AMUXBUS using the HSIOM_PORT_SELx register.

While it is preferred for analog pins to disable the input buffer, it is acceptable to enable the input buffer if simultaneous analog and digital input features are required.

23.7.2.1 AMUXBUS Connection

Static connection of AMUXBUS A or B is made by selecting AMUXA or AMUXB in the HSIOM_PORT_SELx register.

To properly configure a pin as AMUXBUS input, follow these steps:

1. Configure the GPIO_PRTx_CFG register to set the pin in high-Impedance mode with input buffer disabled, enabling analog connectivity on the pin.
2. Configure the HSIOM_PRT_SELx register to connect the pin to AMUXBUS A or B.

23.7.3 LCD Drive

GPIOs have the capability of driving an LCD common or segment line. HSIOM_PORT_SELx registers are used to select pins for the LCD drive. See the LCD Direct Drive chapter on page 483 for details.

23.7.4 CapSense

The pins that support CapSense can be configured as CapSense widgets such as buttons, slider elements, touchpad elements, or proximity sensors. CapSense also requires external capacitors and optional shield lines. See the *PSoC 4 and PSoC 6 MCU CapSense Design Guide* for more details.

23.8 Smart I/O

The Smart I/O block adds programmable logic to an I/O port. This programmable logic integrates board-level Boolean logic functionality such as AND, OR, and XOR into the port. A graphical interface is provided with the ModusToolbox install for configuring the Smart I/O block. For more information about the configurator tool, see the *ModusToolbox Smart I/O Configurator Guide*.

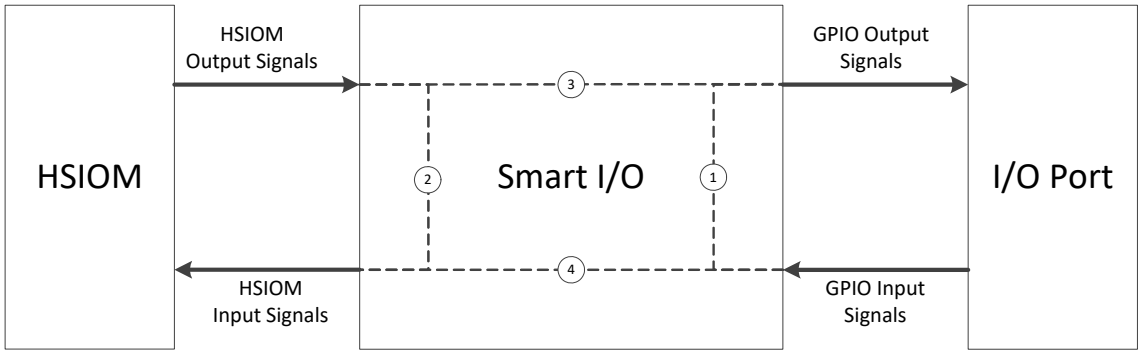
The Smart I/O block has these features:

- Integrate board-level Boolean logic functionality into a port
- Ability to preprocess HSIOM input signals from the GPIO port pins
- Ability to post-process HSIOM output signals to the GPIO port pins
- Support in all device power modes except Hibernate
- Integrate closely to the I/O pads, providing shortest signal paths with programmability

23.8.1 Overview

The Smart I/O block is positioned in the signal path between the HSIOM and the I/O port. The HSIOM multiplexes the output signals from fixed-function peripherals and CPU to a specific port pin and vice-versa. The Smart I/O block is placed on this signal path, acting as a bridge that can process signals between port pins and HSIOM, as shown in Figure 23-7.

Figure 23-7. Smart I/O Interface



The signal paths supported through the Smart I/O block as shown in Figure 23-7 are as follows:

1. Implement self-contained logic functions that directly operate on port I/O signals
2. Implement self-contained logic functions that operate on HSIOM signals
3. Operate on and modify HSIOM output signals and route the modified signals to port I/O signals
4. Operate on and modify port I/O signals and route the modified signals to HSIOM input signals

The following sections discuss the Smart I/O block components, routing, and configuration in detail. In these sections, GPIO signals (io_data) refer to the input/output signals from the I/O port; device or chip (chip_data) signals refer to the input/output signals from HSIOM.

23.8.2 Block Components

The internal logic of the Smart I/O includes these components:

- Clock/reset
- Synchronizers
- Three-input lookup table (LUT)
- Data unit

23.8.2.1 Clock and Reset

The clock and reset component selects the Smart I/O block's clock (clk_block) and reset signal (rst_block_n). A single clock and reset signal is used for all components in the block. The clock and reset sources are determined by the CLOCK_SRC[4:0] bitfield of the SMARTIO_PRTx_CTL register. The selected clock is used for the synchronous logic in the block components, which includes the I/O input synchronizers, LUT, and data unit components. The selected reset is used to asynchronously reset the synchronous logic in the LUT and data unit components.

Note that the selected clock (clk_block) for the block's synchronous logic is not phase-aligned with other synchronous logic in the device, operating on the same clock. Therefore, communication between Smart I/O and other synchronous logic should be treated as asynchronous.

The following clock sources are available for selection:

- GPIO input signals "io_data_in[7:0]". These clock sources have no associated reset.
- HSIOM output signals "chip_data[7:0]". These clock sources have no associated reset.
- Smart I/O clock (clk_smartio). This is derived from the system clock (clk_sys) using a peripheral clock divider. See the Clocking System chapter on page 242 for details on peripheral clock dividers. This clock is available only in System LP and ULP power modes. The clock can have one out of two associated resets: rst_sys_act_n and rst_sys_dpslp_n. These resets determine in which system power modes the block synchronous state is reset; for example, rst_sys_act_n is intended for Smart I/O synchronous functionality in the System LP and ULP power modes and reset is activated in the System Deep Sleep power mode.
- Low-frequency system clock (clk_lf). This clock is available in System Deep Sleep power mode. This clock has an associated reset, rst_lf_dpslp_n. Reset is activated if the system enters Hibernate, or is at POR.

When the block is enabled, the selected clock (clk_block) and associated reset (rst_block_n) are provided to the fabric components. When the fabric is disabled, no clock is released to the fabric components and the reset is activated (the LUT and data unit components are set to the reset value of '0').

The I/O input synchronizers introduce a delay of two clk_block cycles (when synchronizers are enabled). As a result, in the first two cycles, the block may be exposed to stale data from the synchronizer output. Hence, during the first two clock cycles, the reset is activated and the block is in bypass mode.

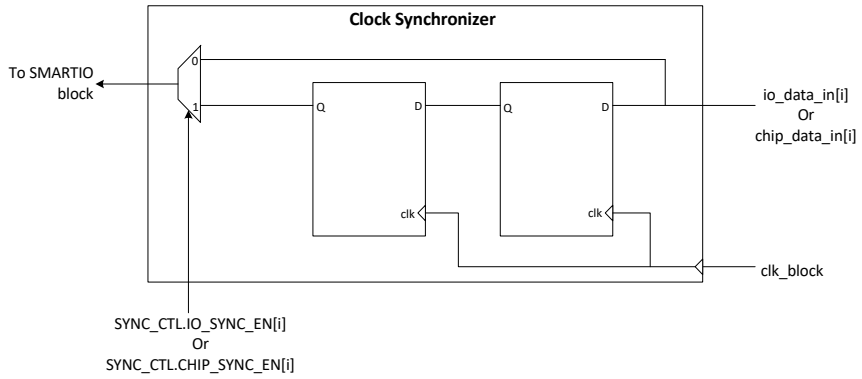
Table 23-6. Clock and Reset Register Control

Register[BIT_POS]	Bit Name	Description
SMARTIO_PRTn_CTL[12:8]	CLOCK_SRC[4:0]	Clock (clk_block)/reset (rst_block_n) source selection: 0: io_data_in[0]/1 ... 7: io_data_in[7]/1 8: chip_data[0]/1 ... 15: chip_data[7]/1 16: clk_smartio/rst_sys_act_n; asserts reset in any power mode other than System LP or ULP; that is, Smart I/O is active only in LP or ULP power modes with clock from the peripheral divider. 17: clk_smartio/rst_sys_dpslp_n. Smart I/O is active in all power modes with a clock from the peripheral divider. However, the clock will not be active in System deep sleep power mode. 19: clk_lf/rst_lf_dpslp_n. Smart I/O is active in all power modes with a clock from ILO. 20-30: Clock source is a constant '0'. Any of these clock sources should be selected when Smart I/O is disabled to ensure low power consumption. 31: clk_sys/1. This selection is not intended for clk_sys operation. However, for asynchronous operation, three clk_sys cycles after enabling, the Smart I/O is fully functional (reset is de-activated). To be used for asynchronous (clockless) block functionality.

23.8.2.2 Synchronizer

Each GPIO input signal and device input signal (HSIOM input) can be used either asynchronously or synchronously. To use the signals synchronously, a double flip-flop synchronizer, as shown in Figure 23-8, is placed on both these signal paths to synchronize the signal to the Smart I/O clock (clk_block). The synchronization for each pin/input is enabled or disabled by setting or clearing the IO_SYNC_EN[i] bitfield for GPIO input signal and CHIP_SYNC_EN[i] for HSIOM signal in the SMARTIO_PRTx_SYNC_CTL register, where 'i' is the pin number.

Figure 23-8. Smart I/O Clock Synchronizer



23.8.2.3 Lookup Table (LUT)

Each Smart I/O block contains eight lookup table (LUT) components. The LUT component consists of a three-input LUT and a flip-flop. Each LUT block takes three input signals and generates an output based on the configuration set in the SMARTIO_PRTx_LUT_CTLy register (y denotes the LUT number). For each LUT, the configuration is determined by an 8-bit lookup vector LUT[7:0] and a 2-bit opcode OPC[1:0] in the SMARTIO_PRTx_LUT_CTLy register. The 8-bit vector is used as a lookup table for the three input signals. The 2-bit opcode determines the usage of the flip-flop. The LUT configuration for different opcodes is shown in Figure 23-9.

The SMARTIO_PRTx_LUT_SELy registers select the three input signals (tr0_in, tr1_in, and tr2_in) going into each LUT. The input can come from the following sources:

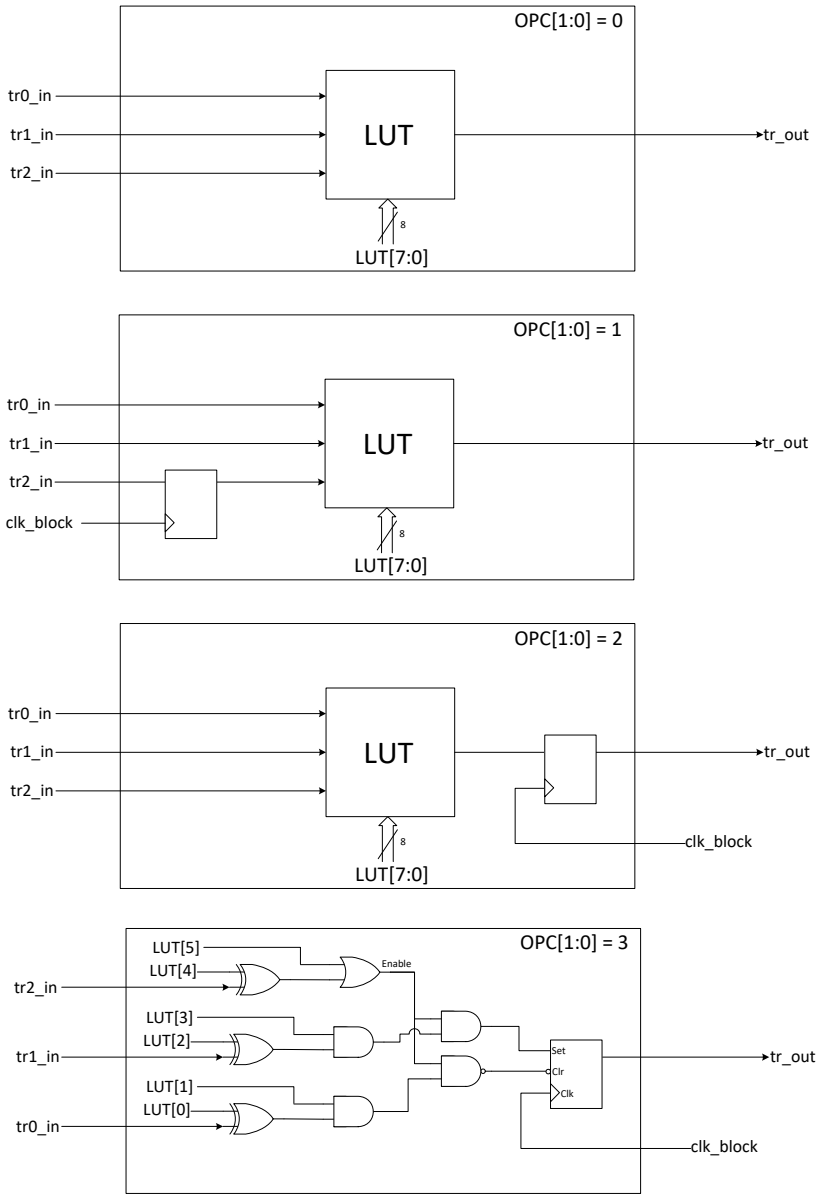
- Data unit output
- Other LUT output signals (tr_out)
- HSIOM output signals (chip_data[7:0])
- GPIO input signals (io_data[7:0])

LUT_TR0_SEL[3:0] bits of the SMARTIO_PRTx_LUT_SELy register selects the tr0_in signal for the y^{th} LUT. Similarly, LUT_TR1_SEL[3:0] bits and LUT_TR2_SEL[3:0] bits select the tr1_in and tr2_in signals, respectively. See Table 23-7 for details.

Table 23-7. LUT Register Control

Register[BIT_POS]	Bit Name	Description
SMARTIO_PRTx_LUT_CTLy[7:0]	LUT[7:0]	LUT configuration. Depending on the LUT opcode (LUT_OPC), the internal state, and the LUT input signals tr0_in, tr1_in, and tr2_in, the LUT configuration is used to determine the LUT output signal and the next sequential state.
SMARTIO_PRTx_LUT_CTLy[9:8]	LUT_OPC[1:0]	LUT opcode specifies the LUT operation as illustrated in Figure 23-9.
SMARTIO_PRTx_LUT_SELy[3:0]	LUT_TR0_SEL[3:0]	LUT input signal “tr0_in” source selection: 0: Data unit output 1: LUT 1 output 2: LUT 2 output 3: LUT 3 output 4: LUT 4 output 5: LUT 5 output 6: LUT 6 output 7: LUT 7 output 8: chip_data[0] (for LUTs 0, 1, 2, 3); chip_data[4] (for LUTs 4, 5, 6, 7) 9: chip_data[1] (for LUTs 0, 1, 2, 3); chip_data[5] (for LUTs 4, 5, 6, 7) 10: chip_data[2] (for LUTs 0, 1, 2, 3); chip_data[6] (for LUTs 4, 5, 6, 7) 11: chip_data[3] (for LUTs 0, 1, 2, 3); chip_data[7] (for LUTs 4, 5, 6, 7) 12: io_data_in[0] (for LUTs 0, 1, 2, 3); io_data[4] (for LUTs 4, 5, 6, 7) 13: io_data_in[1] (for LUTs 0, 1, 2, 3); io_data[5] (for LUTs 4, 5, 6, 7) 14: io_data_in[2] (for LUTs 0, 1, 2, 3); io_data[6] (for LUTs 4, 5, 6, 7) 15: io_data_in[3] (for LUTs 0, 1, 2, 3); io_data[7] (for LUTs 4, 5, 6, 7)
SMARTIO_PRTx_LUT_SELy[11:8]	LUT_TR1_SEL[3:0]	LUT input signal “tr1_in” source selection: 0: LUT 0 output 1: LUT 1 output 2: LUT 2 output 3: LUT 3 output 4: LUT 4 output 5: LUT 5 output 6: LUT 6 output 7: LUT 7 output 8: chip_data[0] (for LUTs 0, 1, 2, 3); chip_data[4] (for LUTs 4, 5, 6, 7) 9: chip_data[1] (for LUTs 0, 1, 2, 3); chip_data[5] (for LUTs 4, 5, 6, 7) 10: chip_data[2] (for LUTs 0, 1, 2, 3); chip_data[6] (for LUTs 4, 5, 6, 7) 11: chip_data[3] (for LUTs 0, 1, 2, 3); chip_data[7] (for LUTs 4, 5, 6, 7) 12: io_data_in[0] (for LUTs 0, 1, 2, 3); io_data[4] (for LUTs 4, 5, 6, 7) 13: io_data_in[1] (for LUTs 0, 1, 2, 3); io_data[5] (for LUTs 4, 5, 6, 7) 14: io_data_in[2] (for LUTs 0, 1, 2, 3); io_data[6] (for LUTs 4, 5, 6, 7) 15: io_data_in[3] (for LUTs 0, 1, 2, 3); io_data[7] (for LUTs 4, 5, 6, 7)
SMARTIO_PRTx_LUT_SELy[19:16]	LUT_TR2_SEL[3:0]	LUT input signal “tr2_in” source selection. Encoding is the same as for LUT_TR1_SEL.

Figure 23-9. Smart I/O LUT Configuration



23.8.2.4 Data Unit (DU)

Each Smart I/O block includes a data unit (DU) component. The DU consists of a simple 8-bit datapath. It is capable of performing simple increment, decrement, increment/decrement, shift, and AND/OR operations. The operation performed by the DU is selected using a 4-bit opcode DU_OPC[3:0] bitfield in the SMARTIO_PRTx_DU_CTL register.

The DU component supports up to three input trigger signals (tr0_in, tr1_in, tr2_in) similar to the LUT component. These signals are used to initiate an operation defined by the DU opcode. In addition, the DU also includes two 8-bit data inputs (data0_in[7:0] and data1_in[7:0]) that are used to initialize the 8-bit internal state (data[7:0]) or to provide a reference. The 8-bit data input source is configured as:

- Constant ‘0x00’
- io_data_in[7:0]
- chip_data_in[7:0]
- DATA[7:0] bitfield of SMARTIO_PRTx_DATA register

The trigger signals are selected using the DU_TRx_SEL[3:0] bitfield of the SMARTIO_PRTx_DU_SEL register. The DUT_-DATAx_SEL[1:0] bits of the SMARTIO_PRTx_DU_SEL register select the 8-bit input data source. The size of the DU (number of bits used by the datapath) is defined by the DU_SIZE[2:0] bits of the SMARTIO_PRTx_DU_CTL register. See Table 23-8 for register control details.

Table 23-8. Data Unit Register Control

Register[BIT_POS]	Bit Name	Description
SMARTIO_PRTx_DU_CTL[2:0]	DU_SIZE[2:0]	Size/width of the data unit (in bits) is DU_SIZE+1. For example, if DU_SIZE is 7, the width is 8 bits.
SMARTIO_PRTx_DU_CTL[11:8]	DU_OPC[3:0]	Data unit opcode specifies the data unit operation: 1: INCR 2: DECR 3: INCR_WRAP 4: DECR_WRAP 5: INCR_DECR 6: INCR_DECR_WRAP 7: ROR 8: SHR 9: AND_OR 10: SHR_MAJ3 11: SHR_EQL Otherwise: Undefined.
SMARTIO_PRTx_DU_SEL[3:0]	DU_TR0_SEl[3:0]	Data unit input signal “tr0_in” source selection: 0: Constant '0'. 1: Constant '1'. 2: Data unit output. 10–3: LUT 7–0 outputs. Otherwise: Undefined.
SMARTIO_PRTx_DU_SEL[11:8]	DU_TR1_SEl[3:0]	Data unit input signal “tr1_in” source selection. Encoding same as DU_TR0_-SEL
SMARTIO_PRTx_DU_SEL[19:16]	DU_TR2_SEl[3:0]	Data unit input signal “tr2_in” source selection. Encoding same as DU_TR0_-SEL
SMARTIO_PRTx_DU_SEL[25:24]	DU_DATA0_SEL[1:0]	Data unit input data “data0_in” source selection: 0: 0x00 1: chip_data[7:0]. 2: io_data[7:0]. 3: SMARTIO_PRTx_DATA.DATA[7:0] register field.
SMARTIO_PRTx_DU_SEL[29:28]	DU_DATA1_SEL[1:0]	Data unit input data “data1_in” source selection. Encoding same as DU_DA-TA0_SEL.
SMARTIO_PRTx_DATA[7:0]	DATA[7:0]	Data unit input data source.

The DU generates a single output trigger signal (tr_out). The internal state (du_data[7:0]) is captured in flip-flops and requires clk_block.

The following pseudo code describes the various datapath operations supported by the DU opcode. Note that “Comb” describes the combinatorial functionality – that is, functions that operate independent of previous output states. “Reg” describes the registered functionality – that is, functions that operate on inputs and previous output states (registered using flip-flops).

```
// The following is shared by all operations.
mask = (2 ^ (DU_SIZE+1) – 1)
data_eql_data1_in = (data & mask) == (data1_in & mask));
data_eql_0        = (data & mask) == 0);
data_incr         = (data + 1) & mask;
data_decr         = (data - 1) & mask;
```

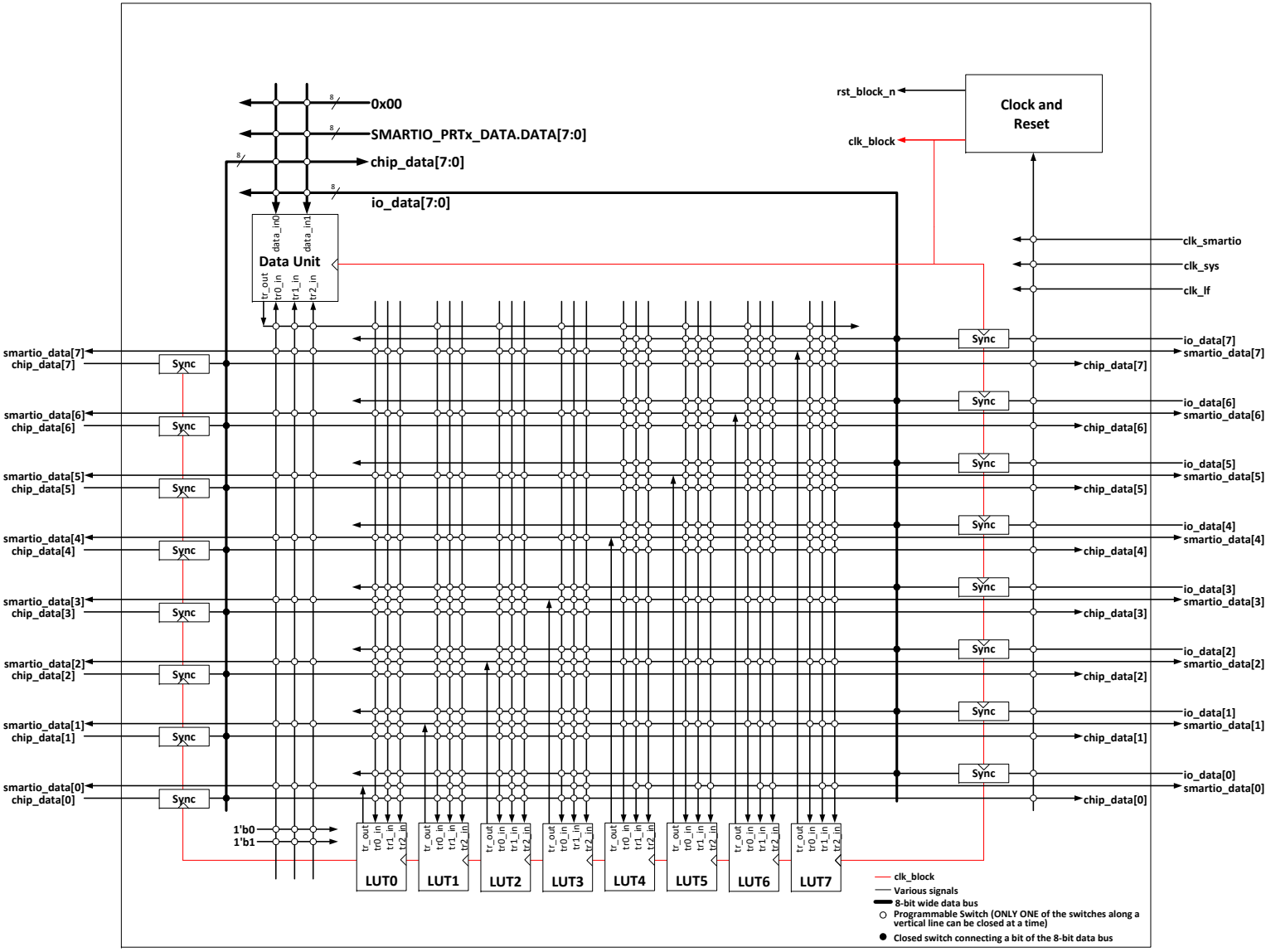
```
data0_masked      = data_in0 & mask;

// INCR operation: increments data by 1 from an initial value (data0) until it reaches a
// final value (data1).
Comb:tr_out = data_eql_data1_in;
Reg:  data <= data;
      if (tr0_in)      data <= data0_masked; //tr0_in is reload signal - loads masked data0
                                             // into data
      else if (tr1_in) data <= data_eql_data1_in ? data : data_incr; //increment data until
                                                                     // it equals data1

// INCR_WRAP operation: operates similar to INCR but instead of stopping at data1, it wraps
// around to data0.
Comb:tr_out = data_eql_data1_in;
Reg:  data <= data;
      if (tr0_in)      data <= data0_masked;
      else if (tr1_in) data <= data_eql_data1_in ? data0_masked : data_incr;

// DECR operation: decrements data from an initial value (data0) until it reaches 0.
Comb:tr_out = data_eql_0;
Reg:  data <= data;
      if (tr0_in)      data <= data0_masked;
      else if (tr1_in) data <= data_eql_0        ? data : data_decr;

// DECR_WRAP operation: works similar to DECR. Instead of stopping at 0, it wraps around to
// data0.
Comb:tr_out = data_eql_0;
Reg:  data <= data;
      if (tr0_in)      data <= data0_masked;
      else if (tr1_in) data <= data_eql_0        ? data0_masked: data_decr;

// INCR_DECR operation: combination of INCR and DECR. Depending on trigger signals it either
// starts incrementing or decrementing. Increment stops at data1 and decrement stops at 0.
Comb:tr_out = data_eql_data1_in | data_eql_0;
Reg:  data <= data;
      if (tr0_in)      data <= data0_masked; // Increment operation takes precedence over
                                             // decrement when both signal are available
      else if (tr1_in) data <= data_eql_data1_in ? data : data_incr;
      else if (tr2_in) data <= data_eql_0  ? data : data_decr;

// INCR_DECR_WRAP operation: same functionality as INCR_DECR with wrap around to data0 on
// reaching the limits.
Comb:tr_out = data_eql_data1_in | data_eql_0;
Reg:  data <= data;
      if (tr0_in)      data <= data0_masked;
      else if (tr1_in) data <= data_eql_data1_in ? data0_masked : data_incr;
      else if (tr2_in) data <= data_eql_0  ? data0_masked : data_decr;

// ROR operation: rotates data right and LSb is sent out. The data for rotation is taken from
// data0.
Comb:tr_out = data[0];
Reg:  data <= data;
      if (tr0_in)      data          <= data0_masked;
      else if (tr1_in) {
                       data          <= {0, data[7:1]} & mask; //Shift right operation
                       data[du_size] <= data[0]; //Move the data[0] (LSb) to MSb
      }
```

```
// SHR operation: performs shift register operation. Initial data (data0) is shifted out and
// data on tr2_in is shifted in.
Comb:tr_out = data[0];
Reg:  data <= data;
      if (tr0_in)      data          <= data0_masked;
      else if (tr1_in) {
                       data          <= {0, data[7:1]} & mask; //Shift right operation
                       data[du_size] <= tr2_in; //tr2_in Shift in operation
      }

// SHR_MAJ3 operation: performs the same functionality as SHR. Instead of sending out the
// shifted out value, it sends out a '1' if in the last three samples/shifted-out values
// (data[0]), the signal high in at least two samples. otherwise, sends a '0'. This function
// sends out the majority of the last three samples.
Comb:tr_out =   (data == 0x03)
              | (data == 0x05)
              | (data == 0x06)
              | (data == 0x07);
Reg:  data <= data;
      if (tr0_in)      data          <= data0_masked;
      else if (tr1_in) {
                       data          <= {0, data[7:1]} & mask;
                       data[du_size] <= tr2_in;
      }

// SHR_EQL operation: performs the same operation as SHR. Instead of shift-out, the output is
// a comparison result (data0 == data1).
Comb:tr_out = data_eql_data1_in;
Reg:  data <= data;
      if      (tr0_in) data          <= data0_masked;
      else if (tr1_in) {
                       data          <= {0, data[7:1]} & mask;
                       data[du_size] <= tr2_in;
      }

// AND_OR operation: ANDs data1 and data0 along with mask; then, ORs all the bits of the
// ANDed output.
Comb:tr_out = | (data & data1_in & mask);
Reg:  data <= data;
      if (tr0_in) data <= data0_masked;
```

23.8.3 Routing

The Smart I/O block includes many switches that are used to route the signals in and out of the block and also between various components present inside the block. The routing switches are handled through the PRTGIO_PRTx_LUT_SELy and SMARTIO_PRTx_DU_SEL registers. Refer to the registers TRM for details. The Smart I/O internal routing is shown in Figure 23-10. In the figure, note that LUT7 to LUT4 operate on io_data/chip_data[7] to io_data/chip_data[4] whereas LUT3 to LUT0 operate on io_data/chip_data[3] to io_data/chip_data[0].

Figure 23-10. Smart I/O Routing



23.8.4 Operation

The Smart I/O block should be configured and operated as follows:

1. Before enabling the block, all the components and routing should be configured as explained in “Block Components” on page 272.
2. In addition to configuring the components and routing, some block level settings must be configured correctly for desired operation.
   a. Bypass control: The Smart I/O path can be bypassed for a particular GPIO signal by setting the BYPASS[i] bitfield in the SMARTIO_PRTx_CTL register. When bit ‘i’ is set in the BYPASS[7:0] bitfield, the i^{th} GPIO signal is bypassed to the HSIOM signal path directly – Smart I/O logic will not be present in that signal path. This is useful when the Smart I/O function is required only on select I/Os.
   b. Pipelined trigger mode: The LUT input multiplexers and the LUT component itself do not include any combinatorial loops. Similarly, the data unit also does not include any combinatorial loops. However, when one LUT interacts with the other or to the data unit, inadvertent combinatorial loops are possible. To overcome this limitation, the PIPELINE_EN bitfield of the SMARTIO_PRTx_CTL register is used. When set, all the outputs (LUT and DU) are registered before branching out to other components.
3. After the Smart I/O block is configured for the desired functionality, the block can be enabled by setting the ENABLED bitfield of the SMARTIO_PRTx_CTL register. If disabled, the Smart I/O block is put in bypass mode, where the GPIO signals are directly controlled by the HSIOM signals and vice-versa. The Smart I/O block must be configured; that is, all register settings must be updated before enabling the block to prevent glitches during register updates.

Table 23-9. Smart I/O Block Controls

Register [BIT_POS]	Bit Name	Description
SMARTIO_PRTx_CTL[25]	PIPELINE_EN	Enable for pipeline register: 0: Disabled (register is bypassed). 1: Enabled
SMARTIO_PRTx_CTL[31]	ENABLED	Enable Smart I/O. Should only be set to '1' when the Smart I/O is completely configured: 0: Disabled (signals are bypassed; behavior as if BYPASS[7:0] is 0xFF). When disabled, the block (data unit and LUTs) reset is activated. If the block is disabled: - The PIPELINE_EN register field should be set to '1', to ensure low power consumption. - The CLOCK_SRC register field should be set to 20 to 30 (clock is constant '0'), to ensure low power consumption. 1: Enabled. When enabled, it takes three clk_block clock cycles until the block reset is de-activated and the block becomes fully functional. This action ensures that the I/O pins' input synchronizer states are flushed when the block is fully functional.
SMARTIO_PRTx_CTL[7:0]	BYPASS[7:0]	Bypass of the Smart I/O, one bit for each I/O pin: BYPASS[i] is for I/O pin i. When ENABLED is '1', this field is used. When ENABLED is '0', this field is not used and Smart I/O is always bypassed. 0: No bypass (Smart I/O is present in the signal path) 1: Bypass (Smart I/O is absent in the signal path)

23.9 Registers

Table 23-10. I/O Registers

Name	Description
GPIO_PRTx_OUT	Port output data register reads and writes the output driver data for I/O pins in the port.
GPIO_PRTx_OUT_CLR	Port output data clear register clears output data of specific I/O pins in the port.
GPIO_PRTx_OUT_SET	Port output data set register sets output data of specific I/O pins in the port.
GPIO_PRTx_OUT_INV	Port output data invert register inverts output data of specific I/O pins in the port.
GPIO_PRTx_IN	Port input state register reads the current pin state present on I/O pin inputs.
GPIO_PRTx_INTR	Port interrupt status register reads the current pin interrupt state.
GPIO_PRTx_INTR_MASK	Port interrupt mask register configures the mask that forwards pin interrupts to the CPU's interrupt controller.
GPIO_PRTx_INTR_MASKED	Port interrupt masked status register reads the masked interrupt status forwarded to the CPU interrupt controller.
GPIO_PRTx_INTR_SET	Port interrupt set register allows firmware to set pin interrupts.
GPIO_PRTx_INTR_CFG	Port interrupt configuration register selects the edge detection type for each pin interrupt.
GPIO_PRTx_CFG	Port configuration register selects the drive mode and input buffer enable for each pin.
GPIO_PRTx_CFG_IN	Port input buffer configuration register configures the input buffer mode for each pin.
GPIO_PRTx_CFG_OUT	Port output buffer configuration register selects the output driver slew rate for each pin.
HSIOM_PORT_SELx	High-speed I/O Mux (HSIOM) port selection register selects the hardware peripheral connection to I/O pins.

Note The 'x' in the GPIO register name denotes the port number. For example, GPIO_PTR1_OUT is the Port 1 output data register.

24. Watchdog Timer

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The watchdog timer (WDT) is a hardware timer that automatically resets the device in the event of an unexpected firmware execution path. The WDT, if enabled, must be serviced periodically in firmware to avoid a reset. Otherwise, the timer elapses and generates a device reset. In addition, the WDT can be used as an interrupt source or a wakeup source in low-power modes.

The PSoC 6 MCU family includes one free-running WDT and two multi-counter WDTs (MCWDT). The WDT has a 16-bit counter. Each MCWDT has two 16-bit counters and one 32-bit counter. Thus, the watchdog system has a total of seven counters – five 16-bit and two 32-bit. All 16-bit counters can generate a watchdog device reset. All seven counters can generate an interrupt on a match event.

24.1 Features

The PSoC 6 MCU WDT supports these features:

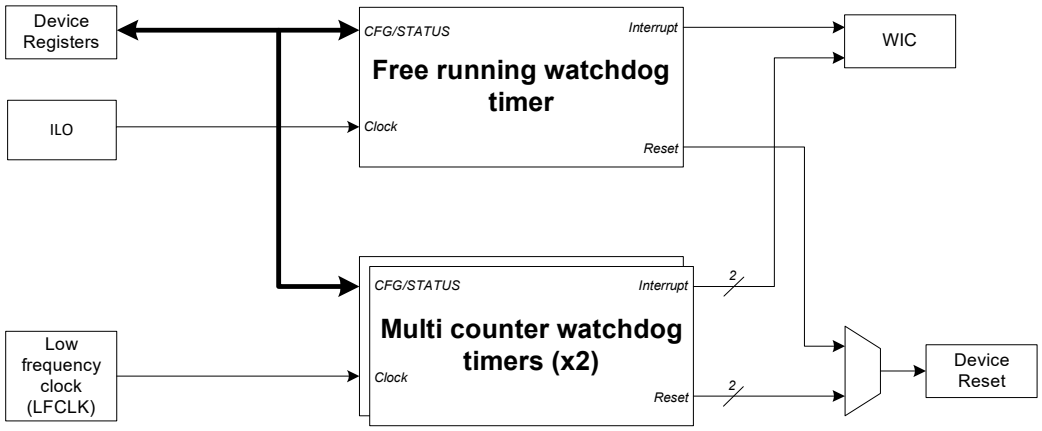
- One 16-bit free-running WDT with:
  - ILO as the input clock source
  - Device reset generation if not serviced within a configurable interval
  - Periodic Interrupt/wakeup generation in LP/ULP Active, LP/ULP Sleep, Deep Sleep, and Hibernate power modes
- Two MCWDTs, each supporting:
  - Device reset generation if not serviced within a configurable interval
  - LFCLK (ILO or WCO) as the input clock source
  - Periodic interrupt/wake up generation in LP/ULP Active, LP/ULP Sleep, and Deep Sleep power modes (Hibernate mode is not supported)
  - Two 16-bit and one 32-bit independent counters, which can be configured as a single 64-bit or 48-bit (with one 16-bit independent counter), or two 32-bit cascaded counters

24.2 Architecture

The PSoC 6 MCU supports 174 system interrupts. The interrupts are routed to both the CPU cores. In the case of CM4 only the first 39 interrupts are routed to WIC while all 174 interrupts are routed to NVIC. The CM0 has access to only eight interrupts of the maximum supported 32 interrupts. The 174 interrupt sources are multiplexed and at a time eight interrupt sources can be connected to the CM0. The CPUSS_CM0_SYSTEM_INT_CTLx register decides which interrupts are connected to the CM0. See the Interrupts chapter on page 56 for details on how to configure the interrupt for the free-running

WDT/MCWDT to the required CPU. The free-running WDT/MCWDT resource must be used by one CPU only; it is not intended for simultaneous use by both CPUs because of the complexity involved in coordination.

Figure 24-1. Watchdog Timer Block Diagram

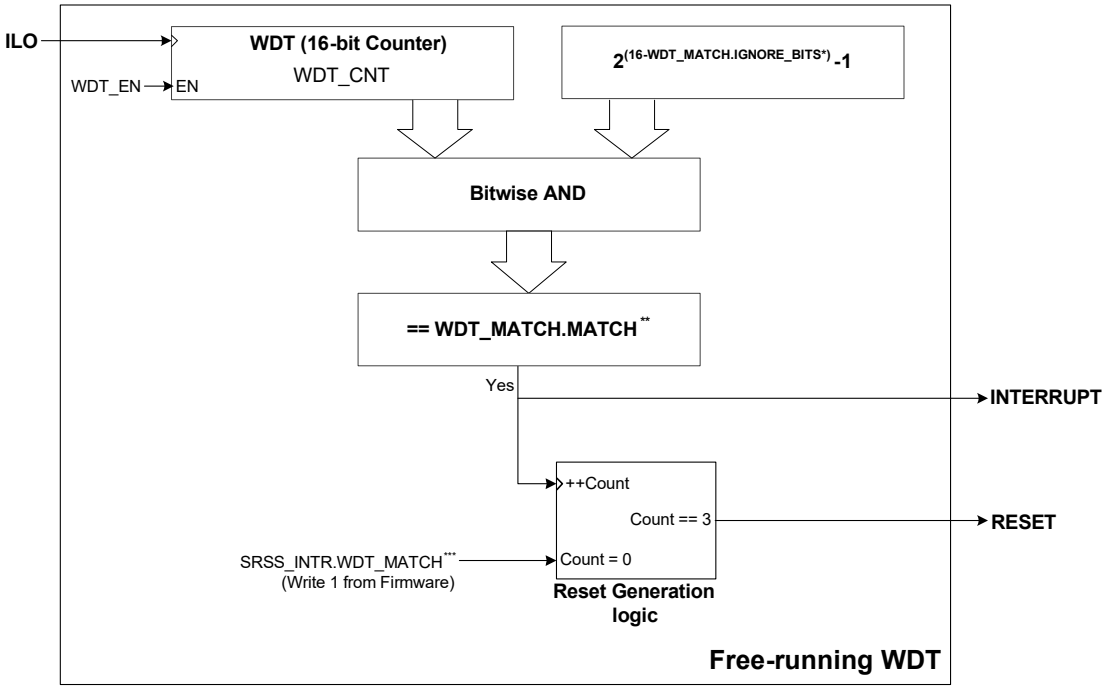


24.3 Free-running WDT

24.3.1 Overview

Figure 24-2 shows the functional overview of the free-running WDT. The WDT has a free-running wraparound up-counter with a maximum of 16-bit resolution. The counter is clocked by the ILO. The timer can generate an interrupt on match and a reset event on the third unhandled interrupt. The number of bits used for a match comparison is configurable as depicted in Figure 24-2.

Figure 24-2. Free-running WDT Functional Diagram



* WDT_MATCH.IGNORE_BITS refer to value held by the bits[19:16] of the WDT_MATCH register.
** WDT_MATCH.MATCH refer to the value held by the bits[15:0] of the WDT_MATCH register.
*** SRSS_INTR.WDT_MATCH refers to the WDT_MATCH bit of the SRSS_INTR register.

When enabled, the WDT counts up on each rising edge of the ILO. When the counter value (WDT_CNT register) equals the match value stored in MATCH bits [15:0] of the WDT_MATCH register, an interrupt is generated. The match event does not reset the WDT counter and the WDT keeps counting until it reaches the 16-bit boundary (65535) at which point, it wraps around to 0 and counts up. The match interrupt is generated every time the counter value equals the match value.

The WDT_MATCH bit of the SRSS_INTR register is set whenever a WDT match interrupt occurs. This interrupt must be cleared by writing a '1' to the same bit. Clearing the interrupt resets the watchdog. If the firmware does not clear the interrupt for two consecutive occasions, the third interrupt generates a device reset.

In addition, the WDT provides an option to set the number of bits to be used for comparison. The IGNORE_BITS (bits [19:16] of the WDT_MATCH register) is used for this purpose. These bits configure the number of MSbs to ignore from the 16-bit count value while performing the match. For instance, when the value of these bits equals 3, the MSb 3 bits are ignored while performing the match and the WDT counter behaves similar to a 13-bit counter. Note that these bits do not reduce the counter size – the WDT_CNT register still counts from 0 to 65535 (16-bit).

The WDT can be enabled or disabled using the WDT_EN bit [0] of the WDT_CTL register. The WDT_CTL register provides a mechanism to lock the WDT configuration registers. The WDT_LOCK bits [31:30] control the lock status of the WDT registers. These bits are special bits, which can enable the lock in a single write; to release the lock, two different writes are required. The WDT_LOCK bits protect the WDT_EN bit, WDT_MATCH register, CLK_ILO_CONFIG register, and LFCLK_SEL bits [1:0] of the CLK_SELECT register. Note that the WDT_LOCK bits are not retained in Deep Sleep mode and reset to their default (LOCK) state after a deep sleep wakeup. As a result, to update any register protected by the WDT_LOCK bits after a deep sleep wakeup, a WDT UNLOCK sequence should be issued before the register update.

Table 24-1 explains various registers and bitfields used to configure and use the WDT.

Table 24-1. Free-running WDT Configuration Options

Register [Bit_Pos]	Bit_Name	Description
WDT_CTL[0]	WDT_EN	Enable or disable the watchdog reset 0: WDT reset disabled 1: WDT reset enabled
WDT_CTL[31:30]	WDT_LOCK	Lock or unlock write access to the watchdog configuration and clock related registers. When the bits are set, the lock is enabled. 0: No effect 1: Clear bit 0 2: Clear bit 1 3: Set both bit 0 and 1 (lock enabled) WDT will lock on a reset. This field is not retained in Deep Sleep or Hibernate mode, so the WDT will be locked after wakeup from these modes.
WDT_CNT[15:0]	COUNTER	Current value of WDT counter
WDT_MATCH[15:0]	MATCH	Match value to a generate watchdog match event
WDT_MATCH[19:16]	IGNORE_BITS	Number of MSbs of the WDT_CNT register to ignore for comparison with the MATCH value. Up to 12 MSbs can be ignored; settings above 12 act same as a setting of 12.
SRSS_INTR[0]	WDT_MATCH	WDT interrupt request This bit is set whenever a watchdog match event happens. The WDT interrupt is cleared by writing a '1' to this bit
SRSS_INTR_MASK[0]	WDT_MATCH	Mask for the WDT interrupt 0: WDT interrupt is blocked 1: WDT interrupt is forwarded to CPU

24.3.2 Watchdog Reset

A watchdog is typically used to protect the device against firmware/system crashes or faults. When the WDT is used to protect against system crashes, the WDT interrupt bit should be cleared by a portion of the code that is not directly associated with the WDT interrupt. Otherwise, even if the main function of the firmware crashes or is in an endless loop, the WDT interrupt vector can still be intact and feed the WDT periodically. Note that when the debug probe is connected, the device reset is blocked and an interrupt is generated instead.

The safest way to use the WDT against system crashes is to:

- Configure the watchdog reset period such that firmware is able to reset the watchdog at least once during the period, even along the longest firmware delay path. If both CM4 and CM0 are being used then, use the free-running WDT for one of the CPU and one MCWDT for the other CPU to prevent firmware crashes on either core.
- Reset (feed) the watchdog by clearing the interrupt bit regularly in the main body of the firmware code by writing a '1' to the WDT_MATCH bit in the SRSS_INTR register. Note that this does not reset the watchdog counter, it feeds only the watchdog so that it does not cause a reset for the next two match events.

Do not reset the watchdog (clear interrupt) in the WDT interrupt service routine (ISR) if WDT is being used as a reset source to protect the system against crashes. Therefore, do not use the WDT reset feature and ISR together.

The recommended steps to use WDT as a reset source are as follows:

1. Make sure the WDT configuration is unlocked by clearing the WDT_LOCK bits[31:30] of the WDT_CTL register. Note that clearing the bits requires two writes to the register with each write clearing one bit as explained in Table 24-1.
2. Write the desired IGNORE_BITS in the WDT_MATCH register to set the counter resolution to be used for the match.
3. Write any match value to the WDT_MATCH register. The match value does not control the period of watchdog reset as the counter is not reset on a match event. This value provides an option to control only the first interrupt interval, after that the successive interrupts' period is defined by the IGNORE_BITS. Approximate watchdog period (in seconds) is given by the following equation:

$$\text{Watchdog reset period} = \text{ILO}_{\text{period}} \times (2 \times 2^{16 - \text{IGNORE\_BITS}} + \text{WDT\_MATCH}) \quad \textbf{Equation 24-1}$$

4. Set the WDT_MATCH bit in the SRSS_INTR register to clear any pending WDT interrupt.
5. Enable ILO by setting the ENABLE bit [31] of the CLK_ILO_CONFIG register.
6. Enable the WDT by setting the WDT_EN bit in WDT_CTL register.
7. Lock the WDT and ILO configuration by writing '3' to the WDT_LOCK bits. This also locks the LFCLK_SEL bits of the CLK_SELECT register.
8. In the firmware, write '1' to the WDT_MATCH bit in the SRSS_INT register to feed (clear interrupt) the watchdog.

24.3.3 Watchdog Interrupt

In addition to generating a device reset, the WDT can be used to generate interrupts. Note that interrupt servicing and watchdog reset cannot be used simultaneously using the free-running WDT.

The watchdog counter can send interrupt requests to the CPU in CPU Active power modes and to the wakeup interrupt controller (WIC) in CPU Sleep and Deep Sleep power modes. In addition, the watchdog is capable of waking up the device from Hibernate power mode. It works as follows:

- **CPU Active Mode**: In Active power mode, the WDT can send the interrupt to the CPU. The CPU acknowledges the interrupt request and executes the ISR. Clear the interrupt in the ISR.
- **CPU Sleep or Deep Sleep Mode**: In this mode, the CPU is powered down. Therefore, the interrupt request from the WDT is directly sent to the WIC, which then wakes up the CPU. The CPU acknowledges the interrupt request and executes the ISR. Clear the interrupt in the ISR.
- **Hibernate Mode**: In this mode, the entire device except a few peripherals (such as WDT and LPCOMP) are powered down. Any interrupt to wake up the device in this mode results in a device reset. Hence, there is no interrupt service routine or mechanism associated with this mode.

For more details on device power modes, see the Device Power Modes chapter on page 225.

Because of its free-running nature, the WDT should not be used for periodic interrupt generation. Use the MCWDT instead; see 24.4 Multi-Counter WDTs. The MCWDT counters can be used to generate periodic interrupts. If absolutely required, follow these steps to use the WDT as a periodic interrupt generator:

1. Unlock the WDT if this is the first update to the WDT registers after a deep sleep or hibernate wakeup, or a device reset.
2. Write the desired IGNORE_BITS in the WDT_MATCH register to set the counter resolution to be used for the match.
3. Write the desired match value to the WDT_MATCH register.
4. Set the WDT_MATCH bit in the SRSS_INTR register to clear any pending WDT interrupt.
5. Enable the WDT interrupt to CPU by setting the WDT_MATCH bit in SRSS_INTR_MASK.
6. Enable SRSS interrupt to the CPU by configuring the appropriate ISER register (See the Interrupts chapter on page 56 for details).
7. In the ISR, unlock the WDT; clear the WDT interrupt and add the desired match value to the existing match value. By doing so, another interrupt is generated when the counter reaches the new match value (period).

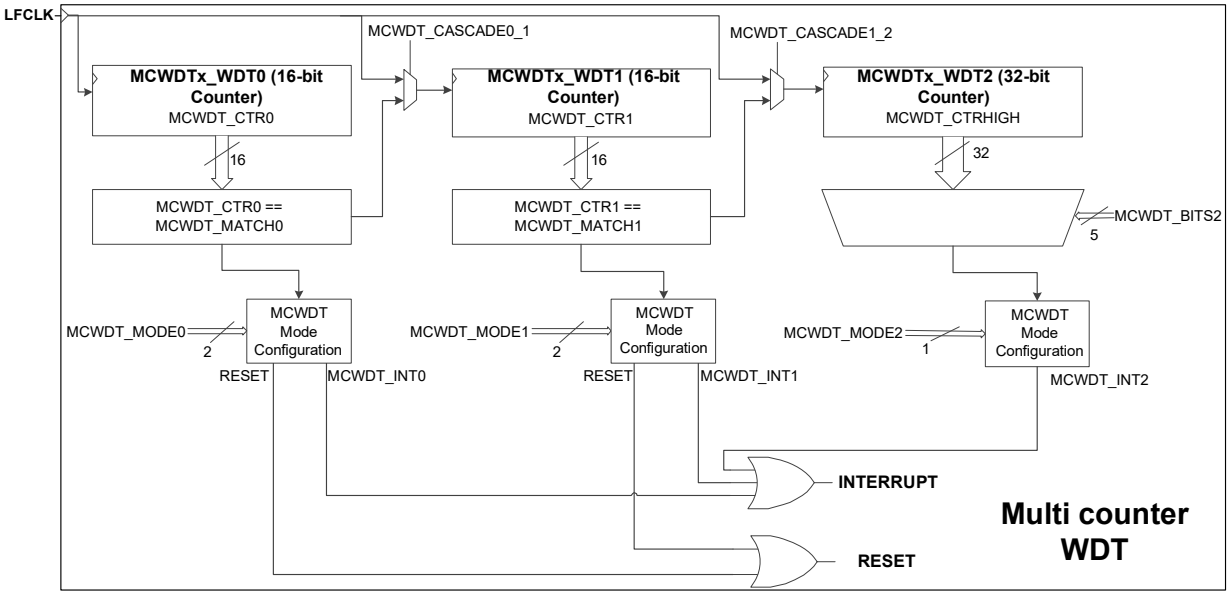
24.4 Multi-Counter WDTs

24.4.1 Overview

Figure 24-3 shows the functional overview of a single multi-counter WDT block. The PSoC 6 MCU has two MCWDT blocks. Each MCWDT block includes two 16-bit counters (MCWDTx_WDT0 and MCWDTx_WDT1) and one 32-bit counter (MCWDTx_WDT2). These counters can be configured to work independently or in cascade (up to 64-bit). The 16-bit counters can generate an interrupt or reset the device. The 32-bit counter can only generate an interrupt. All the counters are clocked by LFCLK.

Note: Because the PSoC 6 MCU includes two CPUs (Cortex-M0+ and Cortex-M4), associate one MCWDT block to only one CPU during runtime. Although both the MCWDT blocks are available for both CPUs, a single MCWDT is not intended to be used by multiple CPUs simultaneously; however, a CPU can be associated with 0 or more MCWDTs.

Figure 24-3. Multi-Counter WDT Functional Diagram



24.4.1.1 MCWDTx_WDT0 and MCWDTx_WDT1 Counters Operation

MCWDTx_WDT0 and MCWDTx_WDT1 are 16-bit up counters, which can be configured to be a 16-bit free-running counter or a counter with any 16-bit period. These counters can be used to generate an interrupt or reset.

The WDT_CTR0 bits [15:0] and WDT_CTR1 bits [16:31] of the MCWDTx_CNTLOW register hold the current counter values of MCWDTx_WDT0 and MCWDTx_WDT1 respectively. The WDT_MATCH0 bits [15:0] and WDT_MATCH1 bits [16:31] of the MCWDTx_MATCH register store the match value for MCWDTx_WDT0 and MCWDTx_WDT1 respectively. The WDT_MODEx bits of the MCWDTx_CONFIG register configure the action the watchdog counter takes on a match event (WDT_MATCHx == WDT_CTRx). The MCWDTx_WDT0/WDT1 counters perform the following actions:

- Assert interrupt (WDT_INTx) on match
- Assert a device reset on match
- Assert an interrupt on match and a device reset on the third unhandled interrupt

In addition to generating reset and interrupt, the match event can be configured to clear the corresponding counter. This is done by setting the WDT_CLEARx bit of the MCWDTx_CONFIG register. MCWDTx_WDT0/WDT1 counter operation is shown in Figure 24-4.

Figure 24-4. MCWDTx_WDT0/WDT1 Operation

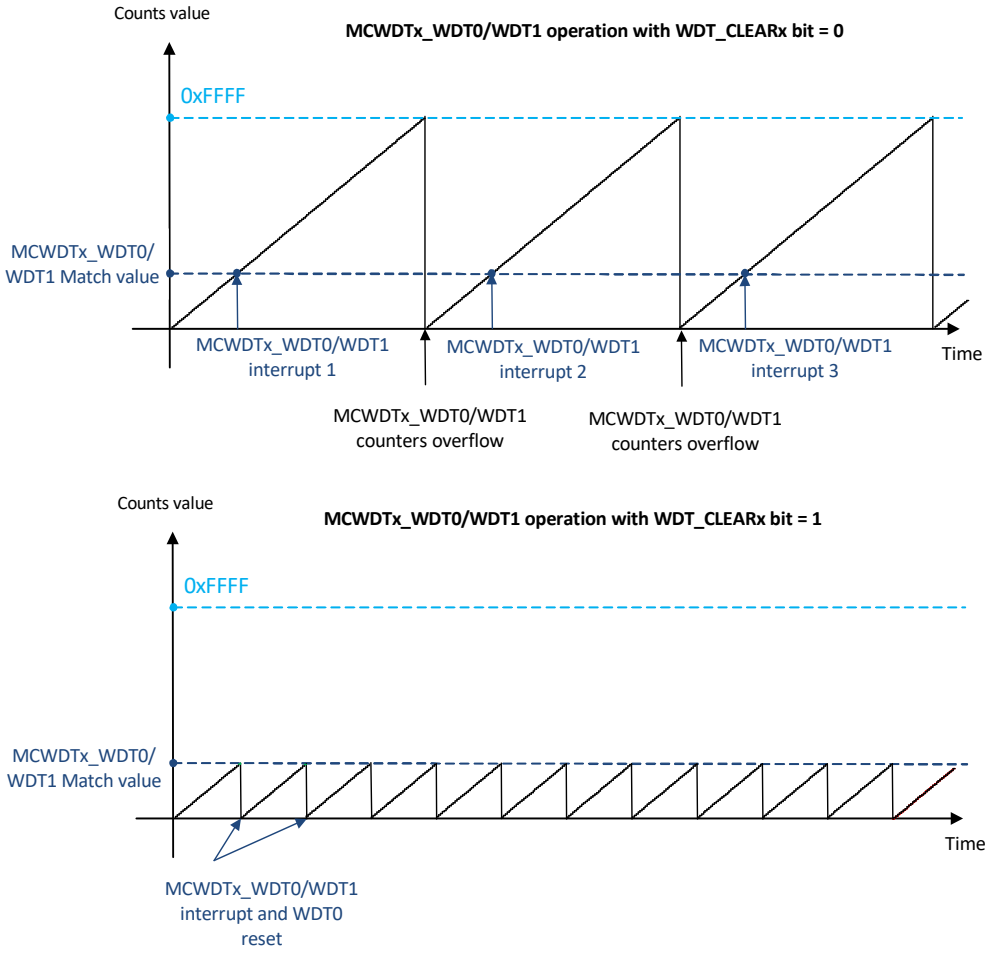


Table 24-2. MCWDTx_WDT0 and MCWDTx_WDT1 Configuration Options

Register [Bit_Pos]	Bit_Name	Description
MCWDTx_CONFIG[1:0] MCWDTx_CONFIG[9:8]	WDT_MODE0 WDT_MODE1	WDT action on a match event (WDT_CTRx == WDT_MATCHx) 0: Do nothing 1: Assert interrupt (WDT_INTx) 2: Assert device reset 3: Assert interrupt on match and a device reset on the third unhandled interrupt
MCWDTx_CONFIG[2] MCWDTx_CONFIG[10]	WDT_CLEAR0 WDT_CLEAR1	Clear the MCWDTx_WDT0/WDT1 counter on match. In other words, (WDT_-MATCHx + 1) acts similar to a period for the MCWDTx_WDT0/WDT1 counter. 0: Free-running counter 1: Clear WDT_CTRx bits on match
MCWDTx_CNTLOW[15:0] MCWDTx_CNTLOW[16:31]	WDT_CTR0 WDT_CTR1	Current counter values. Bits[15:0] contain the current value of the MCWDTx_WDT0 counter and bits[31:16] contain the current value of MCWDTx_WDT1 counter.
MCWDTx_MATCH[15:0] MCWDTx_MATCH[16:31]	WDT_MATCH0 WDT_MATCH1	Match values Changing WDT_MATCHx requires 1.5 LFCLK cycles to come into effect. After changing WDT_MATCHx, do not enter the Deep Sleep mode for at least one LFCLK cycle to ensure the WDT updates to the new setting.

24.4.1.2 *MCWDTx_WDT2 Counter Operation*

The MCWDTx_WDT2 is a 32-bit free-running counter, which can be configured to generate an interrupt. The MCWDTx_CNTHIGH register holds the current value of the MCWDTx_WDT2 counter. MCWDTx_WDT2 does not support a match feature. However, it can be configured to generate an interrupt when one of the counter bits toggle. The WDT_BITS2 bits [28:24] of the MCWDTx_CONFIG register selects the bit on which the MCWDTx_WDT2 interrupt is asserted. WDT_MODE2 bit [16] of the MCWDTx_CONFIG register decides whether to assert an interrupt on bit toggle or not. Figure 24-5 shows the MCWDTx_WDT2 counter operation.

Figure 24-5. MCWDTx_WDT2 Operation

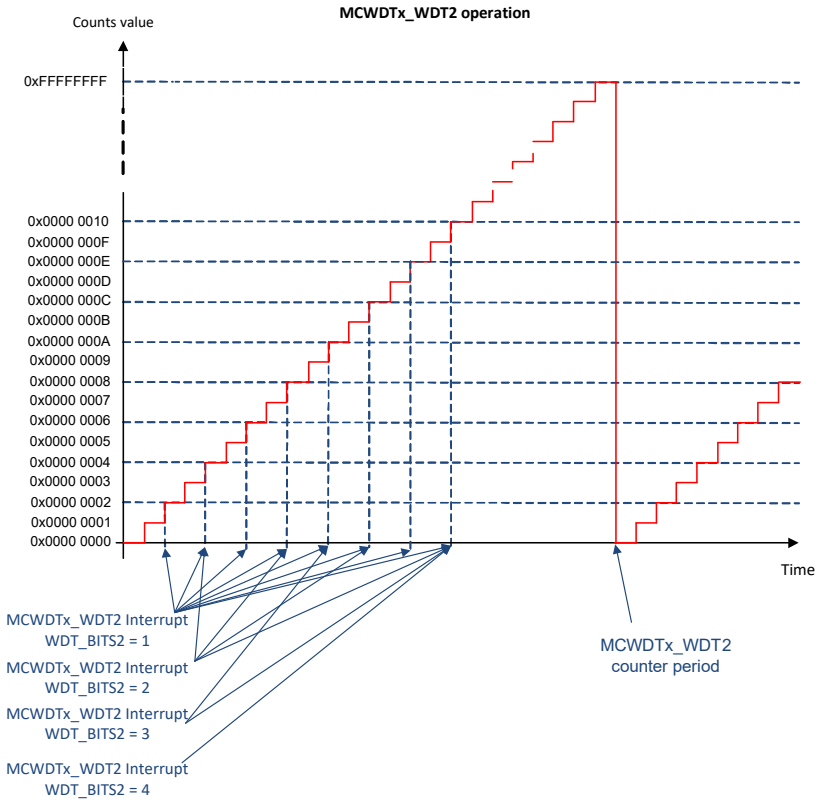


Table 24-3. MCWDTx_WDT2 Configuration Options

Register [Bit_Pos]	Bit_Name	Description
MCWDTx_CONFIG[16]	WDT_MODE2	MCWDTx_WDT2 mode 0: Free-running counter 1: Free-running counter with interrupt (WDT_INTx) request generation. The interrupts are generated when the bit position specified in WDT_BITS2 toggles in the counter value held by MCWDTx_CNTHIGH register.
MCWDTx_CONFIG[28:24]	WDT_BITS2	Bit to monitor for MCWDTx_WDT2 interrupt assertion 0: Asserts when bit [0] of MCWDTx_CNTHIGH register toggles (interrupt every tick) …… 31: Asserts when bit [31] of MCWDTx_CNTHIGH register toggles (interrupt every 2^{31} ticks)
MCWDTx_CNTHIGH[31:0]	WDT_CTR2	Current counter value of MCWDTx_WDT2

24.4.2 Enabling and Disabling WDT

The MCWDT counters are enabled by setting the WDT_ENABLEx bit in the MCWDTx_CTL register and are disabled by clearing it. Enabling or disabling a MCWDT requires 1.5 LFCLK cycles to come into effect. Therefore, the WDT_ENABLEx bit value must not be changed more than once in that period and the WDT_ENABLEDx bit of the MCWDTx_CTL register can be used to monitor enabled/disabled state of the counter.

The WDT_RESETx bit of the MCWDTx_CTL register clears the corresponding MCWDTx counter when set in firmware. The hardware clears the bit after the counter resets. This option is useful when the MCWDTx_WDT0 or MCWDTx_WDT1 is configured to generate a device reset on a match event. In such cases, the device resets when the counter reaches the match value. Thus, setting the WDT_RESET0 or WDT_RESET1 bit resets MCWDTx_WDT0 or MCWDTx_WDT1 respectively, preventing device reset.

After the MCWDT is enabled, do not write to the MCWDT configuration (MCWDTx_CONFIG) and control (MCWDTx_CTL) registers. Accidental corruption of these registers can be prevented by setting the WDT_LOCK bits [31:30] of the MCWDTx_CTL register. If the application requires updating the match value (WDT_MATCH) when the MCWDT is running, the WDT_LOCK bits must be cleared. The WDT_LOCK bits require two different writes to clear both the bits. Writing a ‘1’ to the bits clears bit 0. Writing a ‘2’ clears bit 1. Writing a ‘3’ sets both the bits and writing ‘0’ does not have any effect. Note that the WDT_LOCK bits protects only MCWDTx_CTL (except the WDT_LOCK bits), MCWDTx_CONFIG, and MCWDTx_MATCH registers. The LFCLK select registers are protected by the free-running WDT lock bits.

Table 24-4. Watchdog Configuration Options

Register [Bit_Pos]	Bit_Name	Description
MCWDTx_CTL[0] MCWDTx_CTL[8] MCWDTx_CTL[16]	WDT_ENABLE0 WDT_ENABLE1 WDT_ENABLE2	Enable MCWDT counter x 0: Counter is disabled 1: Counter is enabled
MCWDTx_CTL[1] MCWDTx_CTL[9] MCWDTx_CTL[17]	WDT_ENABLED0 WDT_ENABLED1 WDT_ENABLED2	Indicates the actual enabled/disabled state of counter x. This bit should be monitored after changing the WDT_ENABLEx bit, to receive an acknowledgment of the change
MCWDTx_CTL[3] MCWDTx_CTL[11] MCWDTx_CTL[19]	WDT_RESET0 WDT_RESET1 WDT_RESET2	Reset MCWDT counter x to 0. Hardware clears the bit when the reset is complete 0: Software - No action 1: Software - Resets the counter
MCWDTx_CTL[31:30]	WDT_LOCK	Locks or unlocks write access to the MCWDTx_CTL (except the WDT_LOCK bits), MCWDTx-_CONFIG, and MCWDTx_MATCH registers. When the bits are set, the lock is enabled. 0: No effect 1: Clears bit 0 2: Clears bit 1 3: Sets both bit 0 and 1 (lock enabled)

Note: When the watchdog counters are configured to generate an interrupt every LFCLK cycle, make sure you read the MCWDTx_INTR register after clearing the watchdog interrupt (setting the WDT_INTx bit in the MCWDTx_INTR register). Failure to do this may result in missing the next interrupt. Hence, the interrupt period becomes LFCLK/2.

24.4.3 Watchdog Cascade Options

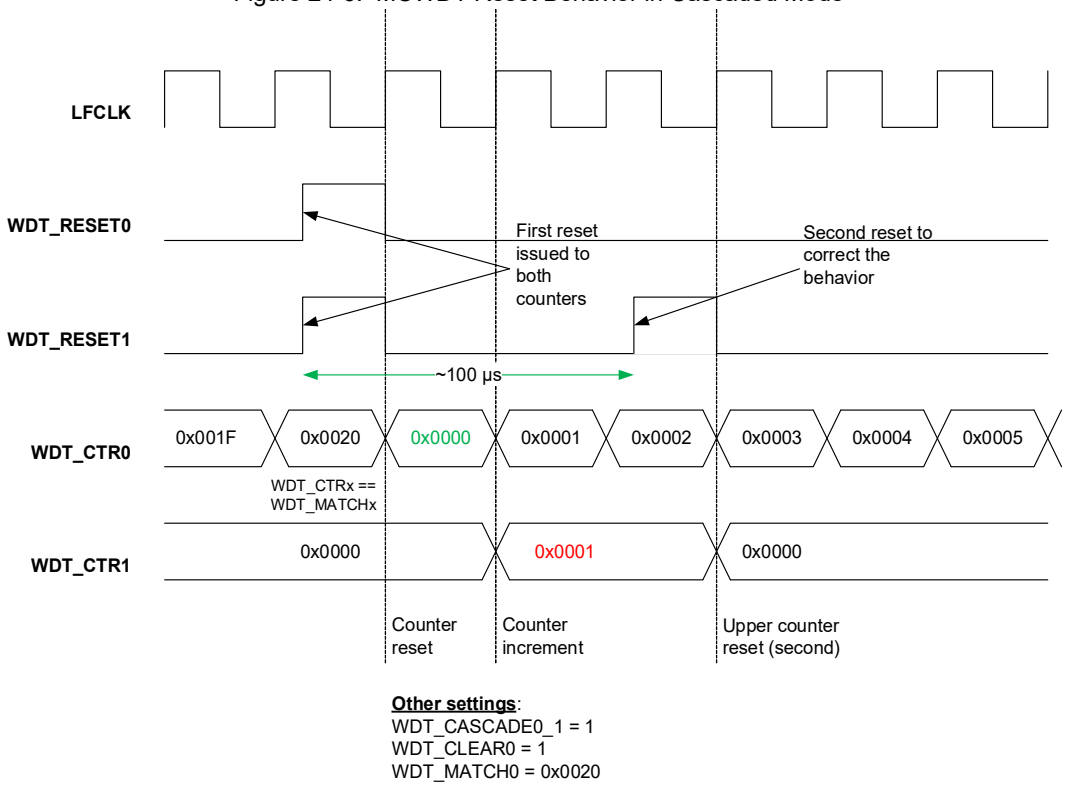
The cascade configuration shown in Figure 24-3 provides an option to increase the MCWDT counter resolution. The WDT_CASCADE0_1 bit [3] of the MCWDTx_CONFIG register cascades MCWDTx_WDT0 and MCWDTx_WDT1 and the WDT_CASCADE1_2 bit [11] of the MCWDTx_CONFIG register cascades MCWDTx_WDT1 and MCWDTx_WDT2. Note that cascading two 16-bit counters does not provide a 32-bit counter; instead, you get a 16-bit period counter with a 16-bit prescaler. For example, when cascading MCWDTx_WDT0 and MCWDTx_WDT1, MCWDTx_WDT0 acts as a prescaler for MCWDTx_WDT1 and the prescaler value is defined by the WDT_MATCH0 bits [15:0] in the MCWDTx_MATCH register. The MCWDTx_WDT1 has a period defined by WDT_MATCH1 bits [31:16] in the MCWDTx_MATCH register. The same logic applies to MCWDTx_WDT1 and MCWDTx_WDT2 cascading.

Table 24-5. Watchdog Cascade Options

Register [Bit_Pos]	Bit_Name	Description
MCWDTx_CONFIG[3]	WDT_CASCADE0_1	Cascade MCWDTx_WDT0 and MCWDTx_WDT1 0: MCWDTx_WDT0 and MCWDTx_WDT1 are independent counters 1: MCWDTx_WDT1 increments two cycles after WDT_CTR0 == WDT_MATCH0
MCWDTx_CONFIG[11]	WDT_CASCADE1_2	Cascade MCWDTx_WDT1 and MCWDTx_WDT2 0: MCWDTx_WDT1 and MCWDTx_WDT2 are independent counters 1: MCWDTx_WDT2 increments two cycles after WDT_CTR1 == WDT_MATCH1

When using cascade (WDT_CASCADE0_1 or WDT_CASCADE1_2 set), resetting the counters when the prescaler or lower counter is at its match value with the counter configured to clear on match, results in the upper counter incrementing to 1 instead of remaining at 0. This behavior can be corrected by issuing a second reset to the upper counter after approximately 100 µs from the first reset. Note that the second reset is required only when the first reset is issued while the prescaler counter value is at its match value. Figure 24-6 illustrates the behavior when MCWDTx_WDT0 and MCWDTx_WDT1 are cascaded along with the second reset timing.

Figure 24-6. MCWDT Reset Behavior in Cascaded Mode



Other settings:
WDT_CASCADE0_1 = 1
WDT_CLEAR0 = 1
WDT_MATCH0 = 0x0020

In addition, the counters exhibit non-monotonicity in the following cascaded conditions:

- If WDT_CASCADE0_1 is set, then WDT_CTR1 does not increment the cycle after WDT_CTR0 = WDT_MATCH0.
- If WDT_CASCADE1_2 is set, then WDT_CTR2 does not increment the cycle after WDT_CTR1 = WDT_MATCH1.
- If both WDT_CASCADE0_1 and WDT_CASCADE1_2 are set, then WDT_CTR2 does not increment the cycle after WDT_CTR1 = WDT_MATCH1 and WDT_CTR1 does not increment the cycle after WDT_CTR0 = WDT_MATCH0.

When cascading is enabled, always read the WDT_CTR1 or WDT_CTR2 counter value only when the prescaler counter (WDT_CTR0 or WDT_CTR1) value is not 0. This makes sure the upper counter is incremented after a match event in the prescaler counter.

24.4.4 MCDWT Reset

MCWDTx_WDT0 and MCWDTx_WDT1 can be configured to generate a device reset similar to the free-running WDT reset. Note that when the debug probe is connected, the device reset is blocked but an interrupt is generated if configured. Follow these steps to use the MCWDTx_WDT0 or MCWDTx_WDT1 counter of a MCWDTx block to generate a system reset:

1. Configure the MCWDT to generate a reset using the WDT_MODEx bits in MCWDTx_CONFIG. Configure the WDT_MODE0 or WDT_MODE1 bits in MCWDTx_CONFIG to '2' (reset on match) or '3' (interrupt on match and reset on the third unhandled interrupt).
2. Optionally, set the WDT_CLEAR0 or WDT_CLEAR1 bit in the MCWDTx_CONFIG register for MCWDTx_WDT0 or MCWDTx_WDT1 to reset the corresponding watchdog counter to '0' on a match event. Otherwise, the counters are free running. See Table 24-2 on page 289 for details.
3. Calculate the watchdog reset period such that firmware is able to reset the watchdog at least once during the period, even along the longest firmware delay path. For WDT_MODEx == 2, match value is same as the watchdog period. For WDT_MODEx == 3, match value is one-third of the watchdog period. Write the calculated match value to the WDT_MATCH register for MCWDTx_WDT0 or MCWDTx_WDT1. Optionally, enable cascading to increase the interval. **Note:** The legal value for the WDT_MATCH field is 1 to 65535.
4. For WDT_MODEx == 2, set the WDT_RESETx bit in the MCWDTx_CONFIG register to reset the WDTx counter to 0. For WDT_MODEx == 3, set the WDT_INTx bit in MCWDTx_INTR to clear any pending interrupts.
5. Enable WDTx by setting the WDT_ENABLEx bit in the MCWDTx_CTL register. Wait until the WDT_ENABLEDx bit is set.
6. Lock the MCWDTx configuration by setting the WDT_LOCK bits of the MCWDTx_CTL register.
7. In the firmware, feed (reset) the watchdog as explained in step 4.

Do not reset watchdog in the WDT ISR. It is also not recommended to use the same watchdog counter to generate a system reset and interrupt. For example, if MCWDTx_WDT0 is used to generate system reset against crashes, then MCWDTx_WDT1 or MCWDTx_WDT2 should be used for periodic interrupt generation.

24.4.5 MCWDT Interrupt

When configured to generate an interrupt, the WDT_INTx bits of the MCWDTx_INTR register provide the status of any pending watchdog interrupts. The firmware must clear the interrupt by setting the WDT_INTx. The WDT_INTx bits of the MCWDTx_INTR_MASK register mask the corresponding WDTx interrupt of the MCWDTx block to the CPU.

Follow these steps to use WDT as a periodic interrupt generator:

1. Write the desired match value to the WDT_MATCH register for MCWDTx_WDT0/WDT1 or the WDT_BITS2 value to the MCWDTx_CONFIG register for MCWDTx_WDT2. **Note:** The legal value for the WDT_MATCH field is 1 to 65535.
2. Configure the WDTx to generate an interrupt using the WDT_MODEx bits in MCWDTx_CONFIG. Configure the WDT_MODE0 or WDT_MODE1 bits in MCWDTx_CONFIG for MCWDTx_WDT0 or MCWDTx_WDT1 to '1' (interrupt on match) or '3' (interrupt on match and reset on third unhandled interrupt). For MCWDTx_WDT2, set the WDT_MODE2 bit in the MCWDTx_CONFIG register.
3. Set the WDT_INT bit in MCWDTx_INTR to clear any pending interrupt.
4. Set the WDT_CLEAR0 or WDT_CLEAR1 bit in the MCWDTx_CONFIG register for MCWDTx_WDT0 or MCWDTx_WDT1 to reset the corresponding watchdog counter to '0' on a match event.
5. Mask the WDTx interrupt to the CPU by setting the WDT_INTx bit in the MCWDTx_INTR_MASK register
6. Enable WDTx by setting the WDT_ENABLEx bit in the MCWDTx_CTL register. Wait until the WDT_ENABLEDx bit is set.
7. Enable MCWDTx interrupt to the CPU by configuring the appropriate ISER register. Refer to the Interrupts chapter on page 56.
8. In the ISR, clear the WDTx interrupt by setting the WDT_INTx bit in the MCWDTx_INTR register.

Note that interrupts from all three WDTx counters of the MCWDT block are mapped as a single interrupt to the CPU. In the interrupt service routine, the WDT_INTx bits of the

MCWDTx_INTR register can be read to identify the interrupt source. However, each MCWDT block has its own interrupt to the CPU. For details on interrupts, see the Interrupts chapter on page 56.

The MCWDT block can send interrupt requests to the CPU in Active power mode and to the WIC in Sleep and Deep Sleep power modes. It works similar to the free-running WDT.

24.5 Reset Cause Detection

The RESET_WDT bit [0] in the RES_CAUSE register indicates the reset generated by the free-running WDT. The RESET_MCWDTx bit in the RES_CAUSE register indicates the reset generated by the MCWDTx block. These bits remain set until cleared or until a power-on reset (POR), brownout reset (BOD), or external reset (XRES) occurs. All other resets leave this bit unaltered.

For more details, see the Reset System chapter on page 257.

24.6 Register List

Table 24-6. WDT Registers

Name	Description
WDT_CTL	Watchdog Counter Control Register
WDT_CNT	Watchdog Counter Count Register
WDT_MATCH	Watchdog Counter Match Register
MCWDTx_MCWDT_CNTLOW	Multi-counter WDT Sub-counters 0/1
MCWDTx_MCWDT_CNTHIGH	Multi-counter WDT Sub-counter 2
MCWDTx_MCWDT_MATCH	Multi-counter WDT Counter Match Register for counters 0 and 1
MCWDTx_MCWDT_CONFIG	Multi-counter WDT Counter Configuration, including bit toggle interrupt generation for counter 2
MCWDTx_MCWDT_CTL	Multi-counter WDT Counter Control
MCWDTx_MCWDT_INTR	Multi-counter WDT Counter Interrupt Register
MCWDTx_MCWDT_INTR_SET	Multi-counter WDT Counter Interrupt Set Register
MCWDTx_MCWDT_INTR_MASK	Multi-counter WDT Counter Interrupt Mask Register
MCWDTx_MCWDT_INTR_MASKED	Multi-counter WDT Counter Interrupt Masked Register
CLK_SELECT	Clock Selection Register
CLK_ILO_CONFIG	ILO Configuration
SRSS_INTR	SRSS Interrupt Register
SRSS_INTR_SET	SRSS Interrupt Set Register
SRSS_INTR_MASK	SRSS Interrupt Mask Register
SRSS_INTR_MASKED	SRSS Interrupt Masked Register
RES_CAUSE	Reset Cause Observation Register

25. Trigger Multiplexer Block

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

Every peripheral in the PSoC 6 MCU is interconnected using trigger signals. Trigger signals are means by which peripherals denote an occurrence of an event or a state. These triggers are used as means to affect or initiate some action in other peripherals. The trigger multiplexer block helps to route triggers from a source peripheral block to a destination.

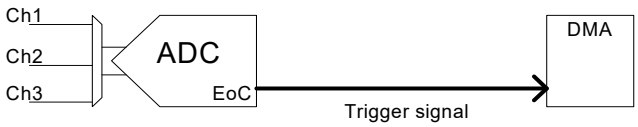
25.1 Features

- Ability to connect any trigger signal from one peripheral to another
- Supports a software trigger, which can trigger any signal in the block
- Supports multiplexing of triggers between peripherals
- One-to-one trigger paths for dedicated triggers that are more commonly routed
- Ability to configure a trigger multiplexer with trigger manipulation features in hardware such as inversion and edge/level detection
- Ability to block triggers in debug mode

25.2 Architecture

The trigger signals in the PSoC 6 MCU are digital signals generated by peripheral blocks to denote a state such as FIFO level, or an event such as the completion of an action. These trigger signals typically serve as initiator of other actions in other peripheral blocks. An example is an ADC peripheral block sampling three channels. After the conversion is complete, a trigger signal will be generated, which in turn triggers a DMA channel that transfers the ADC data to a memory buffer. This example is shown in Figure 25-1.

Figure 25-1. Trigger Signal Example

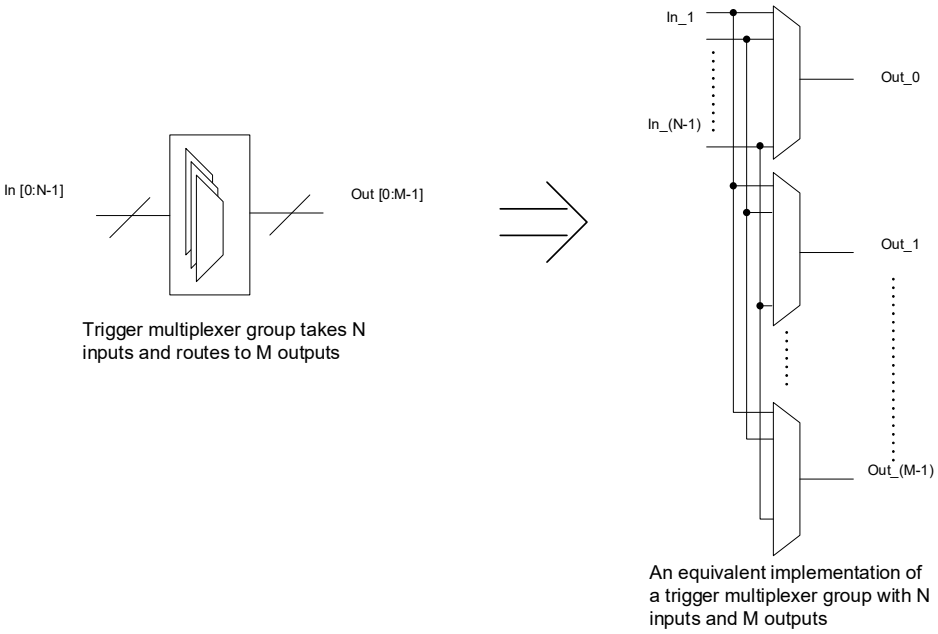


A PSoC 6 MCU has multiple peripheral bocks; each of these blocks can be connected to other blocks through trigger signals, based on the system implementation. To support this, the PSoC 6 MCU has hardware, which is a series of multiplexers used to route the trigger signals from potential sources to destinations. This hardware is called the trigger multiplexer block. The trigger multiplexer can connect to any trigger signal emanating out of any peripheral block in the PSoC 6 MCU and route it to any other peripheral to initiate or affect an operation at the destination peripheral block.

25.2.1 Trigger Multiplexer Group

The trigger multiplexer block is implemented using several trigger multiplexers. A trigger multiplexer selects a signal from a set of trigger output signals from different peripheral blocks to route it to a specific trigger input of another peripheral block. The multiplexers are grouped into trigger groups. All the trigger multiplexers in a trigger group have similar input options and are designed to feed similar destination signals. Hence the trigger group can be considered as a block that multiplexes multiple inputs to multiple outputs. This concept is illustrated in Figure 25-2. Both the trigger multiplexers and the one-to-one triggers offer trigger manipulation, which includes inversion or specifies the output signal as edge or level triggered. For edge triggering, the trigger is synchronized to the consumer block's (the peripheral block that is receiving the trigger) clock and a two-cycle pulse is generated on this clock.

Figure 25-2. Trigger Multiplexer Groups



25.2.2 One-to-one Trigger

In addition to trigger multiplexers, there are dedicated trigger paths called one-to-one triggers. These trigger paths/routes are between fixed source and destination peripherals and cannot be multiplexed. These paths can be enabled or disabled. A group of registers in the format PERI_TR_1TO1_GR[X]_TR_CTL[Y], with X being the trigger group and Y being the output trigger number from the one-to-one trigger, can be used to enable or disable the triggers path.

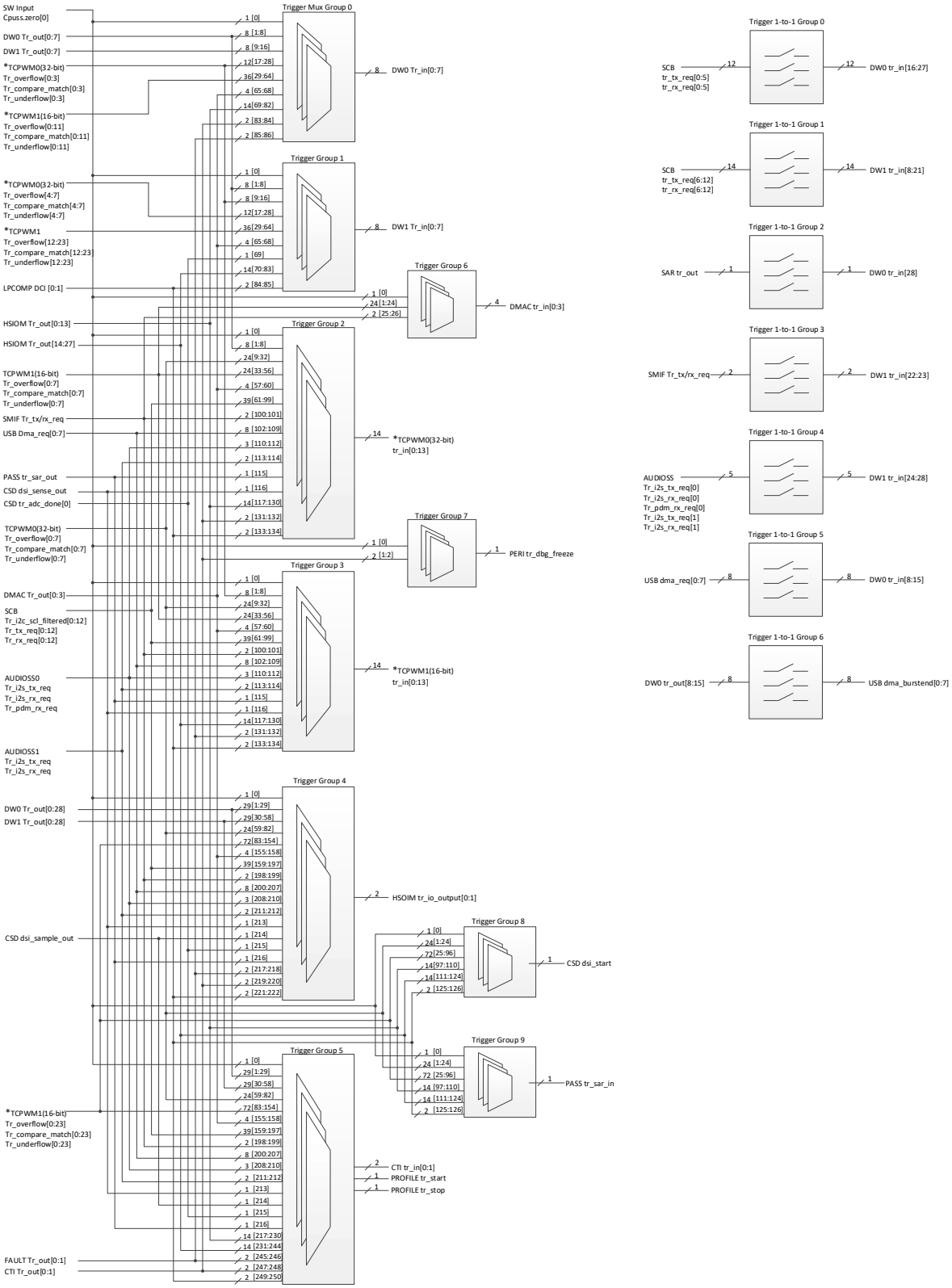
25.2.3 Trigger Multiplexer Block

The trigger multiplexer block has two types of trigger routings: trigger multiplexers that are capable of selecting from multiple trigger sources to a specific trigger destination and one-to-one triggers that are predetermined fixed trigger paths that can be enabled or disabled. See Table 25-2 for more information on each trigger group and its description.

Note: The triggers output into different peripherals, which may have more routing than is shown on the trigger routing diagram. For more information on this routing, go to the trigger destination peripheral block.

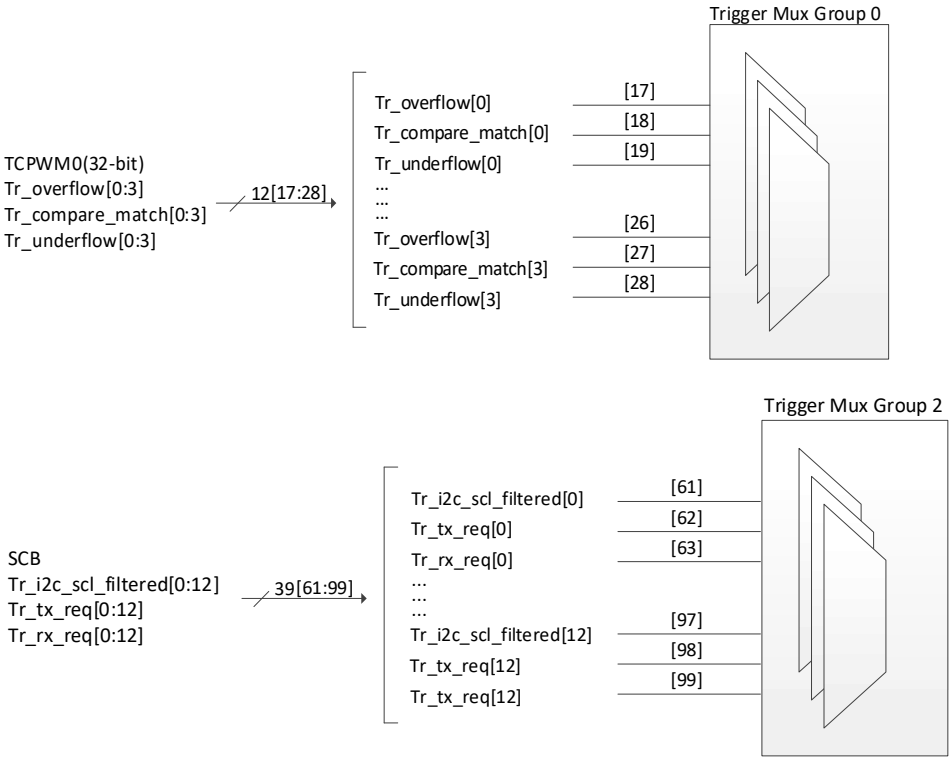
Figure 25-3 shows the trigger multiplexer architecture.

Figure 25-3. Trigger Multiplexer Block Architecture



Note: * indicates that Figure 25-4 shows details about the TCPWM and SCB inputs.

Figure 25-4. TCPWM and Multiplexer SCB Input Layout



Note: The format in these diagrams apply to all TCPWM and SCB connections in the trigger groups that have these connections. The trigger input line number will change between trigger groups and will not always start at 17 and 61 as shown in the TCPWM and SCB diagrams.

25.2.4 Software Triggers

All input and output signals to a trigger multiplexer can be triggered from software. This is accomplished by writing into the PERI_TR_CMD register. This register allows you to trigger the corresponding signal for a number of peripheral clock cycles.

The PERI_TR_CMD[GROUP_SEL] bitfield selects the trigger group of the signal being activated. The PERI_TR_CMD[OUT_SEL] bitfield determines whether the trigger signal is in output or input of the multiplexer. PERI_TR_CMD[TR_SEL] selects the specific line in the trigger group.

The PERI_TR_CMD[COUNT] bitfield sets up the number of peripheral clocks the trigger will be activated.

The PERI_TR_CMD[ACTIVATE] bitfield is set to ‘1’ to activate the trigger line specified. Hardware resets this bit after the trigger is deactivated after the number of cycles set by the PERI_TR_CMD[COUNT].

25.3 Register List

Table 25-1. Register List

Register Name	Description
PERI_TR_CMD	Trigger command register. The control enables software activation of a specific input trigger or output trigger of the trigger multiplexer structure.
PERI_TR_GR[X]_TR_OUT_CTL[Y]	This register specifies the input trigger for a specific output trigger in a trigger group. It can also invert the signal and specify if the output signal should be treated as a level-sensitive or edge-sensitive trigger. Every trigger multiplexer has a group of registers, the number of registers being equal to the output bus size from the multiplexer. In the register format, X is the trigger group and Y is the output trigger line number from the multiplexer.
PERI_TR_1TO1_GR[X]_TR_OUT_CTL[Y]	This register specifies the input trigger for a specific output trigger in a one-to-one trigger group. It also includes trigger manipulation that can invert the signal and specify if the output signal should be treated as a level-sensitive or edge-sensitive trigger. Every one-to-one trigger multiplexer has a group of registers, the number of registers being equal to the output bus size from the multiplexer. The registers are formated with X as the trigger group and Y as the output trigger line number from the multiplexer.

Table 25-2. Trigger Group

Trigger Group Number	Description
0	Routes all input trigger signals to DMA0
1	Routes all input trigger signals to DMA1
2	Routes all input trigger signals to TCPWM0(32-bit)
3	Routes all input trigger signals to TCPWM1(16-bit)
4	Routes all input trigger signals to HSIOM
5	Routes all input trigger signals to CPUSS CTI and the Profiler
6	Routes all input trigger signals to DMAC
7	Routes all input trigger signals to trigger freeze operation
8	Routes all input trigger signals to CSD ADC start
9	Routes all input trigger signals to PASS ADC start
Trigger 1-to-1 Group	
0	Connects SCB triggers to DMA0
1	Connects SCB triggers to DMA1
2	Connects SAR triggers to DMA0
3	Connects SMIF triggers to DMA1
4	Connects AUDIOSS triggers to DMA1
5	Connects USB triggers to DMA0
6	Connects DMA0 to USB

26. Profiler

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU Profiler provides counters that can measure duration or number of events of a particular peripheral. Functions such as DMA transfers or buffered serial communications can happen asynchronously, and are not directly tied to the CPU or code execution. The profiler provides additional insight into the device so you can identify an asynchronous activity that could not be monitored previously.

The profiler manages a set of counters. You configure an available counter to monitor a particular source. Depending on the nature of the source, you count either duration (reference clock cycles) or the number of events.

The ability to monitor specific peripherals enables you to understand and optimize asynchronous hardware, including:

- Identify the activity of a particular peripheral
- Identify asynchronous activity

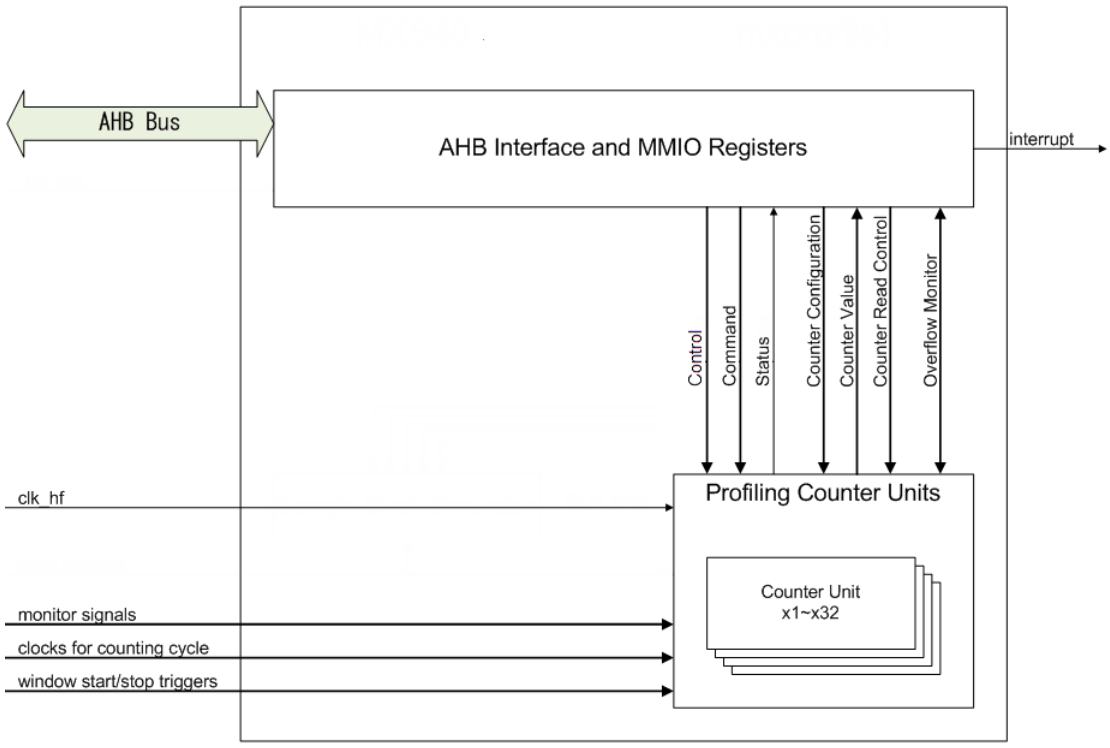
26.1 Features

The profiler has these features and capabilities:

- A variety of sources you can monitor (see Table 26-1)
- Support for eight counters, to monitor up to eight sources simultaneously
- For each counter you specify:
  - what source to monitor (can be changed dynamically if required)
  - what to measure (duration or events)
  - what reference clock to use for the count (only affects duration)
- Provides ability to easily detect active peripherals that may be difficult to measure by external monitoring
- Provides an absolute count of events or reference clock cycles for each monitored source

26.2 Architecture

Figure 26-1. Profiler Block Diagram



The profiler supports up to 32 counters. The actual number of counters is hardware dependent. This device supports up to eight counters.

26.2.1 Profiler Design

With the profiler you can monitor:

- Peripherals that operate asynchronously to software
- CPU activity

This includes flash access, DMA, and other processes. See Available Monitoring Sources on page 301.

You can measure total energy consumption directly using external hardware. The profiler can monitor internal hardware with no external probe or measuring device.

The profiler counts either the amount of time that a source is active (duration) or the number of events that occur. The profiler does not report the actual amount of energy consumed.

Your application may use a system resource or peripheral that is not among the profiler sources. A complete understanding of your application's energy consumption requires that you profile those parts of your code that use those other systems. For example, you could determine the time the peripheral is in use, and then derive energy use based on power consumption data from the data sheet, or external power monitoring.

All counters that are in use start and stop at the same time. The time between start and stop is called the profiling window, or the profiling session. During the profiling session, each counter increments at its own rate based on the monitored signal and (if measuring duration) the selected reference clock. See Start and Stop Profiling on page 303.

Before starting a profiling session, configure the counters you want to use. See Configure and Enable a Counter on page 303.

You can get interim results during a profiling session, or final results after you stop profiling. See Get the Results on page 304.

26.2.2 Available Monitoring Sources

You specify which source a counter monitors. The available sources vary per device series. You can find constants in a header file named *<series>_config.h* (part of the Peripheral Driver Library). For example, Table 26-1 lists the sources in *psoc63_config.h*.

Table 26-1. Available Signals to Monitor

Value	Symbol	Count	Description
0	PROFILE_ONE	Duration	Constant One
1	CPUSS_MONITOR_CM0	Events	CM0+ active cycle count
2	CPUSS_MONITOR_CM4	Events	CM4 active cycle count
3	CPUSS_MONITOR_FLASH	Events	Flash read count
4	CPUSS_MONITOR_DW0_AHB	Events	DW0 AHB transfer count (DMA transfer)
5	CPUSS_MONITOR_DW1_AHB	Events	DW1 AHB transfer count (DMA transfer)
6	CPUSS_MONITOR_CRYPTO	Events	Crypto memory access count
7	USB_MONITOR_AHB	Events	USB AHB transfer count
8	SCB0_MONITOR_AHB	Events	SCB 0 AHB transfer count
9	SCB1_MONITOR_AHB	Events	SCB 1 AHB transfer count
10	SCB2_MONITOR_AHB	Events	SCB 2 AHB transfer count
11	SCB3_MONITOR_AHB	Events	SCB 3 AHB transfer count
12	SCB4_MONITOR_AHB	Events	SCB 4 AHB transfer count
13	SCB5_MONITOR_AHB	Events	SCB 5 AHB transfer count
14	SCB6_MONITOR_AHB	Events	SCB 6 AHB transfer count
15	SCB7_MONITOR_AHB	Events	SCB 7 AHB transfer count
16	SCB8_MONITOR_AHB	Events	SCB 8 AHB transfer count
17–20	Reserved		
21	SMIF_MONITOR_SMIF_SPI_SELECT0	Duration	SPI select to memory 0 active
22	SMIF_MONITOR_SMIF_SPI_SELECT1	Duration	SPI select to memory 1 active
23	SMIF_MONITOR_SMIF_SPI_SELECT2	Duration	SPI select to memory 2 active
24	SMIF_MONITOR_SMIF_SPI_SELECT3	Duration	SPI select to memory 3 active
25	SMIF_MONITOR_SMIF_SPI_SELECT_ANY	Duration	SPI select to any of the memories 0-3 active

26.2.3 Reference Clocks

Each monitored source has its own clock reference, called the sample clock. Table 26-2 lists the six choices for the sample clock. You may use one of two CLK_PROFILE sources, or one of four CLK_REF sources.

The counter units of the profiler are clocked using either CLK_HF or CLK_PERI as CLK_PROFILE. You configure CLK_HF in system resources and CLK_PERI in the CPU subsystem. See Clocking System chapter on page 242 for details on clocks. In addition, there are four available reference clocks: CLK_TIMER, CLK_IMO, CLK_ECO, and CLK_LF. Any of the six clock sources can be used as the sample clock.

If you measure events, the sample clock is selected automatically to be either CLK_HF or CLK_PERI depending on the monitored source. The count represents the number of events that occurred during the profiling session. The profiler counts the edges of the monitored signal, using either CLK_HF or CLK_PERI to detect edges.

If you measure duration, you specify the sample clock from one of the six choices listed in Table 26-2. When you measure duration, while the monitored source is active, the counter increments at each cycle of the sample clock. The profiler synchronizes reference clocks with CLK_PROFILE. As a result the four available reference clocks cannot exceed CLK_PROFILE/2.

To measure duration accurately, the sample clock should have a stable frequency throughout the profiling session. If your application remains in active power states, you can use CLK_HF as the sample clock. CLK_HF gives you the greatest resolution in your results.

The profiler can be enabled in CPU Active and Sleep modes. However, clock frequencies can change based on the power state of the application. The source clock frequency should be stable throughout the profiling session to ensure reliable data.

High-frequency clocks are not available in System Deep Sleep and Hibernate power modes, so the profiler is disabled. The configuration registers maintain state through Deep Sleep. Profiler data is lost (registers are not maintained) in Hibernate mode. See the Device Power Modes chapter on page 225.

Table 26-2. Available Sample Clock Sources for the Profiler

Value	Clock Source	Symbol	Clock Type
0	Timer	CLK_TIMER	CLK_REF
1	Internal main oscillator	CLK_IMO	CLK_REF
2	External crystal oscillator	CLK_ECO	CLK_REF
3	Low frequency clock	CLK_LF	CLK_REF
4	High-frequency clock	CLK_HF	CLK_PROFILE
5	Peripheral clock	CLK_PERI	CLK_PROFILE

26.3 Using the Profiler

This section describes the steps required to use the profiler effectively. To use the profiler, you add instrumentation code to your application, to set or read values in particular registers. See the registers TRM for details.

At the highest level you perform these tasks:

- Enable the profiling block
- Configure and enable counters
- Start and stop profiling
- Handle counter overflow
- Get the results
- Exit gracefully

You can monitor as many sources as there are available counters. The actual number of available counters is hardware dependent. The value is defined in the symbol `PROFILE_PRFL_CNT_NR` in the *<series>_config.h* file. For example, the *psoc63_config.h* file defines this value as '8'.

Instead of manipulating registers directly, you can use the Peripheral Driver Library (PDL). The PDL provides a software API to manage the profiler and an interrupt to handle counter overflow. The software driver maintains an array of data for each counter, including the source you want to monitor, whether you are monitoring duration or events, the reference clock, and an overflow count for each counter. It provides function calls to configure and enable a counter, start and stop a profiling session, and calculate the results for each counter. The PDL API handles all register and bit access.

Refer to the PDL API Reference for more information. The PDL installer puts the documentation here:
`<PDL directory>\doc\pdl_api_reference_manual.html`

26.3.1 Enable or Disable the Profiler

Before performing any operations, enable the profiling block. See Table 26-3. This does not enable individual counters or start a profiling session.

Table 26-3. Enabling the Profiler

Task	Register	Bitfield	Value
Enable the profiler	PROFILE_CTL	ENABLED	0 = disabled; 1 = enabled

26.3.2 Configure and Enable a Counter

Each counter has a PROFILE_CTL register that holds configuration information. For each counter you specify:

- The source you want to monitor
- What you want to measure (events or duration)
- The reference clock for the counter (affects only duration)

You also enable each counter individually. Only the enabled counters will start when START_TR is set to one. See Table 26-4.

Table 26-4. Configuring and Enabling a Counter

Task	Register	Bitfield	Value
Specify source	PROFILE_CTL	MON_SEL	Table 26-1 on page 301
Specify events or duration	PROFILE_CTL	CNT_DURATION	0 = events; 1 = duration. See Table 26-1 on page 301
Specify the reference clock	PROFILE_CTL	REF_CLK_SEL	Table 26-2 on page 302
Enable the counter	PROFILE_CTL	ENABLED	0 = disabled; 1 = enabled

The count value is a 32-bit register. If that is not a sufficiently large number for your purposes, you must also enable the profiling interrupt for the counter. See Handle Counter Overflow on page 303.

When gathering data, you may want to know the actual number of clock cycles that occurred during the profiling session. To get this number, configure a counter to use:

- PROFILE_ONE as the monitored source
- duration (not events)
- a reference clock

The count for that counter is the actual number of reference clock cycles that occurred during the profiling session.

26.3.3 Start and Stop Profiling

After configuring and enabling the counters you want to use, you can start and stop a profiling session. You may wish to ensure that all counters are set to zero before beginning the profiling session. The PROFILE_CMD register applies to all counters, meaning that all enabled counters will be started at the same time. See Table 26-5.

Table 26-5. Starting or Stopping a Profiling Session

Task	Register	Bitfield	Value
Clear all counters	PROFILE_CMD	CLEAR_ALL_CNT	1 = reset all counters to zero
Start profiling	PROFILE_CMD	START_TR	1 = start profiling
Stop profiling	PROFILE_CMD	STOP_TR	1 = stop profiling

By design, all counters that are in use start and stop simultaneously. During the profiling session, each counter increments at its own rate based on the monitored signal and (if measuring duration) the reference clock.

26.3.4 Handle Counter Overflow

Each profiling counter is a 32-bit number. If this is not sufficient for your purposes, then you must handle counter overflow. To do so you must:

- Enable the profiling interrupt for the particular counter
- Provide an interrupt handler

For example, your interrupt handler could maintain an overflow counter for each profiling counter. See the Interrupts chapter on page 56 for details about interrupts.

Counter overflow involves the interaction of three registers, INTR, INTR_MASK, and INTR_MASKED. Each of these registers has one bit per profiling counter.

When an overflow occurs for a particular counter, hardware sets the corresponding bit in the INTR register. You typically do not read this register.

To enable the profiling interrupt for a particular counter, set the corresponding bit in the INTR_MASK register.

When an interrupt occurs, your interrupt handler reads the bits in the INTR_MASKED register. This register reflects a bitwise AND between the INTR register (an overflow interrupt has occurred for a particular counter) and the INTR_MASK register (this counter has the interrupt enabled). When a bit is set in the INTR_MASKED register, the corresponding counter is enabled and has experienced an overflow.

You can artificially trigger an overflow interrupt to test your code. The INTR_SET register also has one bit per counter. For debug purposes, software can set the appropriate bit to activate a specific overflow interrupt. This enables debug of the interrupt without waiting for hardware to cause the interrupt.

26.3.5 Get the Results

For each of your enabled counters, read the counter value in the CNT register for that counter. This is your absolute count. If you are tracking counter overflow, then the absolute count is 0x1 0000 0000 * overflow count + counter value.

The common use case is to get results after you stop profiling. However, you can get a snapshot of the results without stopping the profiler. In this case, however, the profiler is still counting and the results are changing as you gather the data.

After completing a profiling session, you may wish to repeat the same profile. Clear all counters to zero, and then start and stop profiling. See Start and Stop Profiling on page 303. If you are handling counter overflow, set your overflow counters to zero as well.

You can also reconfigure any or all counters to gather different data. See Configure and Enable a Counter on page 303. Reconfigure the profiler, and start another profiling session.

26.3.6 Exit Gracefully

When finished, you should disable the profiler. To do this make sure that you:

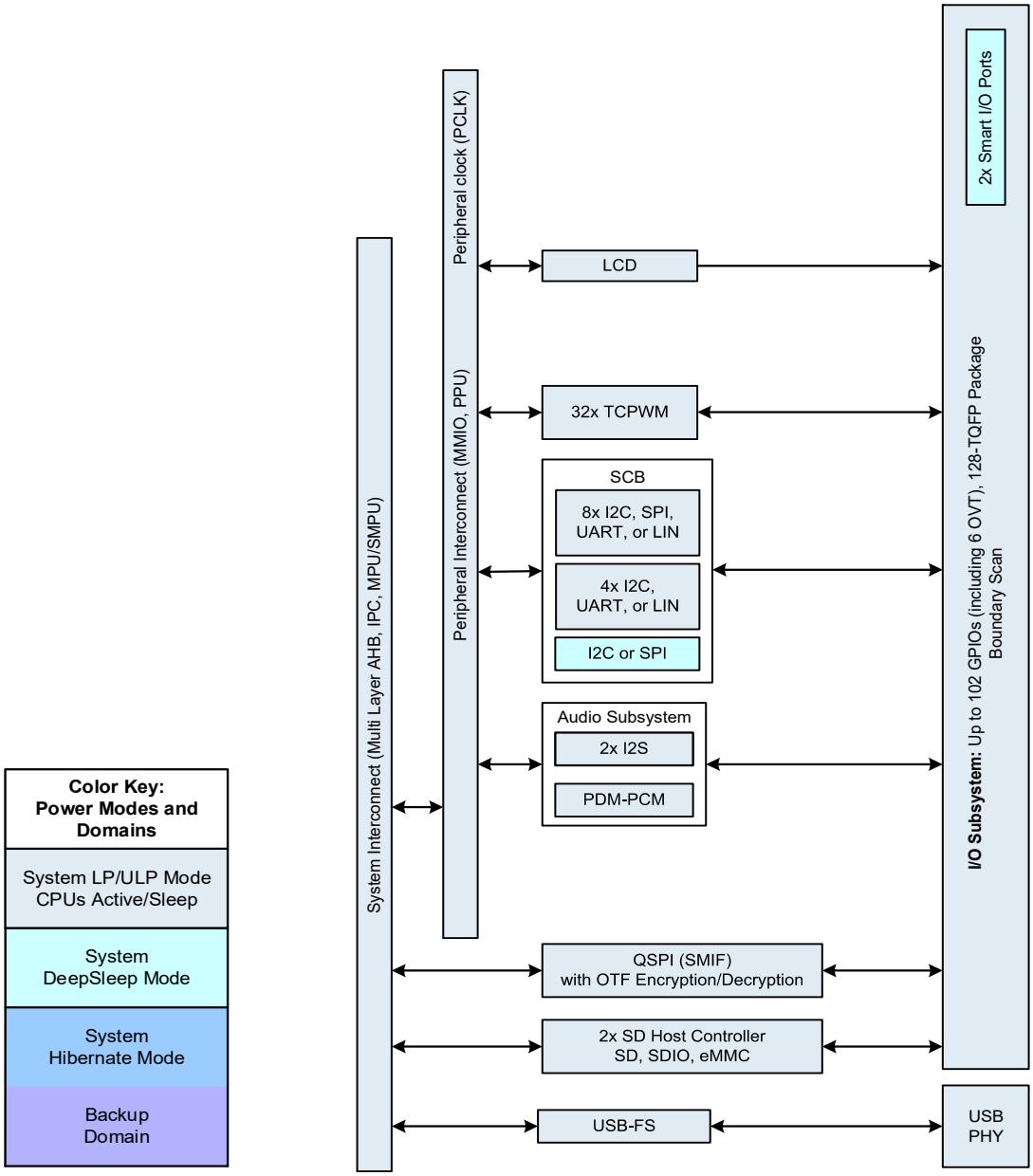
- Stop profiling (see Start and Stop Profiling on page 303)
- Clear any profiling configuration (see Configure and Enable a Counter on page 303)
- Disable the profiling interrupt if you have set it up (see Handle Counter Overflow on page 303)
- Disable the profiler itself (see Enable or Disable the Profiler on page 302)

Section D: Digital Subsystem

This section encompasses the following chapters:

Top Level Architecture

Figure D-1. Digital System Block Diagram



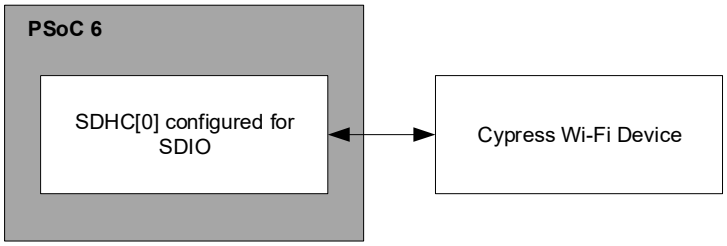
27. Secure Digital Host Controller (SDHC)

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The secure digital host controller (SDHC) in the PSoC 6 MCU allows interfacing with embedded multimedia card (eMMC)-based memory devices, secure digital (SD) cards, and secure digital input output (SDIO) cards. The block supports all three interfaces – SD, SDIO, and eMMC. The block can also work with devices providing SDIO interface, such as Cypress' WiFi products (for example, CYW4343W). Figure 27-1 illustrates a typical application using the SDHC block.

Figure 27-1. Typical SDHC Application



27.1 Features

- Complies with eMMC 5.1, SD 6.0, and SDIO 4.10 standards
- Supports host controller interface (HCI) 4.2 shared by eMMC and SD
- SD interface supports 1-bit and 4-bit bus interfaces, and the following speed modes. The specified data rate is for a 4-bit bus.
  - 3.3-V signal voltage: Default speed (12.5 MB/s at 25 MHz) and high speed (25 MB/s at 50 MHz)
  - UHS-I modes using 1.8-V signal voltage: SDR12 (12.5 MB/s at 25 MHz), SDR25 (25 MB/s at 50 MHz), SDR40 (40 MB/s at 80 MHz), and DDR40 (40 MB/s at 40 MHz)
- eMMC interface supports 1-bit and 4-bit bus interfaces, and the following speed modes. The specified data rate is for a 4-bit bus.
  - Legacy (13 MB/s at 26 MHz), high-speed SDR (26 MB/s at 52 MHz), and high-speed DDR (52 MB/s at 52 MHz)
- Supports three DMA modes – SDMA, ADMA2, and ADMA3 – through a dedicated DMA engine
- Provides 1KB SRAM for buffering up to two 512-byte blocks
- Provides I/O interfaces for bus interface voltage selection (3.3 V/1.8 V) and for power enable/disable
- Provides I/O interfaces for functions such as card detection, mechanical write protection, eMMC card reset, and LED control

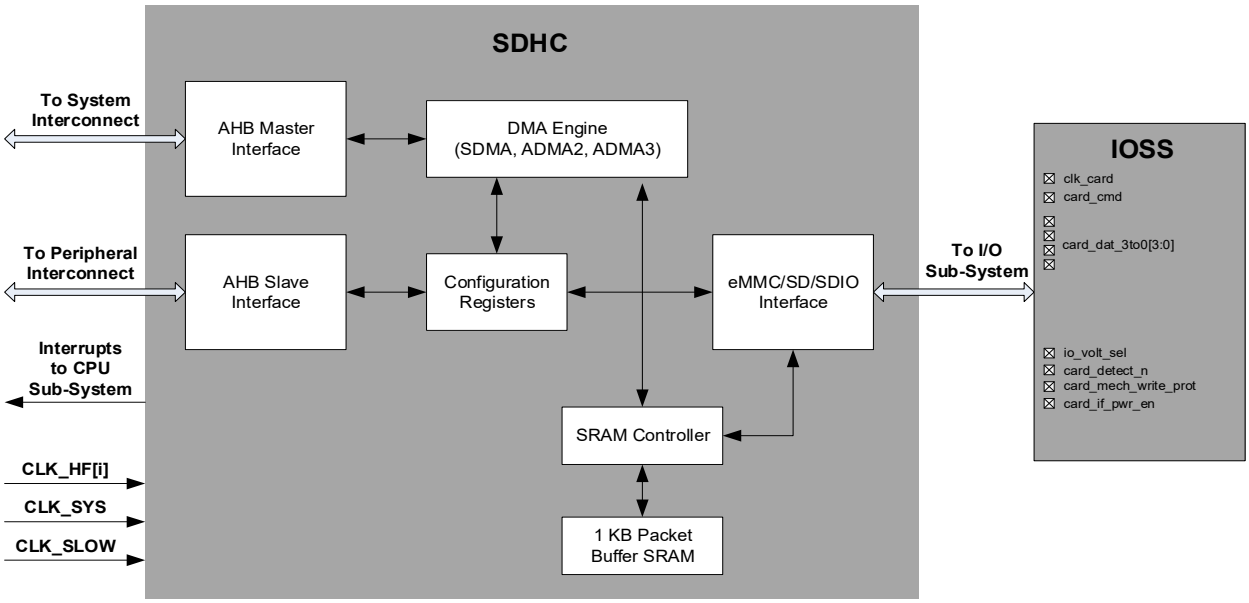
27.1.1 Features Not Supported

The SDHC block does not support the following features.

- SD/SDIO operation in UHS-II mode
- Command queuing engine (CQE)
- eMMC boot operation in dual data rate mode
- Read wait operation by DAT[2] signaling in an SDIO card
- Suspend/resume operation in an SDIO card
- Interrupt input pins for embedded SD systems
- SPI protocol mode of operation

27.2 Block Diagram

Figure 27-2. SDHC Block Diagram



The SDHC controller supports all three interfaces – SD, SDIO, and eMMC; it supports up to 4-bit bus width. The AHB master interface helps to transfer data to and from the system memory and the AHB slave interface provides access to the configuration registers. The register set comprises the standard SD host controller interface (HCI) registers as specified in the SD Specifications Part A2 SD Host Controller Standard Specification. These registers are described in the registers TRM. The DMA engine handles direct data transfer between the SDHC logic and system memory. It supports SDMA, ADMA2, and ADMA3 modes based on the configuration.

The SDHC block complies with the following standards. Refer to the specifications documents for more information on the protocol and operations.

- SD Specifications Part 1 Physical Layer Specification Version 6.00
- SD Specifications Part A2 SD Host Controller Standard Specification Version 4.20
- SD Specifications Part E1 SDIO Specifications Version 4.10
- Embedded Multi-Media Card (eMMC) Electrical Standard 5.1

27.3 Clocking

Table 27-1 lists the different clocks used in the SDHC block. While configuring the clock for SDHC make sure that clk_slow ≥ clk_sys ≥ clk_card.

Table 27-1. Clocks in SDHC

Source	SDHC Clock	Function
CLK_SLOW	Core SDHC Clock	Used for core SDHC functions including the packet buffer SRAM; it is sourced from the slow clock (clk_slow); it must be ≥ AHB slave clock.
	AHB Master Interface Clock	Used by the AHB master interface; it is sourced from the slow clock (clk_slow); it must be ≥ AHB slave clock.
CLK_SYS	AHB Slave Interface Clock	Used by the AHB slave interface; it is clocked by the PERI group clock (clk_sys); it must be ≥ clk_card. The group clock is derived from the PERI clock (clk_peri) using a divider. Because this divider can remain at the default value of '1' for most applications, clk_peri can be considered as clk_sys for SDHC. See Clocking System chapter on page 242 for information on clk_sys and clk_peri.
CLK_HF[i]	Base Clock/Card Clock	Used for sourcing the SD/eMMC interface clock (clk_card); it is derived from CLK_HF[i]; it must be set to 100 MHz to be compatible with the Capabilities register. See 27.3.2 Base Clock (CLK_HF[i]) Configuration for details. See High-Frequency Root Clocks on page 251 to know which CLK_HF[i] drives an SDHC instance.
	Timer Clock	Used for command and data timeout functions; it is derived from CLK_HF[i].

27.3.1 Clock Gating

All the clocks except the slave interface clock can be gated internally to enter standby mode (See Power Modes on page 310). In standby mode, you can also stop the clocks externally if required. The slave clock cannot be gated because it is used for wakeup logic (see Interrupts to CPU on page 310) during the standby mode.

The card clock is gated by clearing the SD_CLK_EN bit and other clocks are gated by clearing the INTERNAL_CLK_EN bit of the CLK_CTRL_R register. See Clock Setup on page 316 for the sequence to be followed while modifying this bit.

27.3.2 Base Clock (CLK_HF[i]) Configuration

The HCI register (Capabilities) has a read-only field (BASE_CLK_FREQ) to indicate the base clock frequency so that an SD HCI-compatible driver can easily configure the divider for the required bus speed. This value is set to 0x64 (100 MHz) and hence CLK_HF[i] must be set to 100 MHz. If this compatibility is not required, CLK_HF[i] can be set to any value. See 27.3.4 Timeout (TOUT) Configuration.

27.3.3 Card Clock (SDCLK) Configuration

The SDCLK or card clock frequency is set by configuring the 10-bit divider in the CLK_CTRL_R register and selecting the 10-bit divided clock mode by clearing the CLK_GEN_SELECT bit of the same register. The default value of this bit is zero. The UPPER_FREQ_SEL field holds the upper two bits (9:8) and the FREQ_SEL field holds the lower eight bits (7:0) of the divider. Base clock frequency is sourced from CLK_HF[i] as explained in Table 27-1. SDCLK frequency is equal to base clock frequency when the divider value is zero.

SDCLK Frequency = Base Clock Frequency / (2 × 10-bit divider value)

These fields are set automatically, based on the selected Bus Speed mode, to a value specified in one of the preset registers when HOST_CTRL2_R.PRESET_VAL_ENABLE is set. The preset registers are selected according to Table 27-2.

27.3.4 Timeout (TOUT) Configuration

An internal timer is used for command and data timeouts. The timeout value is specified through the TOUT_CTRL_R.TOUT_CNT register field. The timer clock (TMCLK) frequency indicated by the read-only fields TOUT_CLK_FREQ and TOUT_CLK_UNIT of CAPABILITIES1_R register is 1 MHz. Timer clock is derived by dividing the CLK_HF[i], which means that CLK_HF[i] must be set to 100 MHz to be compatible with the Capabilities register.

27.4 Bus Speed Modes

The SDHC block can operate in either SD/SDIO mode or in eMMC mode. The SDHC block operates in eMMC mode when the EMMC_CTRL_R.CARD_IS_EMMC bit is set; otherwise, it operates in SD/SDIO mode. The speed mode is configured through HOST_CTRL1_R and HOST_CTRL2_R registers as per Table 27-2. The HOST_CTRL2_R.UHS2_IF_ENABLE bit should remain at its default value of zero because the block does not support UHS-II mode. The card clock must be configured according to the selected speed mode through the CLK_CTRL_R register. See 27.3.3 Card Clock (SDCLK) Configuration for more information.

Table 27-2. Bus Speed Mode Configuration

Bus Speed Mode	HOST_CTRL1_R Field	HOST_CTRL2_R Fields		Selected Preset Register
	HIGH_SPEED_EN	SIGNALING_EN	UHS_MODE_SEL	
SD Default Speed (DS)	0	0	Don't care	PRESET_DS_R
SD High Speed (HS)	1	0	Don't care	PRESET_HS_R
SDR12/eMMC Legacy	Don't care	1	000b	PRESET_SDR12_R
SDR25/eMMC High Speed	Don't care	1	001b	PRESET_SDR25_R
SDR50	Don't care	1	010b	PRESET_SDR50_R
DDR50/eMMC High Speed DDR	Don't care	1	100b	PRESET_DDR50_R

27.5 Power Modes

The block can operate during active and sleep system power modes. It does not support deep sleep mode and cannot wake up from events such as card insertion and removal when the system is in deep sleep. All the core registers except the packet buffer SRAM are retained when the system enters deep sleep mode and the SRAM is switched off to save power. Retention is performed so that the block can resume operation immediately after wakeup from deep sleep without requiring reconfiguration. Make sure that no AHB traffic (such as register read/write and DMA operation) is present, the SD/SDIO/eMMC bus interface is idle, and no data packets are pending in the packet buffer SRAM when the system transitions into deep sleep mode.

27.5.1 Standby Mode

The block can be put into standby mode to save power during the active and sleep system power modes by turning off the clocks. See Clock Gating on page 309 for details. The block can detect wakeup interrupts (see Interrupts to CPU on page 310) in standby mode.

27.6 Interrupts to CPU

The block provides two interrupt signals to CPUSS:

- Wakeup Interrupt Signal – Triggered on events such as card insertion, removal, and SDIO card interrupt. This interrupt source cannot wake up the system from deep sleep mode and is provided so that a host driver can take appropriate action on those events. For example, resuming operation from standby mode on card insertion. See 27.5.1 Standby Mode for details. As card insertion and removal is not applicable to an embedded device, wakeup interrupt should not be used in this case. However it can still be used for SDIO card interrupt.
- General Interrupt Signal – Triggered on all other events, in either normal conditions or error conditions.

A host driver must not enable the wakeup and general interrupt signals at the same time.

To use only the wakeup interrupt signal, clear the NORMAL_INT_STAT_R and NORMAL_INT_SIGNAL_EN_R registers, and then set the enable bits of the required wakeup events in the WUP_CTRL_R and NORMAL_INT_STAT_EN registers.

To use only the general interrupt signal, clear the WUP_CTRL_R and NORMAL_INT_STAT_R registers. Then, set the required bits in NORMAL_INT_SIGNAL_EN_R and NORMAL_INT_STAT_EN registers.

These interrupts remain asserted until the CPU clears the interrupt status through one of the status registers – NORMAL_INT_STAT_R and ERROR_INT_STAT_R.

The SDIO card interrupt status bit, CARD_INTERRUPT, is a read-only bit. The host driver may clear the NORMAL_INT_STAT_EN_R.CARD_INTERRUPT_STAT_EN bit before servicing the SDIO card interrupt and may set this bit again after all interrupt requests from the card are cleared to prevent inadvertent interrupts.

Following is the list of registers used in interrupt configuration.

Table 27-3. Interrupt Control Registers

Register	Description
WUP_CTRL_R	Enables or disables different wakeup interrupts. Host driver must maintain voltage on the SD bus by setting PWR_CTRL_R.SD_BUS_PWR_VDD1 bit for these interrupts to occur. These interrupts cannot wakeup the device from deep sleep.
NORMAL_INT_STAT_R	Reflects the status of wakeup interrupts and non-error general interrupts. It also has a bit to indicate whether any of the bits in ERROR_INT_STAT_R is set.
ERROR_INT_STAT_R	Reflects the status of general interrupts that are triggered by error conditions.
NORMAL_INT_STAT_EN_R	Provides mask bits for wakeup interrupts and non-error general interrupts.
ERROR_INT_STAT_EN_R	Provides mask bits for general interrupts that are triggered by error conditions.
NORMAL_INT_SIGNAL_EN_R	Setting any of these bits to '1' enables interrupt generation for wakeup interrupts and non-error general interrupts.
ERROR_INT_SIGNAL_EN_R	Setting any of these bits to '1' enables interrupt generation for general interrupts that are triggered by error conditions.
FORCE_ERROR_INT_STAT_R	Forces an error interrupt to occur when the corresponding bit is set.

27.6.1 SDIO Interrupt

The SDIO interrupt function is supported on card_dat_3to0[1] line (SD pin 8). See the SDIO specifications for details on this feature. The CARD_INTERRUPT_STAT_EN bit in the NORMAL_INT_STAT_EN_R register and the CARD_INTERRUPT_SIGNAL_EN bit in the NORMAL_INT_SIGNAL_EN_R register must be set to enable this interrupt. To use this interrupt as wakeup interrupt, use WUP_CTRL_R.WUP_CARD_INT instead of NORMAL_INT_SIGNAL_EN_R.

27.7 I/O Interface

SDHC block provides the signals shown in Table 27-4, which can be routed to pins through the I/O subsystem (IOSS). Refer to the I/O System chapter on page 261 to configure the I/Os, and the PSoC 61 datasheet/PSoC 62 datasheet for specific pins available for each signal. SDHC also supports SDIO interrupt on DAT[1] line (card_data_3to0[1]). The output signals must be configured in strong drive mode, bi-directional signals in strong drive with the input buffer ON, and the input pins in high-impedance mode when an external pull-up resistor is available; otherwise, they must be configured in internal pull-up mode. Input buffer must be enabled for the input pins. The drive mode of the DAT lines must be set to high impedance after card removal. See Card Detection on page 314 for details. In addition to configuring the drive mode and HSIOM registers in IOSS, the SDHC_CORE.GP_OUT_R register must be configured to enable the required signals. See Table 27-4. The card_detect_n and card_write_prot should be connected to ground if an eMMC or an embedded SDIO device is connected.

Table 27-4. I/O Signal Interface

Signal	Function	Register Configuration
clk_card	Clock output	GP_OUT_R.CARD_CLOCK_OE
card_cmd	Command (bi-directional)	Always enabled
card_dat_3to0[3:0]	Data (bi-directional)	HOST_CTRL1_R.DAT_XFER_WIDTH
card_detect_n	Card detect signal input, Active low	GP_OUT_R.CARD_DETECT_EN
card_mech_write_prot	Mechanical write protect signal input, Active low	GP_OUT_R.CARD_MECH_WRITE_PROT_EN
io_volt_sel	Signaling voltage select output (see 27.7.1 Switching Signaling Voltage from 3.3 V to 1.8 V)	GP_OUT_R.IO_VOLT_SEL_OE
card_if_pwr_en	Card interface power enable output	GP_OUT_R.CARD_IF_PWR_EN_OE

27.7.1 Switching Signaling Voltage from 3.3 V to 1.8 V

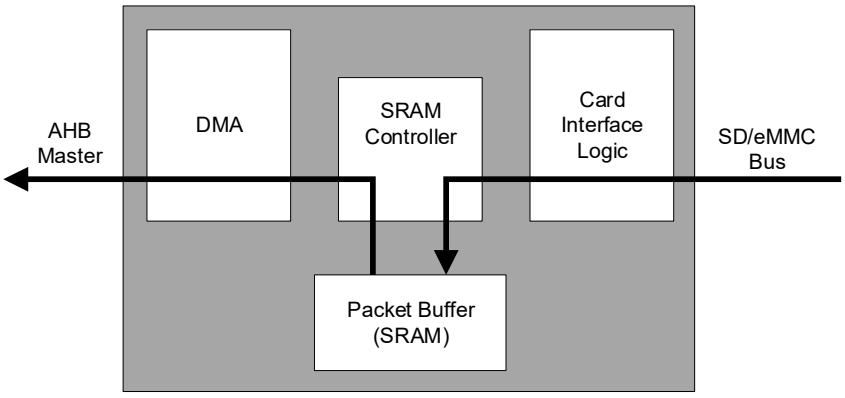
The I/Os operate at the voltage level supplied through the external V_{DDIO} pin. The SD mode supports switching the signaling voltage from 3.3 V to 1.8 V after negotiation with the SD card. The block sets the HOST_CTRL2_R.SIGNALING_EN bit to indicate the switch. This value is reflected on the io_volt_sel pin, which can be connected to an external regulator powering V_{DDIO} to switch between 3.3 V and 1.8 V. Note that PSoC 6 does not provide an internal regulator to power the SD interface I/Os.

27.8 Packet Buffer SRAM

SRAM that is internal to the SDHC block is used as a packet buffer to store data packets while carrying out data transfer to and from the card. The size of the SRAM is 1KB to support buffering of two 512 bytes blocks. As write and read transfers to the cards do not occur simultaneously, a single shared buffer is used for read and write operations. During the data transfer command handshake, the read/write bit of the command register is sampled and stored. This internal bit defines whether the SDHC is in read or write mode.

Figure 27-3 shows how data flows from the card interface to the AHB master interface through the packet buffer for a card read transfer. Received data from the card interface is written into packet buffer. When one block of data is received, DMA starts transmitting that data to the system by reading it from the packet buffer. For a card write transfer, data flows in the reverse direction. DMA writes data into a packet buffer that is subsequently read by the card interface logic. DMA and card interface logic can work simultaneously because read and write to packet buffer can be interleaved. For card read, DMA can send out the previous block while card interface logic is receiving the current block. For card write, DMA can write the current block into packet buffer while card interface logic is sending out the previous block.

Figure 27-3. Data Flow in a Read Transfer



27.8.1 Packet Buffer Full/Empty

When the packet buffer becomes full in card read, the clock to the card is stopped to prevent the card from sending the next data block. When packet buffer is empty, data block is not sent. In both cases, card interface logic is idle. SDHC does not support SDIO Read Wait signaling through DAT[2]. Therefore, the I/O command (CMD52) cannot be performed during a multiple read cycle because the card clock is stopped.

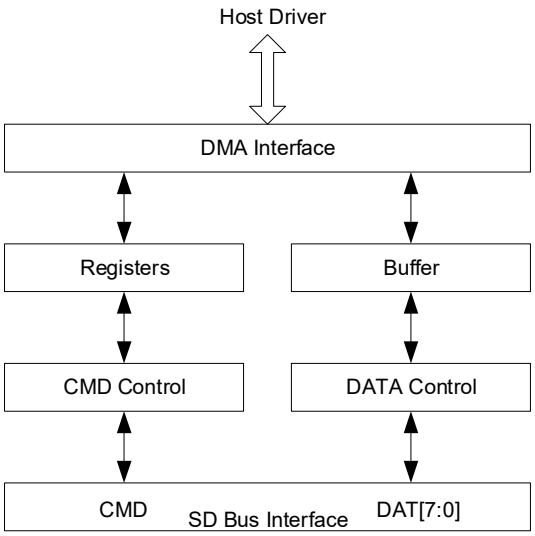
27.9 DMA Engine

The DMA engine handles data transfer between SDHC and system memory. Following are the features of this unit:

- Supports SDMA, ADMA2, and ADMA3 modes based on the configuration.
- The same DMA engine is used to interleave data transfer and descriptor fetch. This enables new task descriptor fetches (for CMD44 and CMD45) while DMA is moving data during task execution (for CMD46 and CMD47).
- Prefetches data for back-to-back eMMC write commands.
- Writes back the received data packets to system memory.

Figure 27-4 shows the data flow between the host driver and SD bus. The host driver can transfer data using either a programmed I/O (PIO) method in which the internal buffer is accessed through the buffer data port (BUF_DATA_R) register or using any of the defined DMA methods. PIO mode is much slower and burdens the processor. Do not use the PIO mode for large transfers.

Figure 27-4. Data Flow



DMA supports both single block and multi-block transfers. The control bits in the block gap control (BGAP_CTRL_R) register is used to stop and restart a DMA operation. SDMA mode is used for short data transfer because it generates interrupts at page boundaries. These interrupts disturb the CPU to reprogram the new system address. Only one SD command transaction can be executed for every SDMA operation.

ADMA2 and ADMA3 are used for long data transfers. They adopt scatter gather algorithm so that higher data transfer speed is available. The host driver can program a list of data transfers between system memory and SD card to the descriptor table. ADMA2 performs one read/write SD command operation at a time. ADMA3 can program multiple read/write SD command operation in a descriptor table.

In SDMA and ADMA2 modes, writing the CMD_R register triggers the DMA operation. In ADMA3 mode, writing ADMA_ID_LOW_R register triggers the DMA operation.

SD mode commands are generated by writing into the following registers – system address (SDMASA_R), block size (BLOCKSIZE_R), block count (BLOCKCOUNT_R), transfer mode (XFER_MODE_R), and command (CMD_R). When HOST_CTRL2_R.HOST_VER4_EN = 0, SDMA uses SDMASA_R as system address register and hence Auto CMD23 cannot be used with SDMA because this register is assigned for Auto CMD23 as the 32-bit block count register. When HOST_CTRL2_R.HOST_VER4_EN = 1, SDMA uses ADMA_SA_LOW_R as system address register and SDMASA_R is reassigned to 32-bit block count and hence SDMA may use Auto CMD23.

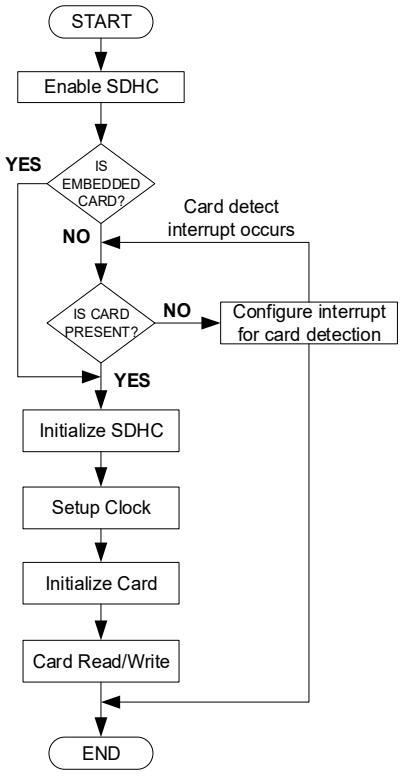
To use the 32-bit block count register when HOST_CTRL2_R.HOST_VER4_EN = 1, it must be programmed with a non-zero value and the value of the 16-bit block count register BLOCK_COUNT_R must be zero.

Refer to the respective specifications documents listed in Block Diagram on page 308 to learn more about the DMA operation.

27.10 Initialization Sequence

Figure 27-5 shows the sequence for initializing SDHC to work with SD/SDIO/eMMC cards. Subsequent sections describe each step. After initialization, SDHC is ready to communicate with the card. Refer to the corresponding specifications document for information on other sequences such as card initialization and identification, changing bus speed mode, signal voltage switch procedure, transaction generation, and error recovery.

Figure 27-5. SDHC Programming Sequence



27.10.1 Enabling SDHC

Ensure clk_sys is configured to be greater than or equal to clk_card and is running. Then, follow the sequence in Figure 27-6 to enable the block. The internal clock can also be enabled later during clock setup. It must be enabled to detect card insertion or removal through general interrupts when SDHC is not in standby mode. See 27.10.2 Card Detection for details.

Figure 27-6. SDHC Enable Sequence

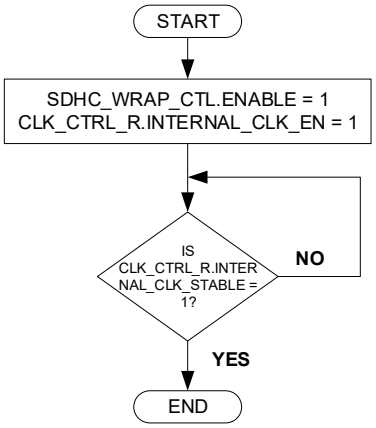
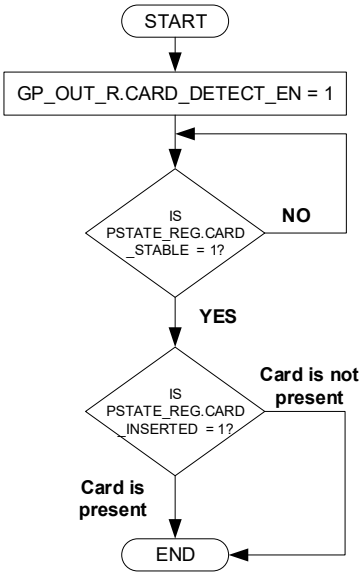


Figure 27-7. Card Status Check Sequence

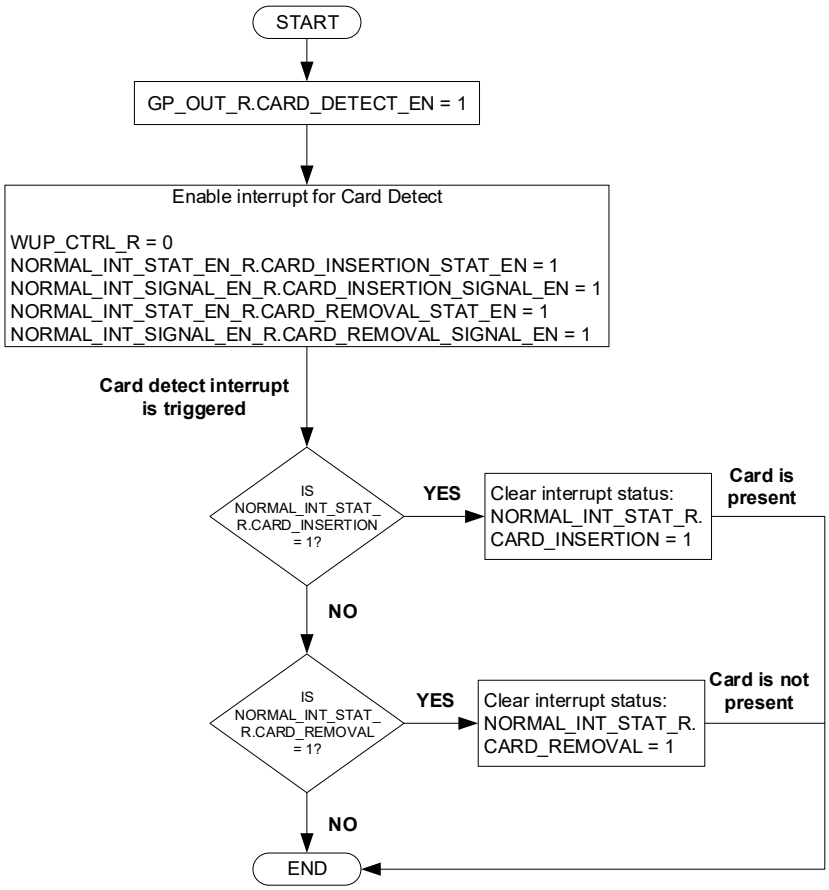


27.10.2 Card Detection

Check if the card is already inserted by following the sequence shown in Figure 27-7. This step is required for a removable card. After the card is detected, the host driver can supply power and clock to the card. If the card is inserted, then proceed with SDHC initialization. To detect card insertion or removal through interrupt when the internal clock is already enabled, follow the sequence shown in Figure 27-8. To detect the card status through interrupt when the internal clock is disabled (when SDHC is in standby mode), the bits in the WUP_CTRL_R register must be set and the NORMAL_INT_SIGNAL_EN register must be cleared. See Interrupts to CPU on page 310 for details. To detect SDIO card interrupt on DAT[1] line, a separate bit is provided in these registers, which must be configured.

SDHC clears the PWR_CTRL_R.SD_BUS_PWR_VDD1 bit when the card is removed and drives the DAT lines low. Therefore, the drive mode of the DAT lines must be changed from strong (with input buffer ON) to HI-Z when the card is removed to keep the lines pulled high. After detecting card insertion, the drive mode must be configured back to strong (with input buffer ON) mode only after PWR_CTRL_R.SD_BUS_PWR_VDD1 is set to 1.

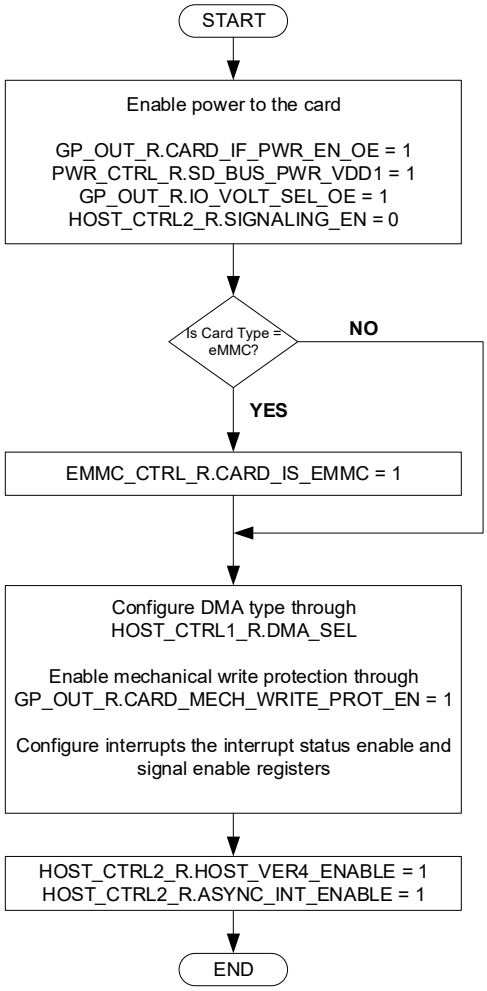
Figure 27-8. Card Detection Through Interrupt



27.10.3 SDHC Initialization

To initialize SDHC, configure the basic settings as shown in Figure 27-9. This step can also be executed immediately after enabling SDHC.

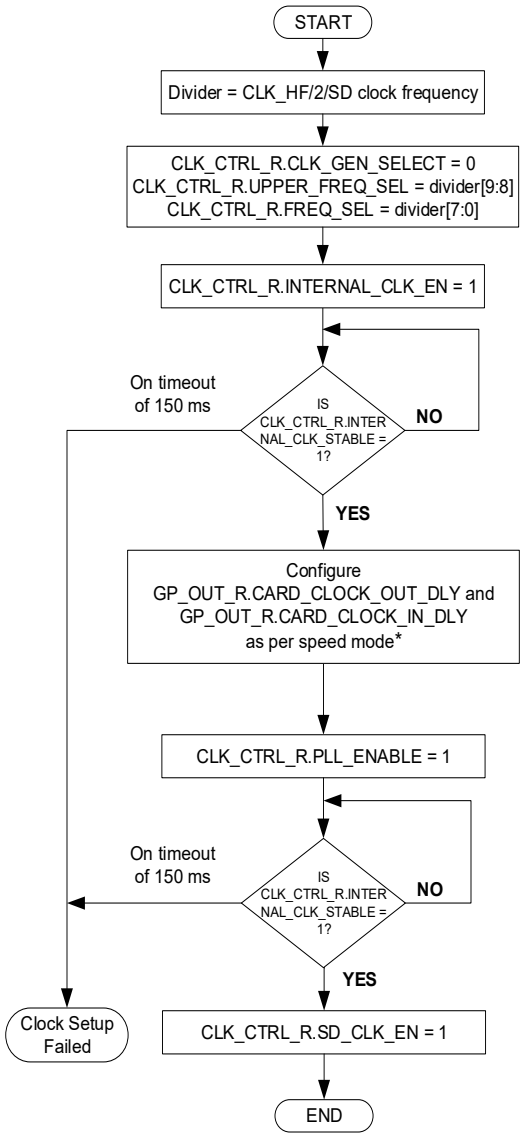
Figure 27-9. SDHC Setup



27.10.4 Clock Setup

Enable the internal clock followed by the card clock (SD clock) by following the sequence shown in Figure 27-10. The SD clock frequency must be 100 kHz to 400 kHz during the card initialization. See Card Clock (SDCLK) Configuration on page 309 for details. SD clock can be started and stopped by toggling the CLK_CTRL_R.SD_CLK_EN bit. The same sequence excluding the step of enabling the internal clock can be used to change the SD clock frequency. The SD clock must be stopped before changing its frequency. Note that GP_OUT_R.CARD_CLOCK_OE should have been set to '1' for the card clock to appear on the pin.

Figure 27-10. Clock Setup



* See the CY8C62x8, CY8C62xA Registers TRM for details.

27.11 Error Detection

SDHC can detect different types of errors in SD and eMMC transactions. Error is detected in either the command or data portion of the transaction. When an error is detected, the ERR_INTERRUPT bit in the NORMAL_INT_STAT_R register is set. The exact error can then be identified through the ERROR_INT_STAT_R register. The Abort command is used to recover from an error detected during data transfer. In addition to these two registers, SDHC has two other error status registers – Auto CMD Error Status (AUTO_CMD_STAT_R) and ADMA Error Status (ADMA_ERR_STAT_R). The following table lists the errors detected by SDHC.

Table 27-5. Errors Detected by SHDC

Type	Error
Command Errors	Command Timeout Error Command CRC Error Command End Bit Error Command Index Error Command Conflict Error Response Error
Auto Command Errors	Command not issued by Auto CMD12 Error Auto Command Timeout Error Auto Command CRC Error Auto Command End Bit Error Auto Command Index Error Auto Command Conflict Error Auto CMD response Error
Data Errors	Data Timeout Error Data CRC Error Data End Bit Error ADMA Error Tuning Error

28. Serial Communications Block (SCB)

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The Serial Communications Block (SCB) supports three serial communication protocols: Serial Peripheral Interface (SPI), Universal Asynchronous Receiver Transmitter (UART), and Inter Integrated Circuit (I^2C or IIC). Only one of the protocols is supported by an SCB at any given time. The number of SCBs in a PSoC 6 MCU varies by part number; consult the PSoC 61 datasheet/PSoC 62 datasheet to determine number of SCBs and the SCB pin locations. Not all SCBs support all three modes (SPI, UART, and I2C); consult the PSoC 61 datasheet/PSoC 62 datasheet to determine which modes are supported by which SCBs. Not all SCBs operate in deep sleep, consult the PSoC 61 datasheet/PSoC 62 datasheet to determine which SCBs operate in deep sleep.

28.1 Features

The SCB supports the following features:

- Standard SPI master and slave functionality with Motorola, Texas Instruments, and National Semiconductor protocols
- Standard UART functionality with SmartCard reader, Local Interconnect Network (LIN), and IrDA protocols
  - Standard LIN slave functionality with LIN v1.3 and LIN v2.1/2.2 specification compliance
- Standard I^2C master and slave functionality
- Trigger outputs for connection to DMA
- Multiple interrupt sources to indicate status of FIFOs and transfers
- Features available only on Deep Sleep-capable SCB:
  - EZ mode for SPI and I^2C slaves; allows for operation without CPU intervention
  - CMD_RESP mode for SPI and I^2C slaves; allows for operation without CPU intervention
  - Low-power (Deep Sleep) mode of operation for SPI and I^2C slaves (using external clocking)
  - Deep Sleep wakeup on I^2C slave address match or SPI slave selection

28.2 Architecture

The operation modes supported by SCB are described in the following sections.

28.2.1 Buffer Modes

Each SCB has 256 bytes of dedicated RAM for transmit and receive operation. This RAM can be configured in three different modes (FIFO, EZ, or CMD_RESP). The following sections give a high-level overview of each mode. The sections on each protocol will provide more details.

- Masters can only use FIFO mode
- I^2C and SPI slaves can use all three modes. **Note:** EZ Mode and CMD Response Mode are available only on the Deep Sleep-capable SCB
- UART only uses FIFO mode

Note: This document discusses hardware implementation of the EZ mode; for the firmware implementation, see the PDL.

28.2.1.1 FIFO Mode

In this mode the RAM is split into two 128-byte FIFOs, one for transmit (TX) and one for receive (RX). The FIFOs can be configured to be 8 bits x 128 elements or 16 bits x 64 elements; this is done by setting the BYTE_MODE bit in the SCB control register.

FIFO mode of operation is available only in Active and Sleep power modes. However, the I^2C address or SPI slave select can be used to wake the device from Deep Sleep on the Deep Sleep-capable SCB.

Statuses are provided for both the RX and TX FIFOs. There are multiple interrupt sources available, which indicate the status of the FIFOs, such as full or empty; see "SCB Interrupts" on page 366.

28.2.1.2 EZ Mode

In easy (EZ) mode the RAM is used as a single 256-byte buffer. The external master sets a base address and reads and writes start from that base address.

EZ Mode is available only for SPI slave and I^2C slave. It is available only on the Deep Sleep capable SCB.

EZ mode is available in Active, Sleep, and Deep Sleep power modes.

Note: This document discusses hardware implementation of the EZ mode; for the firmware implementation, see the PDL.

28.2.1.3 CMD_RESP Mode

Command Response (CMD_RESP) mode is similar to EZ mode except that the base address is provided by the CPU not the external master.

CMD_RESP mode is available only for SPI slave and I^2C slave. It is available only on the Deep Sleep-capable SCB.

CMD_RESP mode operation is available in Active, Sleep, and Deep Sleep power modes.

28.2.2 Clocking Modes

The SCB can be clocked either by an internal clock provided by the peripheral clock dividers (referred to as clk_scb in this document), or it can be clocked by the external master.

- UART, SPI master, and I^2C master modes must use clk_scb.
- Only SPI slave and I^2C slave can use the clock from and external master, and only the Deep Sleep capable SCB supports this.

Internally- and externally-clocked slave functionality is determined by two register fields of the SCB CTRL register:

- EC_AM_MODE indicates whether SPI slave selection or I^2C address matching is internally ('0') or externally ('1') clocked.
- EC_OP_MODE indicates whether the rest of the protocol operation (besides SPI slave selection and I^2C address matching) is internally ('0') or externally ('1') clocked.

Notes:

- FIFO mode supports an internally- or externally-clocked address match (EC_AM_MODE is '0' or '1'); however, data transfer must be done with internal clocking. (EC_OP_MODE is '1').
- EZ and CMD_RESP modes are supported with externally clocked operation (EC_OP_MODE is '1').

Table 28-1 provides an overview of the clocking and buffer modes supported for each communication mode.

Table 28-1. Clock Mode Compatibility

	Internally clocked (IC)			Externally clocked (EC) (Deep Sleep SCB only)		
	FIFO	EZ	CMD_RESP	FIFO	EZ	CMD_RESP
I^2C master	Yes	No	No	No	No	No
I^2C slave	Yes	Yes	No	Yes[a]	Yes	Yes
I^2C master-slave	Yes	No	No	No	No	No
SPI master	Yes	No	No	No	No	No
SPI slave	Yes	Yes	No	Yes[b]	Yes	Yes
UART transmitter	Yes	No	No	No	No	No
UART receiver	Yes	No	No	No	No	No

a. In Deep Sleep mode the external-clocked logic can handle slave address matching, it then triggers an interrupt to wake up the CPU. The slave can be programmed to stretch the clock, or NACK until internal logic takes over. This applies only to the Deep Sleep-capable SCB.
b. In Deep Sleep mode the external-clocked logic can handle slave selection detection, it then triggers an interrupt to wake up the CPU. Writes will be ignored and reads will return 0xFF until internal logic takes over. This applies only to the Deep Sleep-capable SCB.

Table 28-2. Clock Configuration and Mode support

Mode	EC_AM_MODE is '0'; EC_OP_MODE is '0'	'EC_AM_MODE is '1'; EC_OP_MODE is '0'	'EC_AM_MODE is '1'; EC_OP_MODE is '1'
FIFO mode	Yes	Yes	No
EZ mode	Yes	Yes	Yes
CMD_RESP mode	No	No	Yes

28.3 Serial Peripheral Interface (SPI)

The SPI protocol is a synchronous serial interface protocol. Devices operate in either master or slave mode. The master initiates the data transfer. The SCB supports single-master-multiple-slaves topology for SPI. Multiple slaves are supported with individual slave select lines.

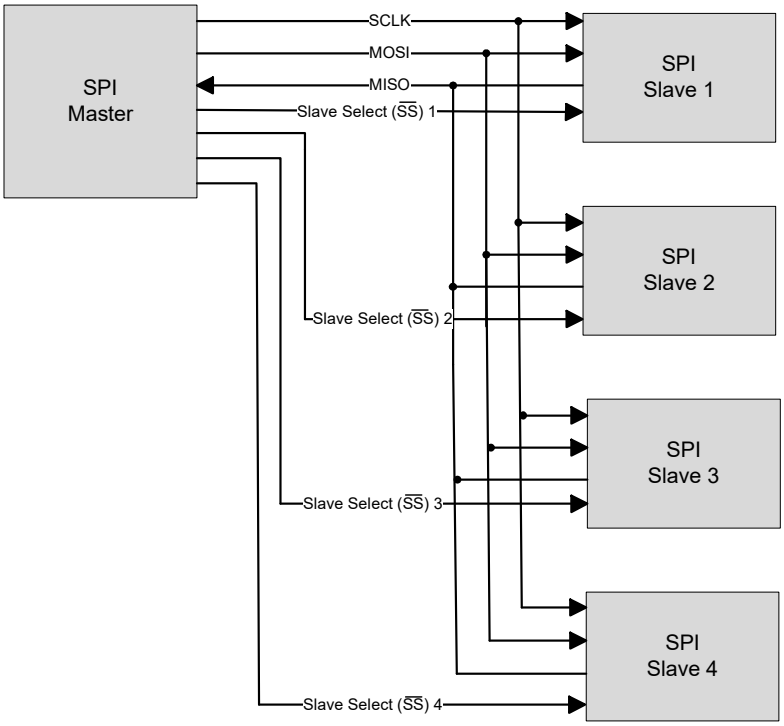
28.3.1 Features

- Supports master and slave functionality
- Supports three types of SPI protocols:
  - Motorola SPI – modes 0, 1, 2, and 3
  - Texas Instruments SPI, with coinciding and preceding data frame indicator – mode 1 only
  - National Semiconductor (MicroWire) SPI – mode 0 only
- Master supports up to four slave select lines
  - Each slave select has configurable active polarity (high or low)
  - Slave select can be programmed to stay active for a whole transfer, or just for each byte
- Master supports late sampling for better timing margin
- Master supports continuous SPI clock
- Data frame size programmable from 4 bits to 16 bits
- Programmable oversampling
- MSb or LSb first
- Median filter available for inputs (when using the median filter, the minimum oversample factor is increased)
- Supports FIFO Mode
- Supports EZ Mode (slave only) and CMD_RESP mode (slave only) on the Deep Sleep-capable SCB

28.3.2 General Description

Figure 28-1 illustrates an example of SPI master with four slaves.

Figure 28-1. SPI Example



A standard SPI interface consists of four signals as follows.

- SCLK: Serial clock (clock output from the master, input to the slave).
- MOSI: Master-out-slave-in (data output from the master, input to the slave).
- MISO: Master-in-slave-out (data input to the master, output from the slave).
- Slave Select ($\overline{SS}$): Typically an active low signal (output from the master, input to the slave).

A simple SPI data transfer involves the following: the master selects a slave by driving its $\overline{SS}$ line, then it drives data on the MOSI line and a clock on the SCLK line. The slave uses either of the edges of SCLK depending on the configuration to capture the data on the MOSI line; it also drives data on the MISO line, which is captured by the master.

28.3.2.1 Transfer Separation

This parameter determines if individual data transfers are separated by slave select deselection, which is controlled by SCB_SPI_CTRL.SSEL_CONTINUOUS (only applicable for Master mode):

- Continuous – The slave select line is held in active state until the end of transfer (default).

  The master assigns the slave select output after data has been written into the TX FIFO and keeps it active as long as there are data elements to transmit. The slave select output becomes inactive when all data elements have been transmitted from the TX FIFO and shifter register.

  Note: This can happen even in the middle of the transfer if the TX FIFO is not loaded fast enough by the CPU or DMA. To overcome this behavior, the slave select can be controlled by firmware.
- Separated – Every data frame 4-16 bits is separated by slave select line deselection by one SCLK period.

By default, the SPI interface supports a data frame size of eight bits (1 byte). The data frame size can be configured to any value in the range 4 to 16 bits. The serial data can be transmitted either most significant bit (MSb) first or least significant bit (LSb) first.

Three different variants of the SPI protocol are supported by the SCB:

- Motorola SPI: This is the original SPI protocol.
- Texas Instruments SPI: A variation of the original SPI protocol, in which data frames are identified by a pulse on the $\overline{SS}$ line.
- National Semiconductors SPI: A half-duplex variation of the original SPI protocol.

28.3.3 SPI Modes of Operation

28.3.3.1 Motorola SPI

The original SPI protocol was defined by Motorola. It is a full duplex protocol. Multiple data transfers may happen with the $\overline{SS}$ line held at '0'. When not transmitting data, the $\overline{SS}$ line is held at '1'.

Clock Modes of Motorola SPI

The Motorola SPI protocol has four different clock modes based on how data is driven and captured on the MOSI and MISO lines. These modes are determined by clock polarity (CPOL) and clock phase (CPHA).

Clock polarity determines the value of the SCLK line when not transmitting data. CPOL = '0' indicates that SCLK is '0' when not transmitting data. CPOL = '1' indicates that SCLK is '1' when not transmitting data.

Clock phase determines when data is driven and captured. CPHA = 0 means sample (capture data) on the leading (first) clock edge, while CPHA = 1 means sample on the trailing (second) clock edge, regardless of whether that clock edge is rising or falling. With CPHA = 0, the data must be stable for setup time before the first clock cycle.

- Mode 0: CPOL is '0', CPHA is '0': Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK.
- Mode 1; CPOL is '0', CPHA is '1': Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK.
- Mode 2: CPOL is '1', CPHA is '0': Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK.
- Mode 3: CPOL is '1', CPHA is '1': Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK.

Figure 28-2 illustrates driving and capturing of MOSI/MISO data as a function of CPOL and CPHA.

Figure 28-2. SPI Motorola, 4 Modes

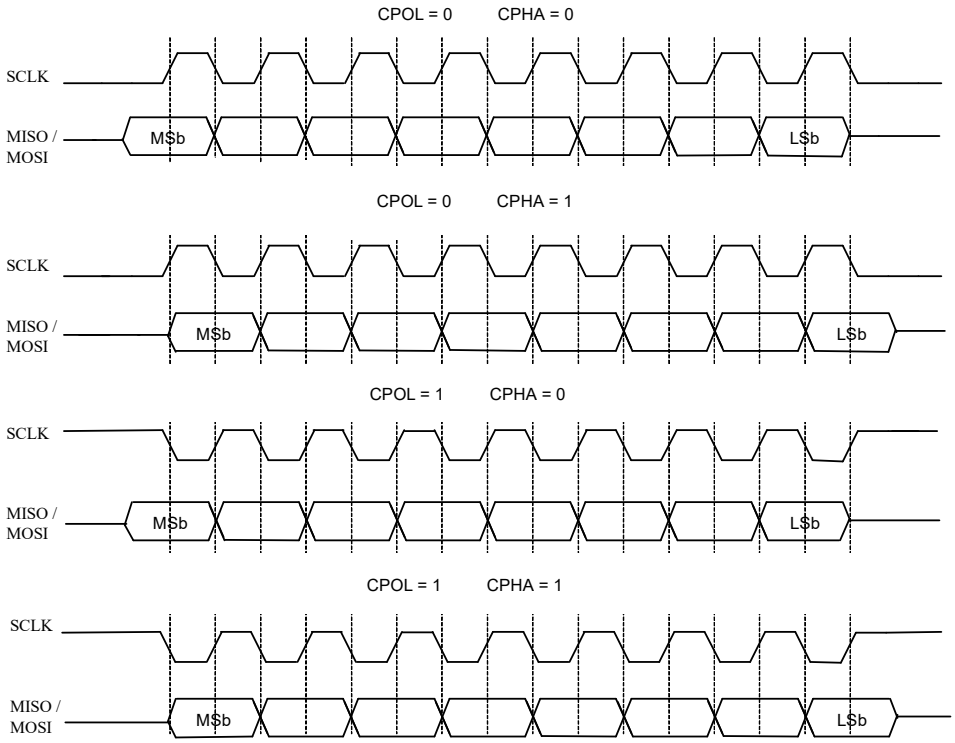


Figure 28-3 illustrates a single 8-bit data transfer and two successive 8-bit data transfers in mode 0 (CPOL is '0', CPHA is '0').

Figure 28-3. SPI Motorola Data Transfer Example

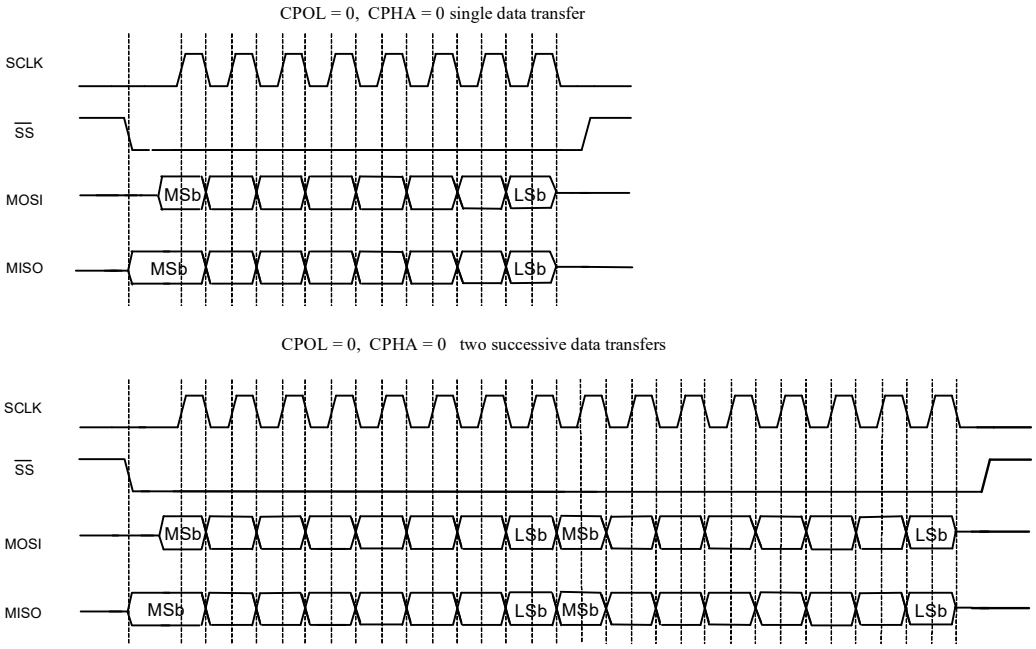
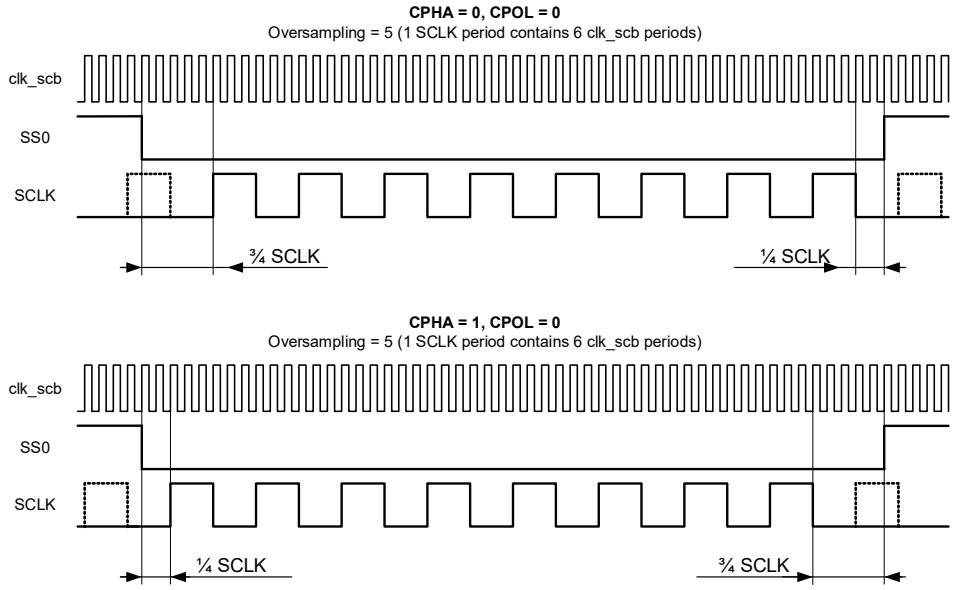


Figure 28-4. SELECT and SCLK Timing Correlation



For example above: OversamplingReg = 6 – 1 = 5.
¾ * SCLK = ((5 / 2) + 1) + (5 / 4 + 1)) * clk_scb = (3 + 2) * clk_scb = 5 * clk_scb.
¼ * SCLK = ((5 / 4) + 1) * clk_scb = 2 * clk_scb.

Note The value ¾ * SCLK is equal to (((OversamplingReg / 2) + 1) + (OversamplingReg / 4) + 1)), where OversamplingReg = Oversampling – 1.
The value ¼ * SCLK is equal to ((OversamplingReg / 4) + 1).
The result of any division operation is rounded down to the nearest integer.

Note The provided timings are guaranteed by SCB block but do not take into account signal propagation time from SCB block to pins.

Configuring SCB for SPI Motorola Mode

To configure the SCB for SPI Motorola mode, set various register bits in the following order:

1. Select SPI by writing ‘01’ to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI Motorola mode by writing ‘00’ to the MODE (bits [25:24]) of the SCB_SPI_CTRL register.
3. Select the clock mode in Motorola by writing to the CPHA and CPOL fields (bits 2 and 3 respectively) of the SCB_SPI_CTRL register.
4. Follow steps 2 to 4 mentioned in “Enabling and Initializing SPI” on page 334.

For more information on these registers, see the registers TRM.

28.3.3.2 Texas Instruments SPI

The Texas Instruments’ SPI protocol redefines the use of the $\overline{SS}$ signal. It uses the signal to indicate the start of a data transfer, rather than a low active slave select signal, as in the case of Motorola SPI. The start of a transfer is indicated by a high active pulse of a single bit transfer period. This pulse may occur one cycle before the transmission of the first data bit, or may coincide with the transmission of the first data bit. The TI SPI protocol supports only mode 1 (CPOL is ‘0’ and CPHA is ‘1’): data is driven on a rising edge of SCLK and data is captured on a falling edge of SCLK.

Figure 28-5 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SELECT pulse precedes the first data bit. Note how the SELECT pulse of the second data transfer coincides with the last data bit of the first data transfer.

Figure 28-5. SPI TI Data Transfer Example

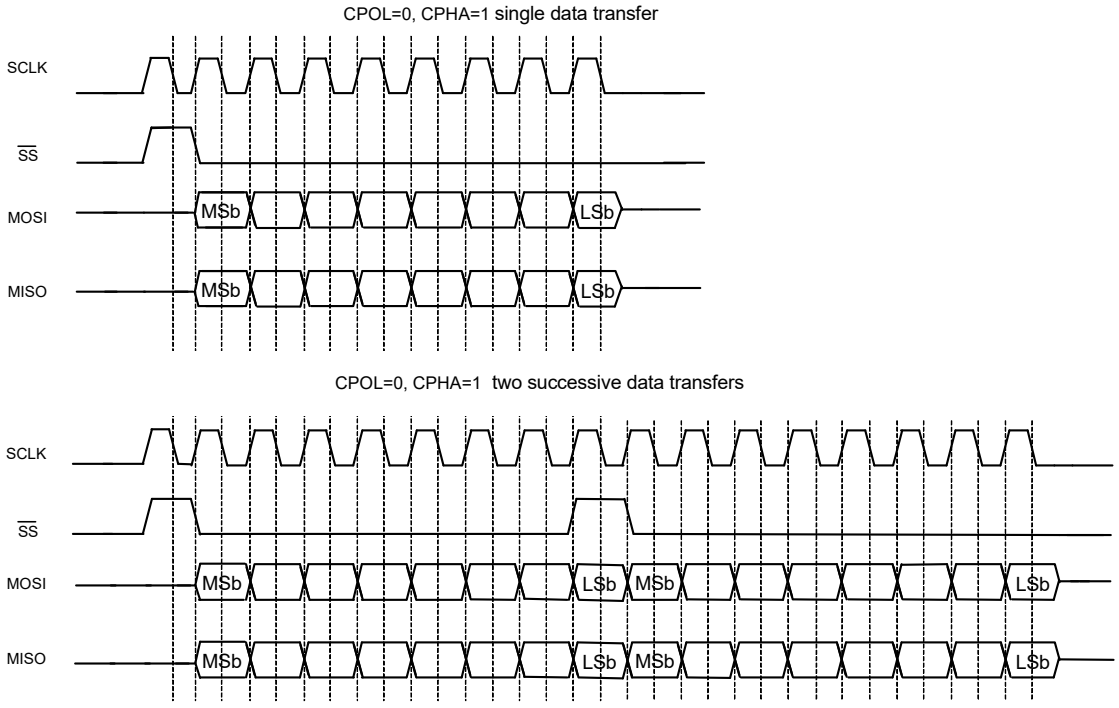
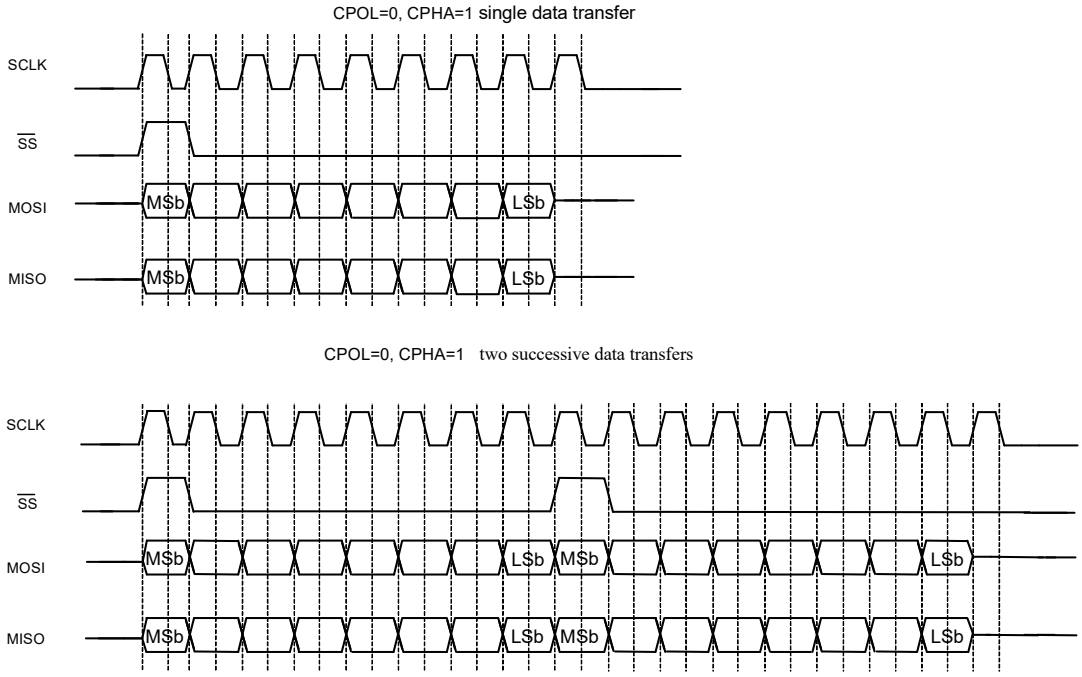


Figure 28-6 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SELECT pulse coincides with the first data bit of a frame.

Figure 28-6. SPI TI Data Transfer Example



Configuring SCB for SPI TI Mode

To configure the SCB for SPI TI mode, set various register bits in the following order:

1. Select SPI by writing ‘01’ to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI TI mode by writing ‘01’ to the MODE (bits [25:24]) of the SCB_SPI_CTRL register.
3. Select the mode of operation in TI by writing to the SELECT_PRECEDE field (bit 1) of the SCB_SPI_CTRL register (‘1’ configures the SELECT pulse to precede the first bit of next frame and ‘0’ otherwise).
4. Set the CPHA bit of the SCB_SPI_CONTROL register to ‘0’, and the CPOL bit of the same register to ‘1’.
5. Follow steps 2 to 4 mentioned in “Enabling and Initializing SPI” on page 334.

For more information on these registers, see the registers TRM.

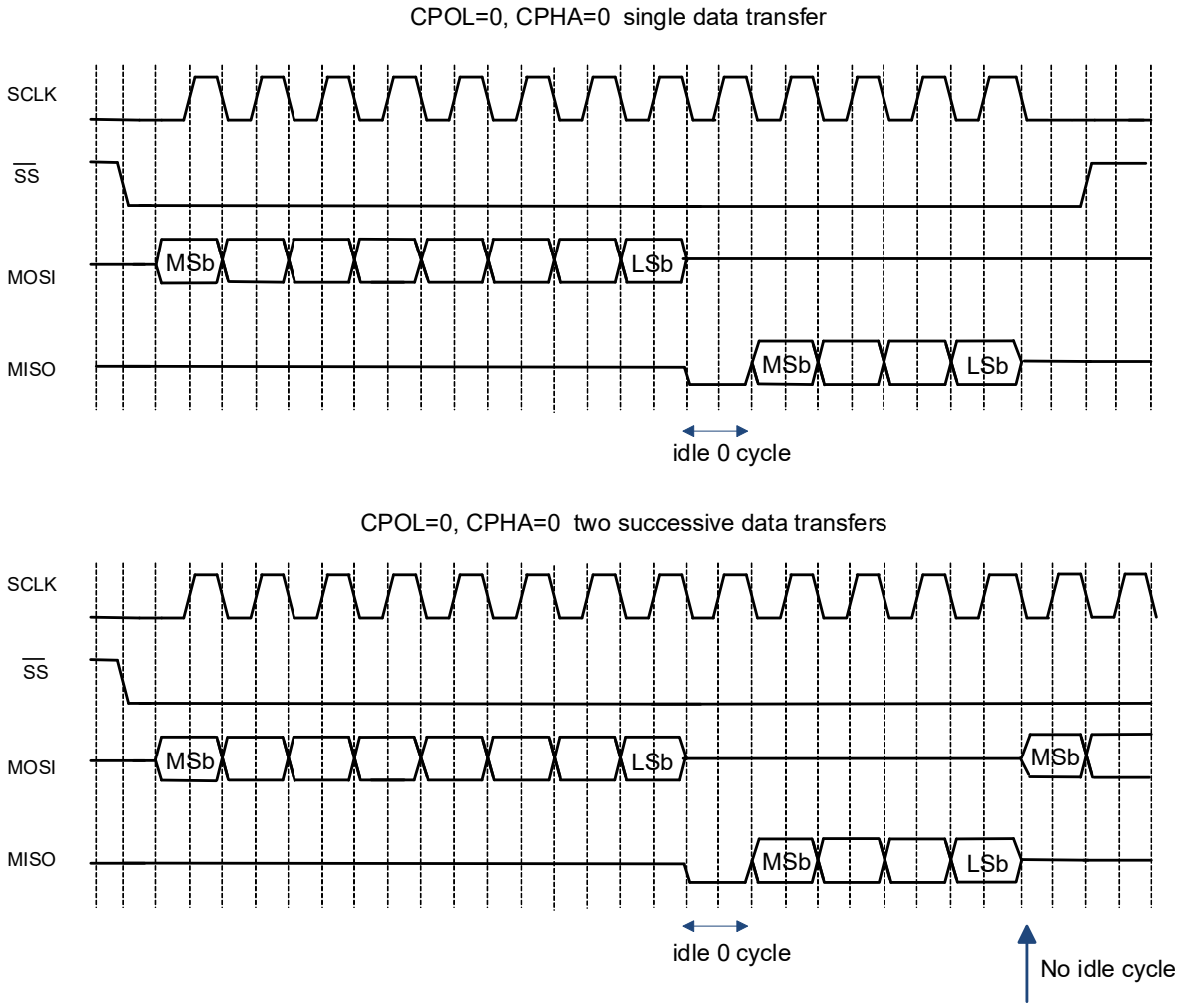
28.3.3.3 National Semiconductors SPI

The National Semiconductors’ SPI protocol is a half-duplex protocol. Rather than transmission and reception occurring at the same time, they take turns. The transmission and reception data sizes may differ. A single ‘idle’ bit transfer period separates transfers from reception. However, successive data transfers are not separated by an idle bit transfer period.

The National Semiconductors SPI protocol supports only mode 0.

Figure 28-7 illustrates a single data transfer and two successive data transfers. In both cases, the transmission data transfer size is eight bits and the reception data transfer size is four bits.

Figure 28-7. SPI NS Data Transfer Example



Configuring SCB for SPI NS Mode

To configure the SCB for SPI NS mode, set various register bits in the following order:

1. Select SPI by writing '01' to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI NS mode by writing '10' to the MODE (bits [25:24]) of the SCB_SPI_CTRL register.
3. Set the CPOL and CHPA bits of the SCB_SPI_CTRL register to '0'.
4. Follow steps 2 to 4 mentioned in "Enabling and Initializing SPI" on page 334.

For more information on these registers, see the registers TRM.

28.3.4 SPI Buffer Modes

SPI can operate in three different buffer modes – FIFO, EZ, and CMD_RESP modes. The buffer is used in different ways in each of these modes. The following subsections explain each of these buffer modes in detail.

28.3.4.1 FIFO Mode

The FIFO mode has a TX FIFO for the data being transmitted and an RX FIFO for the data received. Each FIFO is constructed out of the SRAM buffer. The FIFOs are either 64 elements deep with 16-bit data elements or 128 elements deep with 8-bit data elements. The width of a FIFO is configured using the BYTE_MODE bitfield of the SCB.CTRL register.

FIFO mode of operation is available only in Active and Sleep power modes, and not in the Deep Sleep mode.

Transmit and receive FIFOs allow write and read accesses. A write access to the transmit FIFO uses the TX_FIFO_WR register. A read access from the receive FIFO uses the RX_FIFO_RD register. For SPI master mode, data transfers are started when data is written into the TX FIFO. Note that when a master is transmitting and the FIFO becomes empty the slave is de-selected.

Transmit and receive FIFO status information is available through status registers, TX_FIFO_STATUS and RX_FIFO_STATUS, and through the INTR_TX and INTR_RX registers.

Each FIFO has a trigger output. This trigger output can be routed through the trigger mux to various other peripheral on the device such as DMA or TCPWMs. The trigger output of the SCB is controlled through the TRIGGER_LEVEL field in the RX_CTRL and TX_CTRL registers.

- For a TX FIFO a trigger is generated when the number of entries in the transmit FIFO is less than TX_FIFO_CTRL.TRIGGER_LEVEL.
- For the RX FIFO a trigger is generated when the number of entries in the FIFO is greater than the RX_FIFO_CTRL.TRIGGER_LEVEL.

Note that the DMA has a trigger deactivation setting. For the SCB this should be set to 16.

Active to Deep Sleep Transition

Before going to deep sleep ensure that all active communication is complete. For a master this can easily be done by checking the SPI_DONE bit in the INTR_M register, and ensuring the TX FIFO is empty.

For a slave this can be achieved by checking the BUS_BUSY bit in the SPI Status register. Also the RX FIFO should be empty before going to deep sleep. Any data in the FIFO will be lost during deep sleep.

Also before going to deep sleep the clock to the SCB needs to be disabled. This can be done by setting the SDA_IN_FILT_TRIM[1] bit in the I2C_CFG register to "0"

Lastly, when the device goes to deep sleep the SCB stops driving the GPIO lines. This leads to floating pins and can lead to undesirable current during deep sleep power modes. To avoid this condition before entering deep sleep mode change the HSIOM settings of the SCB pins to GPIO driven, then change the drive mode and drive state to the appropriate state to avoid floating pins. Consult the I/O System chapter on page 261 for more information on pin drive modes.

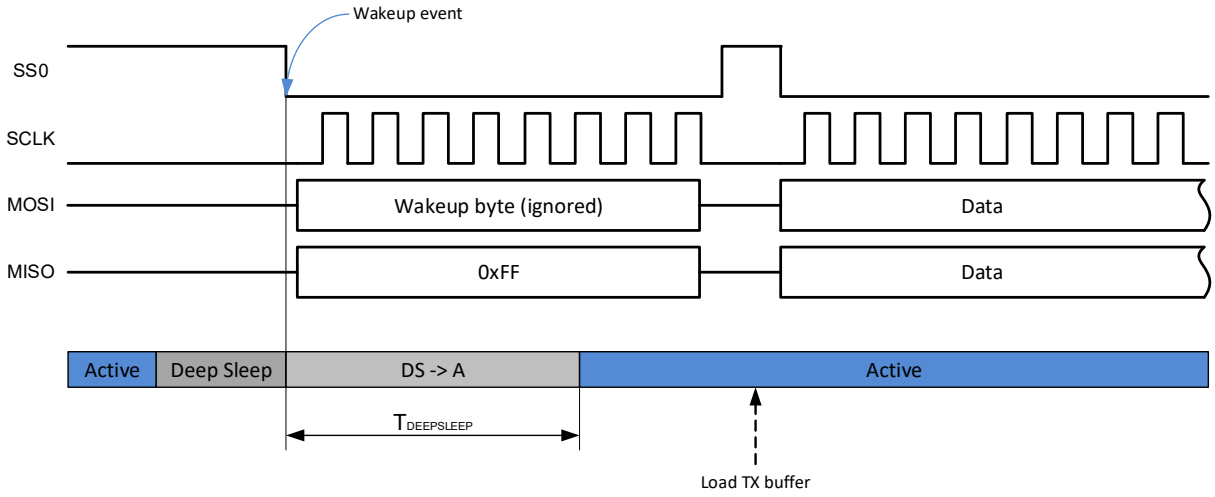
Note: Before going into deep sleep the wakeup interrupt should be cleared. See the “SPI Interrupts” on page 367 for more details.

Deep Sleep to Active Transition

EC_AM = 1, EC_OP = 0, FIFO Mode.

When the SPI Slave Select line is asserted the device will be awoken by an interrupt. After the device is awoken change the SPI pin drive modes and HSIOM settings back to what they were before deep sleep. When clk_hf[0] is at the desired frequency set SDA_IN_FILT_TRIM[1] to ‘1’ to enable the clock to the SCB. Then write data into the TX FIFO. At this point the master can read valid data from the slave. Before that any data read by the master will be invalid.

Figure 28-8. SPI Slave Wakeup from Deep Sleep (Motorola, CPHA = 0, CPOL = 0)



28.3.4.2 EZSPI Mode

The easy SPI (EZSPI) protocol only works in the Motorola mode, with any of the clock modes. It allows communication between master and slave without the need for CPU intervention. In the PSoC 6 MCU, only the deep sleep-capable SCB supports EZSPI mode.

The EZSPI protocol defines a single memory buffer with an 8-bit EZ address that indexes the buffer (256-entry array of eight bit per entry) located on the slave device. The EZ address is used to address these 256 locations. All EZSPI data transfers have 8-bit data frames.

The CPU writes and reads to the memory buffer through the SCB_EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

EZSPI has three types of transfers: a write of the EZ address from the master to the slave, a write of data from the master to an addressed slave memory location, and a read by the master from an addressed slave memory location.

Note: When multiple bytes are read or written the master must keep SSEL low during the entire transfer.

EZ Address Write

A write of the EZ address starts with a command byte (0x00) on the MOSI line indicating the master's intent to write the EZ address. The slave then drives a reply byte on the MISO line to indicate that the command is acknowledged (0xFE) or not (0xFF). The second byte on the MOSI line is the EZ address.

Memory Array Write

A write to a memory array index starts with a command byte (0x01) on the MOSI line indicating the master's intent to write to the memory array. The slave then drives a reply byte on the MISO line to indicate that the command was registered (0xFE) or not (0xFF). Any additional bytes on the MOSI line are written to the memory array at locations indicated by the communicated EZ address. The EZ address is automatically incremented by the slave as bytes are written into the memory array. When the EZ address exceeds the maximum number of memory entries (256), it remains there and does not wrap around to 0. The EZ base address is reset to the address written in the EZ Address Write phase on each slave selection.

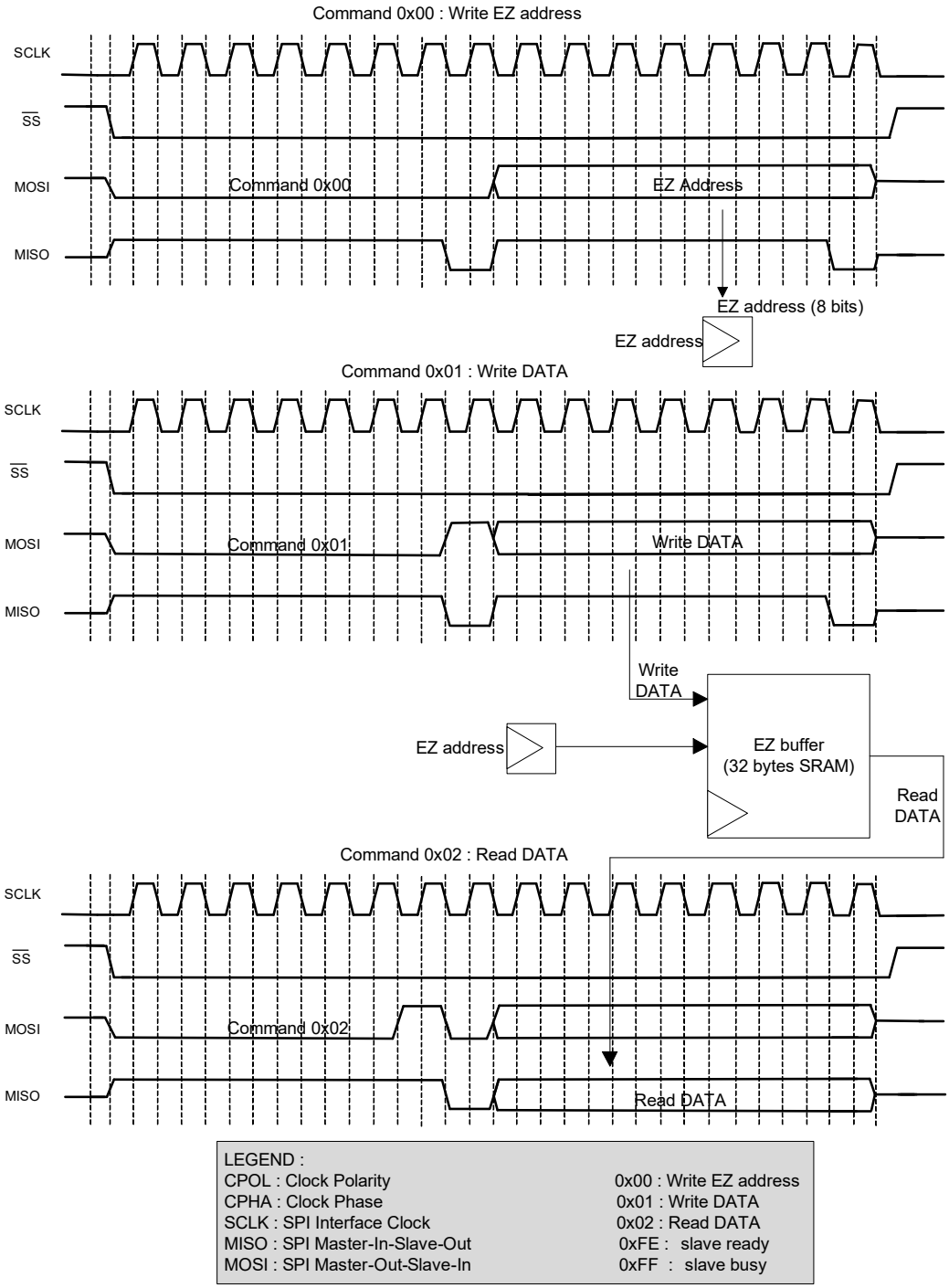
Memory Array Read

A read from a memory array index starts with a command byte (0x02) on the MOSI line indicating the master's intent to read from the memory array. The slave then drives a reply byte on the MISO line to indicate that the command was registered (0xFE) or not (0xFF). Any additional read data bytes on the MISO line are read from the memory array at locations indicated by the communicated EZ address. The EZ address is automatically incremented by the slave as bytes are read from the memory array. When the EZ address exceeds the maximum number of memory entries (256), it remains there and does not

wrap around to 0. The EZ base address is reset to the address written in the EZ Address Write phase on each slave selection.

Figure 28-9 illustrates the write of EZ address, write to a memory array and read from a memory array operations in the EZSPI protocol.

Figure 28-9. EZSPI Example



Configuring SCB for EZSPI Mode

By default, the SCB is configured for non-EZ mode of operation. To configure the SCB for EZSPI mode, set the register bits in the following order:

1. Select EZ mode by writing '1' to the EZ_MODE bit (bit 10) of the SCB_CTRL register.
2. Set the EC_AM and EC_OP modes in the SCB_CTRL register as appropriate.
3. Set the BYTE_MODE bit of the SCB_CTRL register to '1'.
4. Follow the steps in "Configuring SCB for SPI Motorola Mode" on page 324.
5. Follow steps 2 to 4 mentioned in "Enabling and Initializing SPI" on page 334.

For more information on these registers, see the registers TRM.

Active to Deep Sleep Transition

Before going to deep sleep ensure the master is not currently transmitting to the slave. This can be done by checking the BUS_BUSY bit in the SPI_STATUS register.

If the bus is not busy, disable the clock to the SCB by setting the SDA_IN_FILT_TRIM[1] bit to '0' in the I2C_CFG register.

Deep Sleep to Active Transition

- **EC_AM = 1, EC_OP = 0, EZ Mode.** MISO transmits 0xFF until the internal clock is enabled. Data on MOSI is ignored until the internal clock is enabled. Do not enable the internal clock until clk_hf[0] is at the desired frequency. After clk_hf[0] is at the desired frequency set the SDA_IN_FILT_TRIM[1] bit to '1' to enable the clock. The external master needs to be aware that when it reads 0xFF on MISO the device is not ready yet.
- **EC_AM = 1, EC_OP = 1, EZ Mode.** Do not enable the internal clock until clk_hf[0] is at the desired frequency. After clk_hf[0] is at the desired frequency set the SDA_IN_FILT_TRIM[1] bit to '1' to enable the clock.

28.3.4.3 Command-Response Mode

The command-response mode is defined only for an SPI slave. In the PSoC 6 MCU, only the deep sleep-capable SCB supports this mode. This mode has a single memory buffer, a base read address, a current read address, a base write address, and a current write address that are used to index the memory buffer. The base addresses are provided by the CPU. The current addresses are used by the slave to index the memory buffer for sequential accesses of the memory buffer. The memory buffer holds 256 8-bit data elements. The base and current addresses are in the range [0, 255]. This mode is only supported by the Motorola mode of operation.

The CPU writes and reads to the memory buffer through the SCB_EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

The slave interface accesses the memory buffer using the current addresses. At the start of a write transfer (SPI slave selection), the base write address is copied to the current write address. A data element write is to the current write address location. After the write access, the current address is incremented by '1'. At the start of a read transfer, the base read address is copied to the current read address. A data element read is to the current read address location. After the read data element is transmitted, the current read address is incremented by '1'.

If the current addresses equal the last memory buffer address (address equals 255), the current addresses are not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in command-response mode.

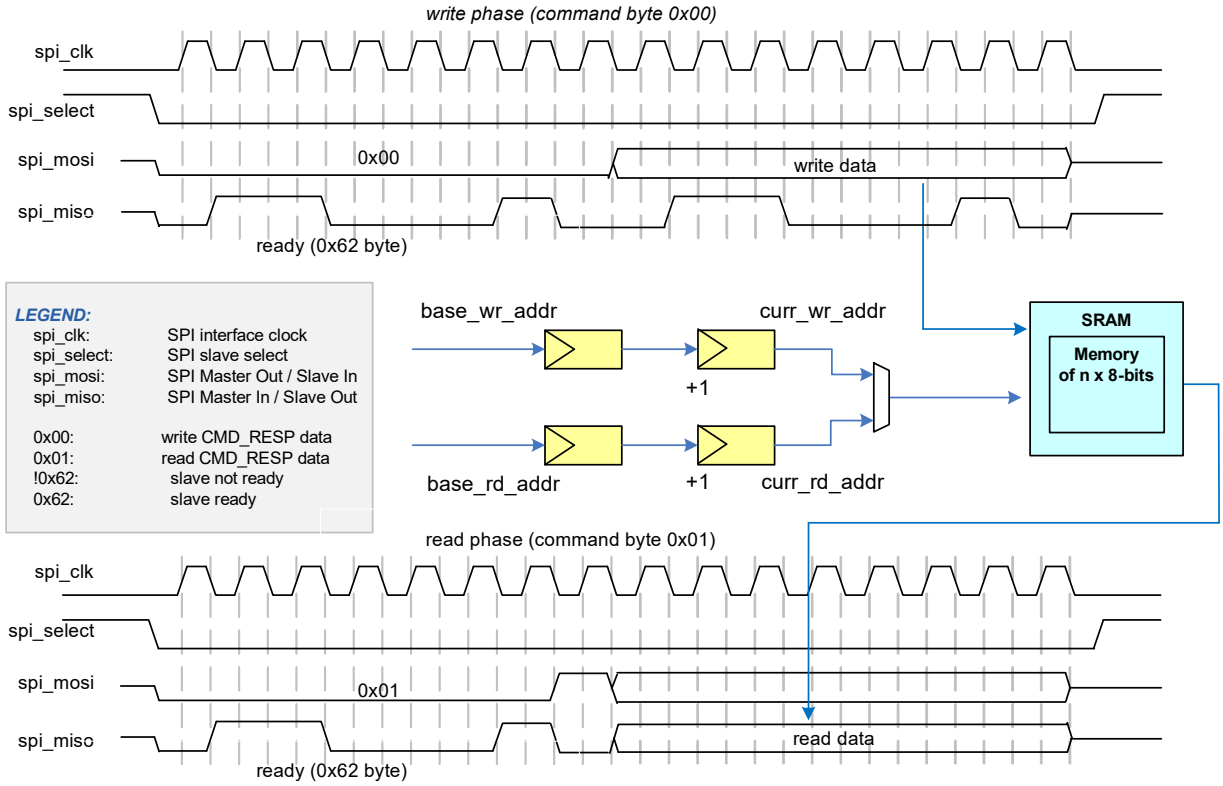
The base addresses are provided through CMD_RESP_CTRL. The current addresses are provided through CMD_RESP_STATUS. At the end of a transfer (SPI slave de-selection), the difference between a base and current address indicates how many read/write accesses were performed. The block provides interrupt cause fields to identify the end of a transfer. Command-response mode operation is available in Active, Sleep, and Deep Sleep power modes.

The command-response mode has two phases of operation:

- Write phase – The write phase begins with a selection byte, which has its last bit set to '0' indicating a write. The master writes 8-bit data elements to the slave's memory buffer following the selection byte. The slave's current write address is set to the slave's base write address. Received data elements are written to the current write address memory location. After each memory write, the current write address is incremented.
- Read phase – The read phase begins with a selection byte, which has its last bit set to '1' indicating a read. The master reads 8-bit data elements from the slave's memory buffer. The slave's current read address is set to the slave's base read address. Transmitted data elements are read from the current address memory location. After each read data element is transferred, the current read address is incremented.

During the reception of the first byte, the slave (MISO) transmits either 0x62 (ready) or a value different from 0x62 (busy). When disabled or reset, the slave transmits 0xFF (busy). The byte value can be used by the master to determine whether the slave is ready to accept the SPI request.

Figure 28-10. Command-Response Mode Example



Note that a slave's base addresses are updated by the CPU and not by the master.

Active to Deep Sleep Transition

Before going to deep sleep ensure the master is not currently transmitting to the slave. This can be done by checking the BUS_BUSY bit in the SPI_STATUS register.

If the bus is not busy, disable the clock to the SCB by setting the SDA_IN_FILT_TRIM[1] bit to 0 in the I2C_CFG register.

Deep Sleep to Active Transition

EC_AM = 1, EC_OP = 1, CMD_RESP Mode.

Do not enable the internal clock until clk_hf[0] is at the desired frequency. When clk_hf[0] is at the desired frequency, set the SDA_IN_FILT_TRIM[1] bit to '1' to enable the clock.

Configuring SCB for CMD_RESP Mode

By default, the SCB is configured for non-CMD_RESP mode of operation. To configure the SCB for CMD_RESP mode, set the register bits in the following order:

1. Select the CMD_RESP mode by writing '1' to the CMD_RESP_MODE bit (bit 12) of the SCB_CTRL register.
2. Set the EC_AM and EC_OP modes to '1' in the SCB_CTRL register.
3. Set the BYTE_MODE bit in the SCB_CTRL register.
4. Follow the steps in "Configuring SCB for SPI Motorola Mode" on page 324.
5. Follow steps 2 to 4 mentioned in"Enabling and Initializing SPI" on page 334.

For more information on these registers, see the registers TRM.

28.3.5 Clocking and Oversampling

28.3.5.1 Clock Modes

The SCB SPI supports both internally and externally clocked operation modes. Two bitfields (EC_AM_MODE and EC_OP MODE) in the SCB_CTRL register determine the SCB clock mode. EC_AM_MODE indicates whether SPI slave selection is internally (0) or externally (1) clocked. EC_OP_MODE indicates whether the rest of the protocol operation (besides SPI slave selection) is internally (0) or externally (1) clocked.

An externally-clocked operation uses a clock provided by the external master (SPI SCLK). **Note:** In the PSoC 6 MCU only the Deep Sleep-capable SCB supports externally-clocked mode of operation and only for SPI slave mode.

An internally-clocked operation uses the programmable clock dividers. For SPI, an integer clock divider must be used for both master and slave. For more information on system clocking, see the Clocking System chapter on page 242.

The SCB_CTRL bitfields EC_AM_MODE and EC_OP_MODE can be configured in the following ways.

- EC_AM_MODE is ‘0’ and EC_OP_MODE is ‘0’: Use this configuration when only Active mode functionality is required.
  - FIFO mode: Supported.
  - EZ mode: Supported.
  - Command-response mode: Not supported. The slave (MISO) transmits a value different from a ready (0x62) byte during the reception of the first byte if command-response mode is attempted in this configuration.
- EC_AM_MODE is ‘1’ and EC_OP_MODE is ‘0’: Use this configuration when both Active and Deep Sleep functionality are required. This configuration relies on the externally-clocked functionality to detect the slave selection and relies on the internally-clocked functionality to access the memory buffer.

  The “hand over” from external to internal functionality relies on a busy/ready byte scheme. This scheme relies on the master to retry the current transfer when it receives a busy byte and requires the master to support busy/ready byte interpretation. When the slave is selected, INTR_SPI_EC.WAKE_UP is set to ‘1’. The associated Deep Sleep functionality interrupt brings the system into Active power mode.
  - FIFO mode: Supported. The slave (MISO) transmits 0xFF until the CPU is awoken and the TX FIFO is populated. Any data on the MOSI line will be dropped until clk_scb is enabled see “Deep Sleep to Active Transition” on page 331 for more details
  - EZ mode: Supported. In Deep Sleep power mode, the slave (MISO) transmits a busy (0xFF) byte during the reception of the command byte. In Active power mode, the slave (MISO) transmits a ready (0xFE) byte during the reception of the command byte.
  - CMD_RESP mode: Not supported. The slave transmits (MISO) a value different from a ready (0x62) byte during the reception of the first byte.
- EC_AM_MODE is ‘1’ and EC_OP_MODE is ‘1’. Use this mode when both Active and Deep Sleep functionality are required. When the slave is selected, INTR_SPI_EC.WAKE_UP is set to ‘1’. The associated Deep Sleep functionality interrupt brings the system into Active power mode. When the slave is deselected, INTR_SPI_EC.EZ_STOP and/or INTR_SPI_EC.EZ_WRITE_STOP are set to ‘1’.
  - FIFO mode: Not supported.
  - EZ mode: Supported.
  - CMD_RESP mode: Supported.

If EC_OP_MODE is ‘1’, the external interface logic accesses the memory buffer on the external interface clock (SPI SCLK). This allows for EZ and CMD_RESP mode functionality in Active and Deep Sleep power modes.

In Active system power mode, the memory buffer requires arbitration between external interface logic (on SPI SCLK) and the CPU interface logic (on system peripheral clock). This arbitration always gives the highest priority to the external interface logic (host accesses). The external interface logic takes two serial interface clock/bit periods for SPI. During this period, the internal logic is denied service to the memory buffer. The PSoC 6 MCU provides two programmable options to address this “denial of service”:

- If the BLOCK bitfield of SCB_CTRL is ‘1’: An internal logic access to the memory buffer is blocked until the memory buffer is granted and the external interface logic has completed access. This option provides normal SCB register functionality, but the blocking time introduces additional internal bus wait states.
- If the BLOCK bitfield of SCB_CTRL is ‘0’: An internal logic access to the memory buffer is not blocked, but fails when it conflicts with an external interface logic access. A read access returns the value 0xFFFF:FFFF and a write access is ignored. This option does not introduce additional internal bus wait states, but an access to the memory buffer may not take effect. In this case, the following failures are detected:
  - Read Failure: A read failure is easily detected because the returned value is 0xFFFF:FFFF. This value is unique as non-failing memory buffer read accesses return an unsigned byte value in the range 0x0000:0000-0x0000:00ff.
  - Write Failure: A write failure is detected by reading back the written memory buffer location, and confirming that the read value is the same as the written value.

For both options, a conflicting internal logic access to the memory buffer sets INTR_TX.BLOCKED field to ‘1’ (for write accesses) and INTR_RX.BLOCKED field to ‘1’ (for read accesses). These fields can be used as either status fields or as interrupt cause fields (when their associated mask fields are enabled).

If a series of read or write accesses is performed and CTRL.BLOCKED is ‘0’, a failure is detected by comparing the “logical-or” of all read values to 0xFFFF:FFFF and checking the INTR_TX.BLOCKED and INTR_RX.BLOCKED fields to determine whether a failure occurred for a series of write or read operations.

Table 28-3. SPI Modes Compatibility

	Internally clocked (IC)			Externally clocked (EC) (Deep Sleep SCB only)		
	FIFO	EZ	CMD_RESP	FIFO	EZ	CMD_RESP
SPI master	Yes	No	No	No	No	No
SPI slave	Yes	Yes	No	Yes[a]	Yes	Yes

a. In SPI slave FIFO mode, the external-clocked logic does selection detection, then triggers an interrupt to wake up the CPU. Writes will be ignored and reads will return 0xFF until the CPU is ready and the FIFO is populated.

28.3.5.2 Using SPI Master to Clock Slave

In a normal SPI Master mode transmission, the SCLK is generated only when the SCB is enabled and data is being transmitted. This can be changed to always generate a clock on the SCLK line while the SCB is enabled. This is used when the slave uses the SCLK for functional operations other than just the SPI functionality. To enable this, write ‘1’ to the SCLK_CONTINUOUS (bit 5) of the SCB_SPI_CTRL register.

28.3.5.3 Oversampling and Bit Rate

SPI Master Mode

The SPI master does not support externally clocked mode. In internally clocked mode, the logic operates under internal clock. The internal clock has higher frequency than the interface clock (SCLK), such that the master can oversample its input signals (MISO).

The OVS (bits [3:0]) of the SCB_CTRL register specify the oversampling. The oversampling rate is calculated as the value in OVS register + 1. In SPI master mode, the valid range for oversampling is 4 to 16, when MISO is used; if MISO is not used then the valid range is 2 to 16. The bit rate is calculated as follows.

$$Bit\ Rate = clk\_scb/OVS$$

Equation 28-1

Hence, with clk_scb at 100 MHz, the maximum bit rate is 25 Mbps with MISO, or 50 Mbps without MISO.

The numbers above indicate how fast the SCB hardware can run SCLK. It does not indicate that the master will be able to correctly receive data from a slave at those speeds. To determine that, the path delay of MISO must be calculated. It can be calculated using the following equation:

$$\frac{1}{2}t_{SCLK} \geq t_{SCLK\_PCB\_D} + t_{DSO} + t_{SCLK\_PCB\_D} + t_{DSI}$$

Equation 28-2

Where:

t_{SCLK} is the period of the SPI clock

$t_{SCLK\_PCB\_D}$ is the SCLK PCB delay from master to slave

t_{DSO} is the total internal slave delay, time from SCLK edge at slave pin to MISO edge at slave pin

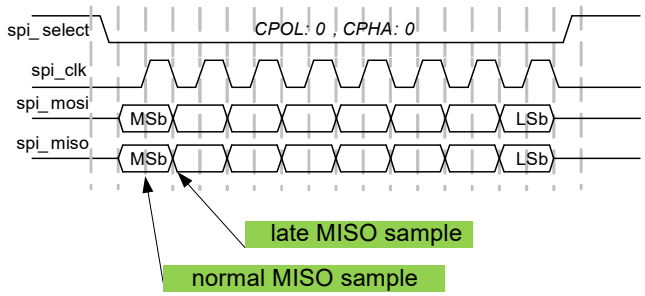
$t_{SCLK\_PCB\_D}$ is the MISO PCB delay from slave to master

t_{DSI} is the master setup time

Most slave datasheets will list tDSO, It may have a different name; look for MISO output valid after SCLK edge. Most master datasheets will also list tDSI, or master setup time. tSCLK_PCB_D and tSCLK_PCB_D must be calculated based on specific PCB geometries.

If after doing these calculations the desired speed cannot be achieved, then consider using the MISO late sample feature of the SCB. This can be done by setting the SPI_CTRL.LATE_MISO_SAMPLE register. Late sampling addresses the round-trip delay associated with transmitting SCLK from the master to the slave and transmitting MISO from the slave to the master. MISO late sample tells the SCB to sample the incoming MISO signal on the next edge of SCLK, thus allowing for ½ SCLK cycle more timing margin, see Figure 28-11.

Figure 28-11. MISO Sampling Timing



This changes the equation to:

$$t_{SCLK} \geq t_{SCLK\_PCB\_D} + t_{DSO} + t_{SCLK\_PCB\_D} + t_{DSI}$$

Equation 28-3

Because late sample allows for better timing, leave it enabled all the time. The t_{DSI} specification in the PSoC 6 MCU datasheet assumes late sample is enabled.

Note: The SPI_CTRL.LATE_MISO_SAMPLE is set to '0' by default.

SPI Slave Mode

In SPI slave mode, the OVS field (bits [3:0]) of SCB_CTRL register is not used. The data rate is determined by Equation 26-2 and Equation 26-3. Late MISO sample is determined by the external master in this case, not by SPI_CTRL.LATE_MISO_SAMPLE.

For PSoC 6 MCUs, t_{DSO} is given in the PSoC 61 datasheet/PSoC 62 datasheet. For internally-clocked mode, it is proportional to the frequency of the internal clock. For example it may be 20 ns + 3 *t_{CLK_SCB}. Assuming 0 ns PCB delays, and a 0 ns external master t_{DSI} Equation 26-1 can be re-arranged to $t_{CLK\_SCB} \leq ((t_{SCLK}) - 40\ ns)/6$.

28.3.6 Enabling and Initializing SPI

The SPI must be programmed in the following order:

1. Program protocol specific information using the SCB_SPI_CTRL register. This includes selecting the sub-modes of the protocol and selecting master-slave functionality. EZSPI and CMD_RESP can be used with slave mode only.
2. Program the OVS field and configure clk_scb as appropriate. See the Clocking System chapter on page 242 for more information on how to program clocks and connect it to the SCB.
3. Configure SPI GPIO by setting appropriate drive modes and HSIOM settings.
4. Select the desired Slave Select line and polarity in the SCB_SPI_CTRL register.
5. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers:
   a. Specify the data frame width. This should always be 8 for EZSPI and CMD_RESP.
   b. Specify whether MSb or LSb is the first bit to be transmitted/received. This should always be MSb first for EZSPI and CMD_RESP.
6. Program the transmitter and receiver FIFOs using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers respectively, as shown in SCB_TX_FIFO_CTRL/SCB_RX_FIFO_CTRL registers. Only for FIFO mode.
   a. Set the trigger level.
   b. Clear the transmitter and receiver FIFO and Shift registers.
7. Enable the block (write a '1' to the ENABLED bit of the SCB_CTRL register). After the block is enabled, control bits should not be changed. Changes should be made after disabling the block; for example, to modify the operation mode (from

Motorola mode to TI mode) or to go from externally clocked to internally clocked operation. The change takes effect only after the block is re-enabled. **Note:** Re-enabling the block causes re-initialization and the associated state is lost (for example, FIFO content).

28.3.7 I/O Pad Connection

28.3.7.1 SPI Master

Figure 28-12 and Table 28-4 list the use of the I/O pads for SPI Master.

Figure 28-12. SPI Master I/O Pad Connections

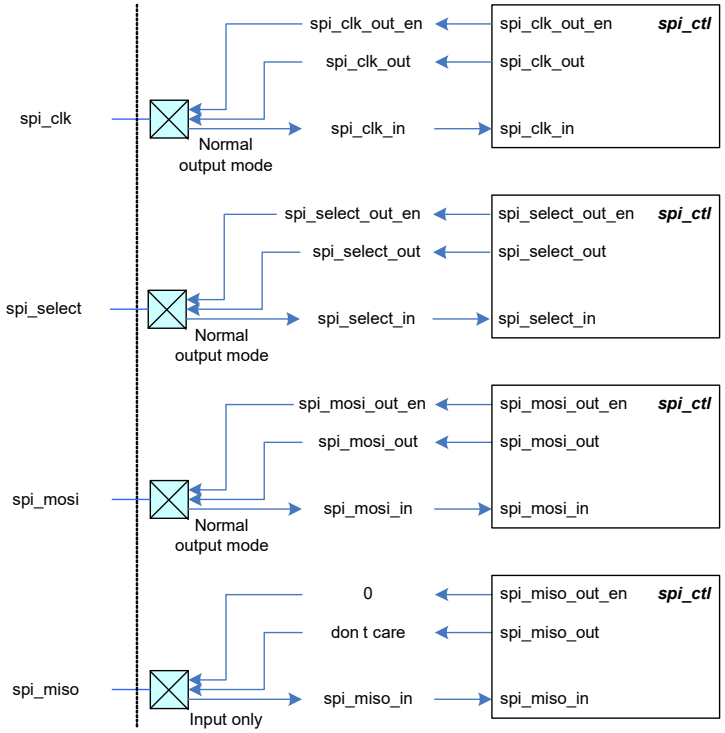


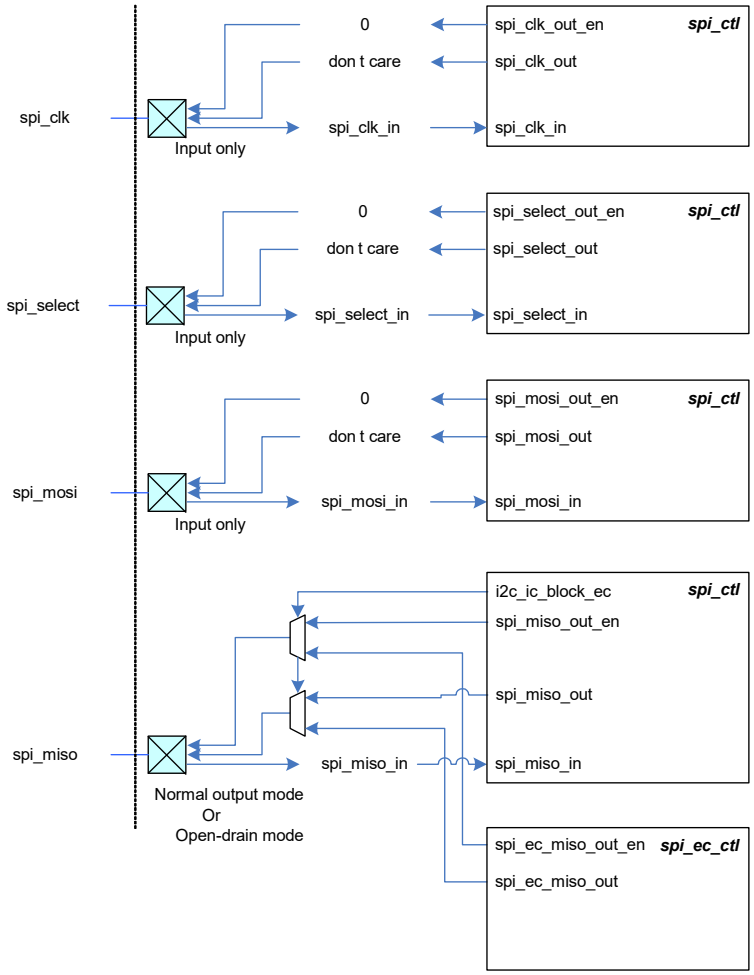
Table 28-4. SPI Master I/O Pad Connection Usage

I/O Pads	Drive Mode	On-chip I/O Signals	Usage
spi_clk	Normal output mode	spi_clk_out_en spi_clk_out	Transmit a clock signal
spi_select	Normal output mode	spi_select_out_en spi_select_out	Transmit a select signal
spi_mosi	Normal output mode	spi_mosi_out_en spi_mosi_out	Transmit a data element
spi_miso	Input only	spi_miso_in	Receive a data element

28.3.7.2 SPI Slave

Figure 28-13 and Table 28-5 list the use of I/O pads for SPI Slave.

Figure 28-13. SPI Slave I/O Pad Connections



Open_Drain is set in the TX_CTRL register. In this mode the SPI MISO pin is actively driven low, and then high-z for driving high. This means an external pull-up is required for the line to go high. This mode is useful when there are multiple slaves on the same line. This helps to avoid bus contention issues.

Table 28-5. SPI Slave I/O Signal Description

I/O Pads	Drive Mode	On-chip I/O Signals	Usage
spi_clk	Input mode	spi_clk_in	Receive a clock signal
spi_select	Input mode	spi_select_in	Receive a select signal
spi_mosi	Input mode	spi_mosi_in	Receive a data element
spi_miso	Normal output mode	spi_miso_out_en spi_miso_out	Transmit a data element

28.3.7.3 Glitch Avoidance at System Reset

The SPI outputs are in high-impedance digital state when the device is coming out of system reset. This can cause glitches on the outputs. This is important if you are concerned with SPI master SS0 – SS3 or SCLK output pins activity at either device startup or when coming out of Hibernate mode. External pull-up or pull-down resistor can be connected to the output pin to keep it in the inactive state.

28.3.7.4 Median Filter

This parameter applies a three-tap digital median filter on the input line. The master has one input line: MISO, and the slave has three input lines: SCLK, MOSI, and SS. This filter reduces the susceptibility to errors. However, minimum oversampling factor value is increased. The default value is 'Disabled'. To enable the median filter write a '1' to RX_CTRL.MEDIAN.

28.3.8 SPI Registers

The SPI interface is controlled using a set of 32-bit control and status registers listed in Table 28-6. Some of these registers are used specifically with the Deep Sleep SCB; for more information on these registers, see the registers TRM.

Table 28-6. SPI Registers

Register Name	Operation
SCB_CTRL	Enables the SCB, selects the type of serial interface (SPI, UART, I^2C), and selects internally and externally clocked operation, and EZ and non-EZ modes of operation.
SCB_STATUS (Deep Sleep SCB only)	In EZ mode, this register indicates whether the externally clocked logic is potentially using the EZ memory.
SCB_SPI_CTRL	Configures the SPI as either a master or a slave, selects SPI protocols (Motorola, TI, National) and clock-based submodes in Motorola SPI (modes 0,1,2,3), selects the type of $\overline{SS}$ signal in TI SPI.
SCB_SPI_STATUS	Indicates whether the SPI bus is busy and sets the SPI slave EZ address in the internally clocked mode.
SCB_TX_CTRL	Specifies the data frame width and specifies whether MSb or LSb is the first bit in transmission.
SCB_RX_CTRL	Performs the same function as that of the SCB_TX_CTRL register, but for the receiver. Also decides whether a median filter is to be used on the input interface lines.
SCB_TX_FIFO_CTRL	Specifies the trigger level, clears the transmitter FIFO and shift registers, and performs the FREEZE operation of the transmitter FIFO.
SCB_RX_FIFO_CTRL	Performs the same function as that of the SCB_TX_FIFO_CTRL register, but for the receiver.
SCB_TX_FIFO_WR	Holds the data frame written into the transmitter FIFO. Behavior is similar to that of a PUSH operation.
SCB_RX_FIFO_RD	Holds the data frame read from the receiver FIFO. Reading a data frame removes the data frame from the FIFO - behavior is similar to that of a POP operation. This register has a side effect when read by software: a data frame is removed from the FIFO.
SCB_RX_FIFO_RD_SILENT	Holds the data frame read from the receiver FIFO. Reading a data frame does not remove the data frame from the FIFO; behavior is similar to that of a PEEK operation.
SCB_TX_FIFO_STATUS	Indicates the number of bytes stored in the transmitter FIFO, the location from which a data frame is read by the hardware (read pointer), the location from which a new data frame is written (write pointer), and decides whether the transmitter FIFO holds the valid data.
SCB_RX_FIFO_STATUS	Performs the same function as that of the SCB_TX_FIFO_STATUS register, but for the receiver.
SCB_EZ_DATA (Deep Sleep SCB only)	Holds the data in EZ memory location

28.4 UART

The Universal Asynchronous Receiver/Transmitter (UART) protocol is an asynchronous serial interface protocol. UART communication is typically point-to-point. The UART interface consists of two signals:

- TX: Transmitter output
- RX: Receiver input

Additionally, two side-band signals are used to implement flow control in UART. Note that the flow control applies only to TX functionality.

- Clear to Send (CTS): This is an input signal to the transmitter. When active, the receiver signals to the transmitter that it is ready to receive.
- Ready to Send (RTS): This is an output signal from the receiver. When active, it indicates that the receiver is ready to receive data.

Not all SCBs support UART mode; refer to the PSoC 61 datasheet/PSoC 62 datasheet for details.

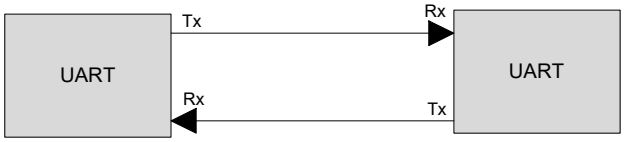
28.4.1 Features

- Supports UART protocol
  - Standard UART
  - Multi-processor mode
- SmartCard (ISO7816) reader
- IrDA
- Supports Local Interconnect Network (LIN)
  - Break detection
  - Baud rate detection
  - Collision detection (ability to detect that a driven bit value is not reflected on the bus, indicating that another component is driving the same bus)
- Data frame size programmable from 4 to 16 bits
- Programmable number of STOP bits, which can be set in terms of half bit periods between 1 and 4
- Parity support (odd and even parity)
- Median filter on RX input
- Programmable oversampling
- Start skipping
- Hardware flow control

28.4.2 General Description

Figure 28-14 illustrates a standard UART TX and RX.

Figure 28-14. UART Example



A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start and stop bits indicate the start and end of data transmission. The parity bit is sent by the transmitter and is used by the receiver to detect single bit errors. Because the interface does not have a clock (asynchronous), the transmitter and receiver use their own clocks; thus, the transmitter and receiver need to agree on the baud rate.

By default, UART supports a data frame width of eight bits. However, this can be configured to any value in the range of 4 to 9. This does not include start, stop, and parity bits. The number of stop bits can be in the range of 1 to 4. The parity bit can be either enabled or disabled. If enabled, the type of parity can be set to either even parity or odd parity. The option of using the parity bit is available only in the Standard UART and SmartCard UART modes. For IrDA UART mode, the parity bit is automatically disabled.

Note: UART interface does not support external clocking operation. Hence, UART operates only in the Active and Sleep system power modes. UART also supports only the FIFO buffer mode.

28.4.3 UART Modes of Operation

28.4.3.1 Standard Protocol

A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start bit value is always '0', the data bits values are dependent on the data transferred, the parity bit value is set to a value guaranteeing an even or odd parity over the data bits, and the stop bit value is '1'. The parity bit is generated by the transmitter and can be used by the receiver to detect single bit transmission errors. When not transmitting data, the TX line is '1' – the same value as the stop bits.

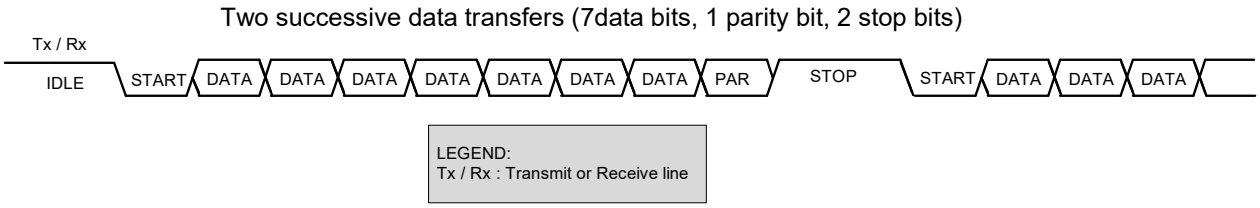
Because the interface does not have a clock, the transmitter and receiver must agree upon the baud rate. The transmitter and receiver have their own internal clocks. The receiver clock runs at a higher frequency than the bit transfer frequency, such that the receiver may oversample the incoming signal.

The transition of a stop bit to a start bit is represented by a change from '1' to '0' on the TX line. This transition can be used by the receiver to synchronize with the transmitter clock. Synchronization at the start of each data transfer allows error-free transmission even in the presence of frequency drift between transmitter and receiver clocks. The required clock accuracy is dependent on the data transfer size.

The stop period or the amount of stop bits between successive data transfers is typically agreed upon between transmitter and receiver, and is typically in the range of 1 to 3-bit transfer periods.

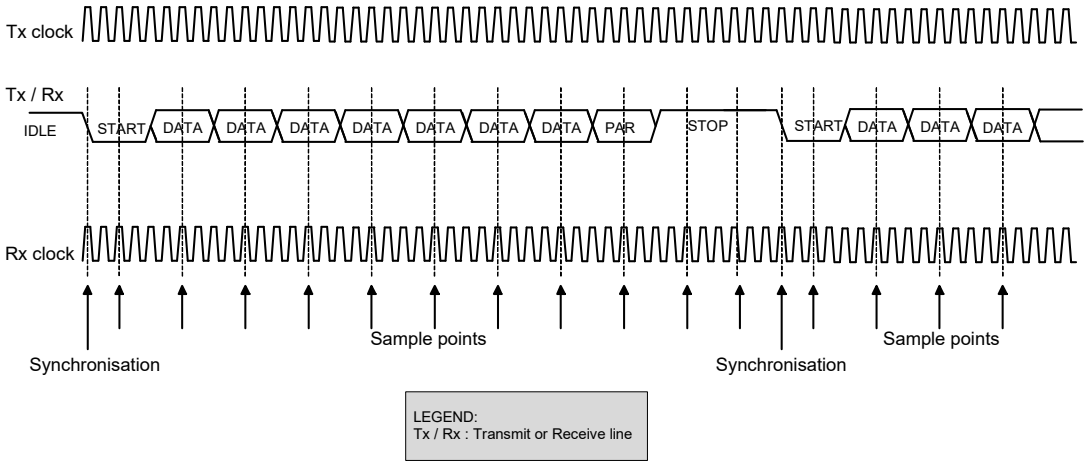
Figure 28-15 illustrates the UART protocol.

Figure 28-15. UART, Standard Protocol Example



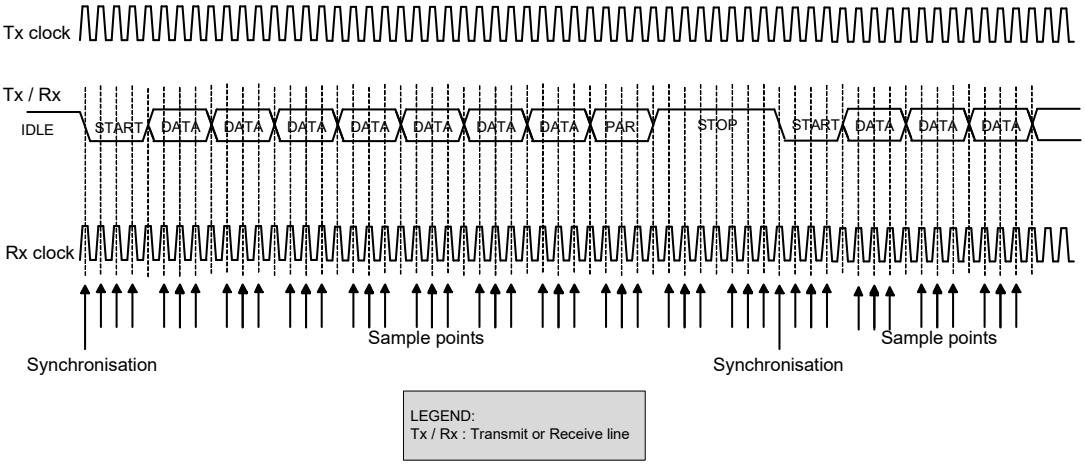
The receiver oversamples the incoming signal; the value of the sample point in the middle of the bit transfer period (on the receiver's clock) is used. Figure 28-16 illustrates this.

Figure 28-16. UART, Standard Protocol Example (Single Sample)



Alternatively, three samples around the middle of the bit transfer period (on the receiver's clock) are used for a majority vote to increase accuracy; this is enabled by enabling the MEDIAN filter in the SCB_RX_CTRL register. Figure 28-17 illustrates this.

Figure 28-17. UART, Standard Protocol (Multiple Samples)



Parity

This functionality adds a parity bit to the data frame and is used to identify single-bit data frame errors. The parity bit is always directly after the data frame bits.

The transmitter calculates the parity bit (when UART_TX_CTRL.PARITY_ENABLED is 1) from the data frame bits, such that data frame bits and parity bit have an even (UART_TX_CTRL.PARITY is 0) or odd (UART_TX_CTRL.PARITY is 1) parity. The receiver checks the parity bit (when UART_RX_CTRL.PARITY_ENABLED is 1) from the received data frame bits, such that data frame bits and parity bit have an even (UART_RX_CTRL.PARITY is 0) or odd (UART_RX_CTRL.PARITY is 1) parity.

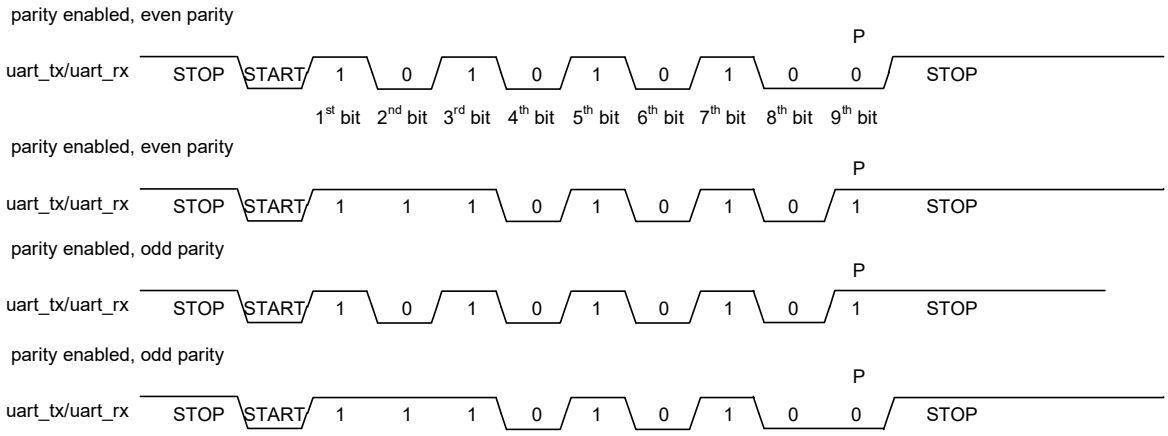
Parity applies to both TX and RX functionality and dedicated control fields are available.

- Transmit functionality: UART_TX_CTRL.PARITY and UART_TX_CTRL.PARITY_ENABLED.
- Receive functionality: UART_RX_CTRL.PARITY and UART_RX_CTRL.PARITY_ENABLED.

When a receiver detects a parity error, the data frame is either put in RX FIFO (UART_RX_CTRL.DROP_ON_PARITY_ERROR is 0) or dropped (UART_RX_CTRL.DROP_ON_PARITY_ERROR is 1).

The following figures illustrate the parity functionality (8-bit data frame).

Figure 28-18. UART Parity Examples



Start Skipping

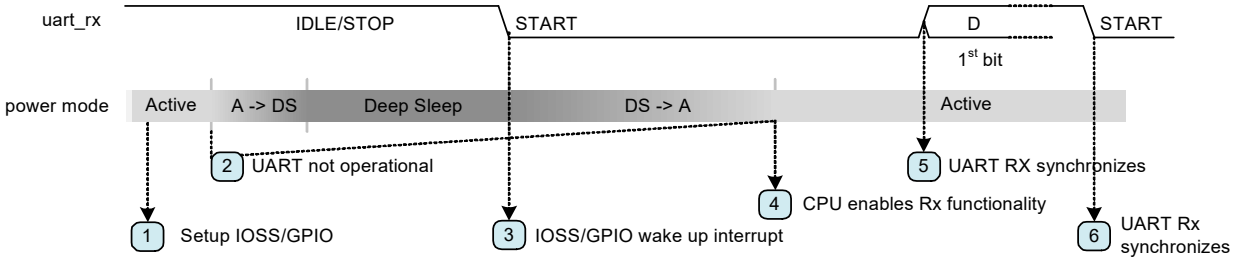
Start skipping applies only to receive functionality. The standard UART mode supports “start skipping”. Regular receive operation synchronizes on the START bit period (a 1 to 0 transition on the UART RX line), start skipping receive operation synchronizes on the first received data frame bit, which must be a ‘1’ (a 0 to 1 transition on UART RX).

Start skipping is used to allow for wake up from system Deep Sleep mode using UART. The process is described as follows:

1. Before entering Deep Sleep power mode, UART receive functionality is disabled and the GPIO is programmed to set an interrupt cause to ‘1’ when UART RX line has a ‘1’ to ‘0’ transition (START bit).
2. While in Deep Sleep mode, the UART receive functionality is not functional.
3. The GPIO interrupt is activated on the START bit and the system transitions from Deep Sleep to Active power mode.
4. The CPU enables UART receive functionality, with UART_RX_CTRL.SKIP_START bitfield set to ‘1’.
5. The UART receiver synchronizes data frame receipt on the next ‘0’ to ‘1’ transition. If the UART receive functionality is enabled in time, this is the transition from the START bit to the first received data frame bit.
6. The UART receiver proceeds with normal operation; that is, synchronization of successive data frames is on the START bit period.

Figure 28-19 illustrates the process.

Figure 28-19. UART Start Skip and Wakeup from Deep Sleep



Note that the above process works only for lower baud rates. The Deep Sleep to Active power mode transition and CPU enabling the UART receive functionality should take less than 1-bit period to ensure that the UART receiver is active in time to detect the '0' to '1' transition.

In step 4 of the above process, the firmware takes some time to finish the wakeup interrupt routine and enable the UART receive functionality before the block can detect the input rising edge on the UART RX line.

If the above steps cannot be completed in less than 1 bit time, first send a "dummy" byte to the device to wake it up before sending real UART data. In this case, the SKIP_START bit can be left as 0. For more information on how to perform this in firmware, visit the UART section of the PDL.

Break Detection

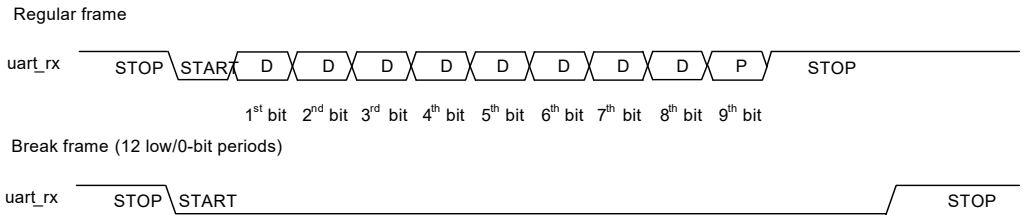
Break detection is supported in the standard UART mode. This functionality detects when UART RX line is low (0) for more than UART_RX_CTRL.BREAK_WIDTH bit periods. The break width should be larger than the maximum number of low (0) bit periods in a regular data transfer, plus an additional 1-bit period. The additional 1-bit period is a minimum requirement and preferably should be larger. The additional bit periods account for clock inaccuracies between transmitter and receiver.

For example, for an 8-bit data frame with parity support, the maximum number of low (0) bit periods is 10 (START bit, 8 '0' data frame bits, and one '0' parity bit). Therefore, the break width should be larger than 10 + 1 = 11 (UART_RX_CTRL.BREAK_WIDTH can be set to 11).

Note that the break detection applies only to receive functionality. A UART transmitter can generate a break by temporarily increasing TX_CTRL.DATA_WIDTH and transmitting an all zeroes data frame. A break is used by the transmitter to signal a special condition to the receiver. This condition may result in a reset, shut down, or initialization sequence at the receiver.

Break detection is part of the LIN protocol. When a break is detected, the INTR_RX.BREAK_DETECT interrupt cause is set to '1'. Figure 28-20 illustrates a regular data frame and break frame (8-bit data frame, parity support, and a break width of 12-bit periods).

Figure 28-20. UART – Regular Frame and Break Frame



Flow Control

The standard UART mode supports flow control. Modem flow control controls the pace at which the transmitter transfers data to the receiver. Modem flow control is enabled through the UART_FLOW_CTRL.CTS_ENABLED register field. When this field is '0', the transmitter transfers data when its TX FIFO is not empty. When '1', the transmitter transfers data when UART CTS line is active and its TX FIFO is not empty.

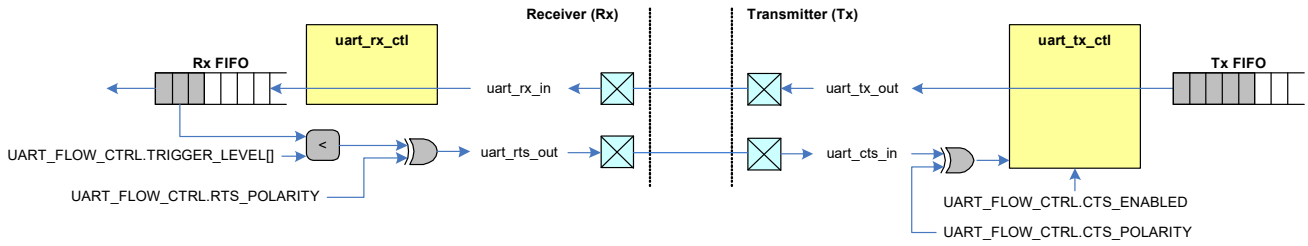
Note that the flow control applies only to TX functionality. Two UART side-band signal are used to implement flow control:

- UART RTS (uart_rts_out): This is an output signal from the receiver. When active, it indicates that the receiver is ready to receive data (RTS: Ready to Send).
- UART CTS (uart_cts_in): This is an input signal to the transmitter. When active, it indicates that the transmitter can transfer data (CTS: Clear to Send).

The receiver's uart_rts_out signal is connected to the transmitter's uart_cts_in signal. The receiver's uart_rts_out signal is derived by comparing the number of used receive FIFO entries with the UART_FLOW_CTRL.TRIGGER_LEVEL field. If the number of used receive FIFO entries are less than UART_FLOW_CTRL.TRIGGER_LEVEL, uart_rts_out is activated.

Typically, the UART side-band signals are active low. However, sometimes active high signaling is used. Therefore, the polarity of the side-band signals can be controlled using bitfields UART_FLOW_CTRL.RTS_POLARITY and UART_FLOW_CTRL.CTS_POLARITY. Figure 28-21 gives an overview of the flow control functionality.

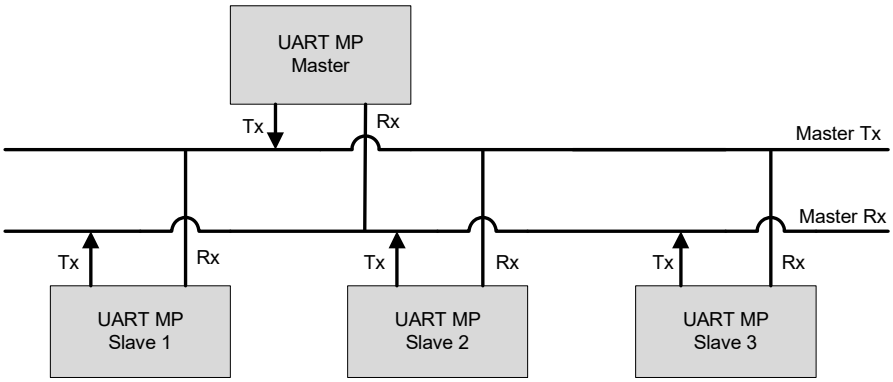
Figure 28-21. UART Flow Control Connection



UART Multi-Processor Mode

The UART_MP (multi-processor) mode is defined with single-master-multi-slave topology, as Figure 28-22 shows. This mode is also known as UART 9-bit protocol because the data field is nine bits wide. UART_MP is part of Standard UART mode.

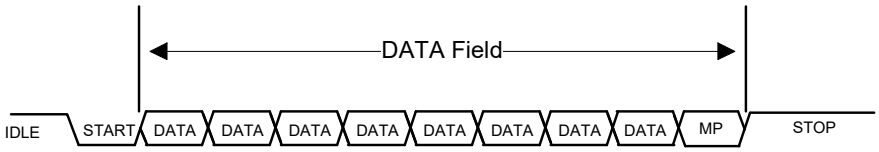
Figure 28-22. UART MP Mode Bus Connections



The main properties of UART_MP mode are:

- Single master with multiple slave concept (multi-drop network).
- Each slave is identified by a unique address.
- Using 9-bit data field, with the ninth bit as address/data flag (MP bit). When set high, it indicates an address byte; when set low it indicates a data byte. A data frame is illustrated in Figure 28-23.
- Parity bit is disabled.

Figure 28-23. UART MP Address and Data Frame



The SCB can be used as either master or slave device in UART_MP mode. Both SCB_TX_CTRL and SCB_RX_CTRL registers should be set to 9-bit data frame size. When the SCB works as UART_MP master device, the firmware changes the MP flag for every address or data frame. When it works as UART_MP slave device, the MP_MODE field of the SCB_UART_RX_CTRL register should be set to '1'. The SCB_RX_MATCH register should be set for the slave address and address mask. The matched address is written in the RX_FIFO when ADDR_ACCEPT field of the SCB_CTRL register is set to '1'. If the received address does not match its own address, then the interface ignores the following data, until next address is received for compare.

Configuring the SCB as Standard UART Interface

To configure the SCB as a standard UART interface, set various register bits in the following order:

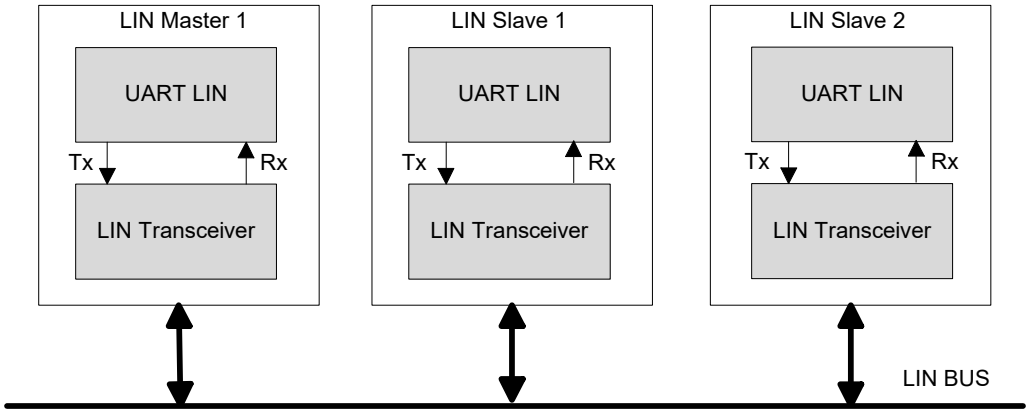
1. Configure the SCB as UART interface by writing '10b' to the MODE field (bits [25:24]) of the SCB_CTRL register.
2. Configure the UART interface to operate as a Standard protocol by writing '00' to the MODE field (bits [25:24]) of the SCB_UART_CTRL register.
3. To enable the UART MP Mode or UART LIN Mode, write '1' to the MP_MODE (bit 10) or LIN_MODE (bit 12) respectively of the SCB_UART_RX_CTRL register.
4. Follow steps 2 to 4 described in "Enabling and Initializing the UART" on page 348.

For more information on these registers, see the registers TRM.

28.4.3.2 UART Local Interconnect Network (LIN) Mode

The LIN protocol is supported by the SCB as part of the standard UART. LIN is designed with single-master-multi-slave topology. There is one master node and multiple slave nodes on the LIN bus. The SCB UART supports both LIN master and slave functionality. The LIN specification defines both physical layer (layer 1) and data link layer (layer 2). Figure 28-24 illustrates the UART_LIN and LIN transceiver.

Figure 28-24. UART_LIN and LIN Transceiver

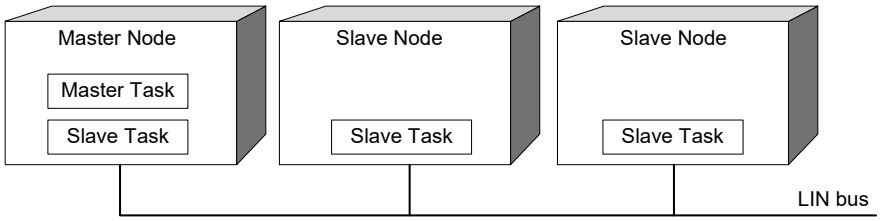


LIN protocol defines two tasks:

- Master task: This task involves sending a header packet to initiate a LIN transfer.
- Slave task: This task involves transmitting or receiving a response.

The master node supports master task and slave task; the slave node supports only slave task, as shown in Figure 28-25.

Figure 28-25. LIN Bus Nodes and Tasks

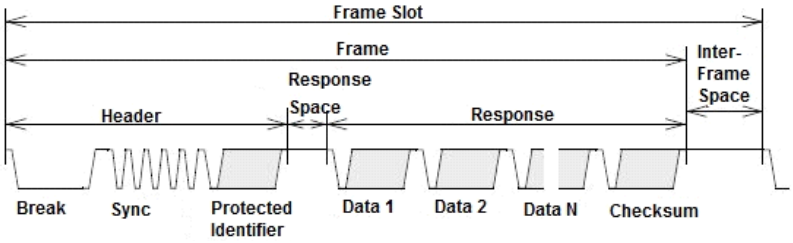


LIN Frame Structure

LIN is based on the transmission of frames at pre-determined moments of time. A frame is divided into header and response fields, as shown in Figure 28-26.

- The header field consists of:
  - Break field (at least 13 bit periods with the value '0').
  - Sync field (a 0x55 byte frame). A sync field can be used to synchronize the clock of the slave task with that of the master task.
  - Identifier field (a frame specifying a specific slave).
- The response field consists of data and checksum.

Figure 28-26. LIN Frame Structure

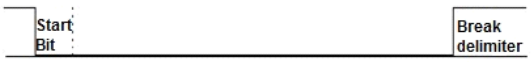


In LIN protocol communication, the least significant bit (LSb) of the data is sent first and the most significant bit (MSb) last. The start bit is encoded as zero and the stop bit is encoded as one. The following sections describe all the byte fields in the LIN frame.

Break Field

Every new frame starts with a break field, which is always generated by the master. The break field has logical zero with a minimum of 13 bit times and followed by a break delimiter. The break field structure is as shown in Figure 28-27.

Figure 28-27. LIN Break Field



Sync Field

This is the second field transmitted by the master in the header field; its value is 0x55. A sync field can be used to synchronize the clock of the slave task with that of the master task for automatic baud rate detection. Figure 28-28 shows the LIN sync field structure.

Figure 28-28. LIN Sync Field

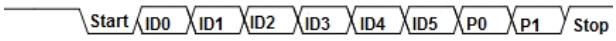
Protected Identifier (PID) Field

A protected identifier field consists of two sub-fields: the frame identifier (bits 0-5) and the parity (bit 6 and bit 7). The PID field structure is shown in Figure 28-29.

- Frame identifier: The frame identifiers are divided into three categories
  - Values 0 to 59 (0x3B) are used for signal carrying frames
  - 60 (0x3C) and 61 (0x3D) are used to carry diagnostic and configuration data
  - 62 (0x3E) and 63 (0x3F) are reserved for future protocol enhancements
- Parity: Frame identifier bits are used to calculate the parity

Figure 28-29 shows the PID field structure.

Figure 28-29. PID Field



Data. In LIN, every frame can carry a minimum of one byte and maximum of eight bytes of data. Here, the LSb of the data byte is sent first and the MSb of the data byte is sent last.

Checksum. The checksum is the last byte field in the LIN frame. It is calculated by inverting the 8-bit sum along with carryover of all data bytes only or the 8-bit sum with the carryover of all data bytes and the PID field. There are two types of checksums in LIN frames. They are:

- Classic checksum: the checksum calculated over all the data bytes only (used in LIN 1.x slaves).
- Enhanced checksum: the checksum calculated over all the data bytes along with the protected identifier (used in LIN 2.x slaves).

LIN Frame Types

The type of frame refers to the conditions that need to be valid to transmit the frame. According to the LIN specification, there are five different types of LIN frames. A node or cluster does not have to support all frame types.

Unconditional Frame. These frames carry the signals and their frame identifiers (of 0x00 to 0x3B range). The subscriber will receive the frames and make it available to the application; the publisher of the frame will provide the response to the header.

Event-Triggered Frame. The purpose of an event-triggered frame is to increase the responsiveness of the LIN cluster without assigning too much of the bus bandwidth to polling of multiple slave nodes with seldom occurring events. Event-triggered frames carry the response of one or more unconditional frames. The unconditional frames associated with an event triggered frame should:

- Have equal length
- Use the same checksum model (either classic or enhanced)
- Reserve the first data field to its protected identifier
- Be published by different slave nodes
- Not be included directly in the same schedule table as the event-triggered frame

Sporadic Frame. The purpose of the sporadic frames is to merge some dynamic behavior into the schedule table without affecting the rest of the schedule table. These frames have a group of unconditional frames that share the frame slot. When the sporadic frame is due for transmission, the unconditional frames are checked whether they have any updated signals. If no signals are updated, no frame will be transmitted and the frame slot will be empty.

Diagnostic Frames. Diagnostic frames always carry transport layer, and contains eight data bytes.

The frame identifier for diagnostic frame is:

- Master request frame (0x3C), or
- Slave response frame (0x3D)

Before transmitting a master request frame, the master task queries its diagnostic module to see whether it will be transmitted or whether the bus will be silent. A slave response frame header will be sent unconditionally. The slave tasks publish and subscribe to the response according to their diagnostic modules.

Reserved Frames. These frames are reserved for future use; their frame identifiers are 0x3E and 0x3F.

LIN Go-To-Sleep and Wake-Up

The LIN protocol has the feature of keeping the LIN bus in Sleep mode, if the master sends the go-to-sleep command. The go-to-sleep command is a master request frame (ID = 0x3C) with the first byte field is equal to 0x00 and rest set to 0xFF. The slave node application may still be active after the go-to-sleep command is received. This behavior is application specific. The LIN slave nodes automatically enter Sleep mode if the LIN bus inactivity is more than four seconds.

Wake-up can be initiated by any node connected to the LIN bus – either LIN master or any of the LIN slaves by forcing the bus to be dominant for 250 µs to 5 ms. Each slave should detect the wakeup request and be ready to process headers within 100 ms. The master should also detect the wakeup request and start sending headers when the slave nodes are active.

To support LIN, a dedicated (off-chip) line driver/receiver is required. Supply voltage range on the LIN bus is 7 V to 18 V.

Typically, LIN line drivers will drive the LIN line with the value provided on the SCB TX line and present the value on the LIN line to the SCB RX line. By comparing TX and RX lines in the SCB, bus collisions can be detected (indicated by the SCB_UART_ARB_LOST field of the SCB_INTR_TX register).

28.4.3.3 SmartCard (ISO7816)

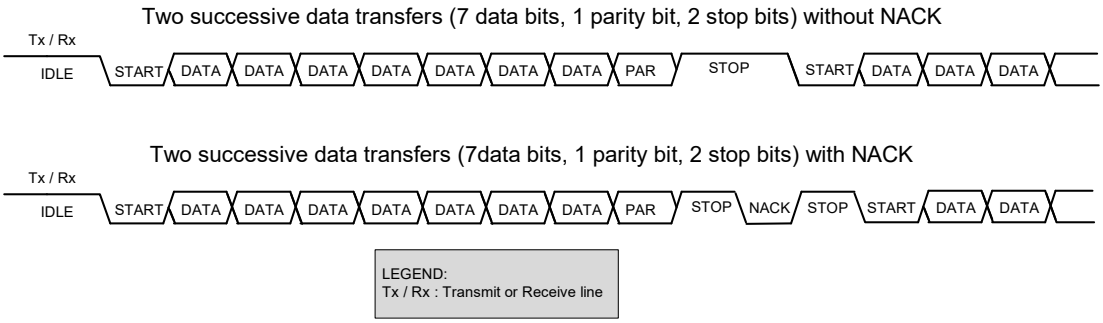
ISO7816 is asynchronous serial interface, defined with single-master-single slave topology. ISO7816 defines both Reader (master) and Card (slave) functionality. For more information, refer to the ISO7816 Specification. Only the master (reader) function is supported by the SCB. This block provides the basic physical layer support with asynchronous character transmission. The UART_TX line is connected to SmartCard I/O line by internally multiplexing between UART_TX and UART_RX control modules.

The SmartCard transfer is similar to a UART transfer, with the addition of a negative acknowledgement (NACK) that may be sent from the receiver to the transmitter. A NACK is always '0'. Both master and slave may drive the same line, although never at the same time.

A SmartCard transfer has the transmitter drive the start bit and data bits (and optionally a parity bit). After these bits, it enters its stop period by releasing the bus. Releasing results in the line being '1' (the value of a stop bit). After one bit transfer period into the stop period, the receiver may drive a NACK on the line (a value of '0') for one bit transfer period. This NACK is observed by the transmitter, which reacts by extending its stop period by one bit transfer period. For this protocol to work, the stop period should be longer than one bit transfer period. Note that a data transfer with a NACK takes one bit transfer period longer, than a data transfer without a NACK. Typically, implementations use a tristate driver with a pull-up resistor, such that when the line is not transmitting data or transmitting the Stop bit, its value is '1'.

Figure 28-30 illustrates the SmartCard protocol.

Figure 28-30. SmartCard Example



The communication Baud rate while using SmartCard is given as:

Baud rate = Fscbclk/Oversample

Configuring SCB as UART SmartCard Interface

To configure the SCB as a UART SmartCard interface, set various register bits in the following order; note that ModusToolbox does all this automatically with the help of GUIs. For more information on these registers, see the registers TRM.

1. Configure the SCB as UART interface by writing '10b' to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Configure the UART interface to operate as a SmartCard protocol by writing '01' to the MODE (bits [25:24]) of the SCB_UART_CTRL register.
3. Follow steps 2 to 4 described in "Enabling and Initializing the UART" on page 348.

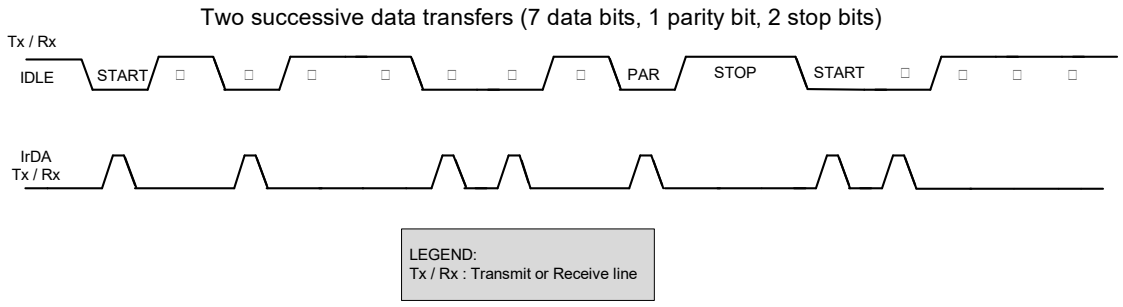
28.4.3.4 Infrared Data Association (IrDA)

The SCB supports the IrDA protocol for data rates of up to 115.2 kbps using the UART interface. It supports only the basic physical layer of IrDA protocol with rates less than 115.2 kbps. Hence, the system instantiating this block must consider how to implement a complete IrDA communication system with other available system resources.

The IrDA protocol adds a modulation scheme to the UART signaling. At the transmitter, bits are modulated. At the receiver, bits are demodulated. The modulation scheme uses a Return-to-Zero-Inverted (RZI) format. A bit value of '0' is signaled by a short '1' pulse on the line and a bit value of '1' is signaled by holding the line to '0'. For these data rates (<=115.2 kbps), the RZI modulation scheme is used and the pulse duration is 3/16 of the bit period. The sampling clock frequency should be set 16 times the selected baud rate, by configuring the SCB_OVS field of the SCB_CTRL register. In addition, the PSoC 6 MCU SCB supports a low-power IrDA receiver mode, which allows it to detect pulses with a minimum width of 1.41 µs.

Different communication speeds under 115.2 kbps can be achieved by configuring clk_scb frequency. Additional allowable rates are 2.4 kbps, 9.6 kbps, 19.2 kbps, 38.4 kbps, and 57.6 kbps. Figure 28-31 shows how a UART transfer is IrDA modulated.

Figure 28-31. IrDA Example



Configuring the SCB as a UART IrDA Interface

To configure the SCB as a UART IrDA interface, set various register bits in the following order; note that ModusToolbox does all this automatically with the help of GUIs. For more information on these registers, see the registers TRM.

1. Configure the SCB as a UART interface by writing '10b' to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Configure the UART interface to operate as IrDA protocol by writing '10' to the MODE (bits [25:24]) of the SCB_UART_CTRL register.
3. Enable the Median filter on the input interface line by writing '1' to MEDIAN (bit 9) of the SCB_RX_CTRL register.
4. Configure the SCB as described in "Enabling and Initializing the UART" on page 348.

28.4.4 Clocking and Oversampling

The UART protocol is implemented using clk_scb as an oversampled multiple of the baud rate. For example, to implement a 100-kHz UART, clk_scb could be set to 1 MHz and the oversample factor set to ‘10’. The oversampling is set using the SCB_CTRL.OVS register field. The oversampling value is SCB_CTRL.OVS + 1. In the UART standard sub-mode (including LIN) and the SmartCard sub-mode, the valid range for the OVS field is [7, 15].

In UART transmit IrDA sub-mode, this field indirectly specifies the oversampling. Oversampling determines the interface clock per bit cycle and the width of the pulse. This sub-mode has only one valid OVS value–16 (which is a value of 0 in the OVS field of the SCB_CTRL register); the pulse width is roughly 3/16 of the bit period (for all bit rates).

In UART receive IrDA sub-mode (1.2, 2.4, 9.6, 19.2, 38.4, 57.6, and 115.2 kbps), this field indirectly specifies the oversampling. In normal transmission mode, this pulse is approximately 3/16 of the bit period (for all bit rates). In low-power transmission mode, this pulse is potentially smaller (down to 1.62 µs typical and 1.41 µs minimal) than 3/16 of the bit period (for less than 115.2 kbps bit rates).

Pulse widths greater than or equal to two SCB input clock cycles are guaranteed to be detected by the receiver. Pulse widths less than two clock cycles and greater than or equal to one SCB input clock cycle may be detected by the receiver. Pulse widths less than one SCB input clock cycle will not be detected by the receiver. Note that the SCB_RX_CTRL.MEDIAN should be set to ‘1’ for IrDA receiver functionality.

The SCB input clock and the oversampling together determine the IrDA bit rate. Refer to the registers TRM for more details on the OVS values for different baud rates.

28.4.5 Enabling and Initializing the UART

The UART must be programmed in the following order:

1. Program protocol specific information using the UART_TX_CTRL, UART_RX_CTRL, and UART_FLOW_CTRL registers. This includes selecting the submodes of the protocol, transmitter-receiver functionality, and so on.
2. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers.
   a. Specify the data frame width.
   b. Specify whether MSb or LSb is the first bit to be transmitted or received.
3. Program the transmitter and receiver FIFOs using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers, respectively.
   a. Set the trigger level.
   b. Clear the transmitter and receiver FIFO and Shift registers.
4. Enable the block (write a ‘1’ to the ENABLE bit of the SCB_CTRL register). After the block is enabled, control bits should not be changed. Changes should be made after disabling the block; for example, to modify the operation mode (from SmartCard to IrDA). The change takes effect only after the block is re-enabled. Note that re-enabling the block causes re-initialization and the associated state is lost (for example FIFO content).

28.4.6 I/O Pad Connection

28.4.6.1 Standard UART Mode

Figure 28-32 and Table 28-7 list the use of the I/O pads for the Standard UART mode.

Figure 28-32. Standard UART Mode I/O Pad Connections

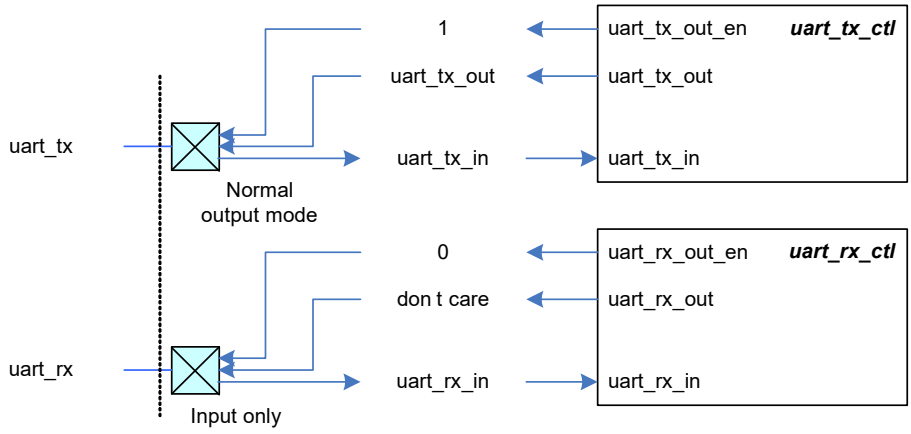


Table 28-7. UART I/O Pad Connection Usage

I/O Pads	Drive Mode	On-chip I/O Signals	Usage
uart_tx	Normal output mode	uart_tx_out_en uart_tx_out	Transmit a data element
uart_rx	Input only	uart_rx_in	Receive a data element

28.4.6.2 SmartCard Mode

Figure 28-33 and Table 28-8 list the use of the I/O pads for the SmartCard mode.

Figure 28-33. SmartCard Mode I/O Pad Connections

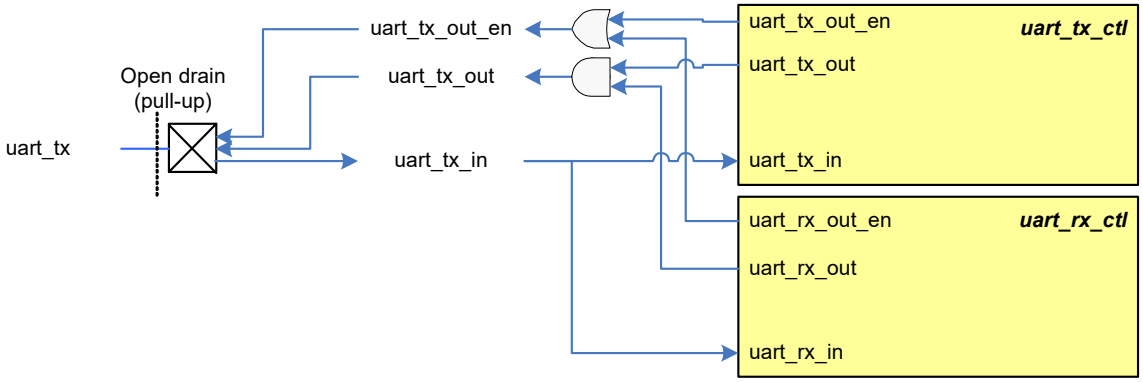


Table 28-8. SmartCard Mode I/O Pad Connections

I/O Pads	Drive Mode	On-chip I/O Signals	Usage
uart_tx	Open drain with pull-up	uart_tx_in	Used to receive a data element. Receive a negative acknowledge-ment of a transmitted data element
		uart_tx_out_en uart_tx_out	Transmit a data element. Transmit a negative acknowledgement to a received data element.

28.4.6.3 LIN Mode

Figure 28-34 and Table 28-9 list the use of the I/O pads for LIN mode.

Figure 28-34. LIN Mode I/O Pad Connections

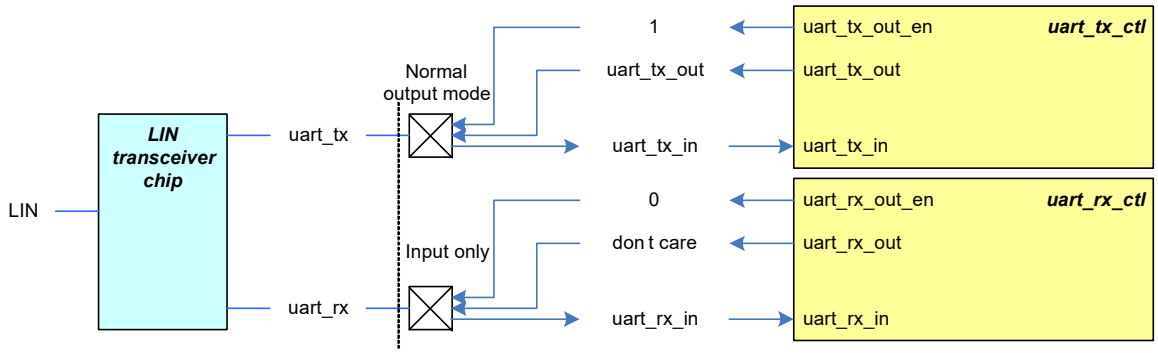


Table 28-9. LIN Mode I/O Pad Connections

I/O Pads	Drive Mode	On-chip I/O Signals	Usage
uart_tx	Normal output mode	uart_tx_out_en uart_tx_out	Transmit a data element.
uart_rx	Input only	uart_rx_in	Receive a data element.

28.4.6.4 IrDA Mode

Figure 28-35 and Table 28-10 list the use of the I/O pads for IrDA mode.

Figure 28-35. IrDA Mode I/O Pad Connections

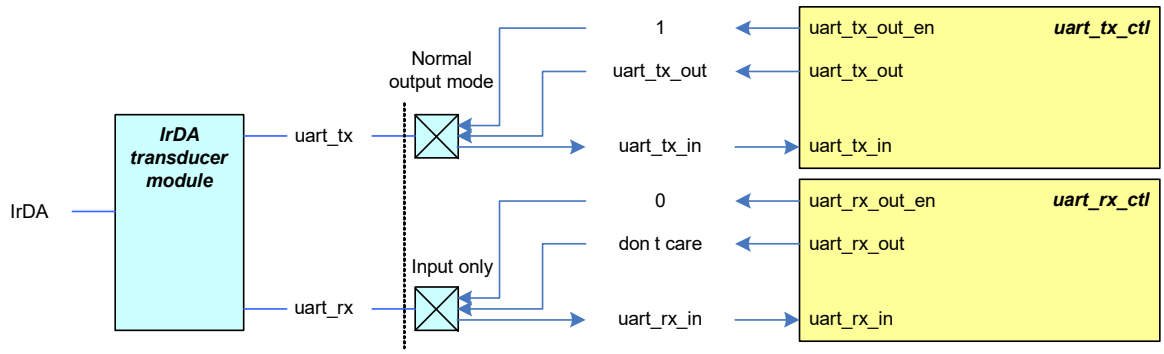


Table 28-10. IrDA Mode I/O Pad Connections

I/O Pads	Drive Mode	On-chip I/O Signals	Usage
uart_tx	Normal output mode	uart_tx_out_en uart_tx_out	Transmit a data element.
uart_rx	Input only	uart_rx_in	Receive a data element.

28.4.7 UART Registers

The UART interface is controlled using a set of 32-bit registers listed in Table 28-11. For more information on these registers, see the registers TRM.

Table 28-11. UART Registers

Register Name	Operation
SCB_CTRL	Enables the SCB; selects the type of serial interface (SPI, UART, I^2C)
SCB_UART_CTRL	Used to select the sub-modes of UART (standard UART, SmartCard, IrDA), also used for local loop back control.
SCB_UART_RX_STATUS	Used to specify the BR_COUNTER value that determines the bit period. This is used to set the accuracy of the SCB clock. This value provides more granularity than the OVS bit in SCB_CTRL register.
SCB_UART_TX_CTRL	Used to specify the number of stop bits, enable parity, select the type of parity, and enable retransmission on NACK.
SCB_UART_RX_CTRL	Performs same function as SCB_UART_TX_CTRL but is also used for enabling multi processor mode, LIN mode drop on parity error, and drop on frame error.
SCB_TX_CTRL	Used to specify the data frame width and to specify whether MSb or LSb is the first bit in transmission.
SCB_RX_CTRL	Performs the same function as that of the SCB_TX_CTRL register, but for the receiver. Also decides whether a median filter is to be used on the input interface lines.
SCB_UART_FLOW_CONTROL	Configures flow control for UART transmitter.

28.5 Inter Integrated Circuit (I^2C)

This section explains the I^2C implementation in the PSoC 6 MCU. For more information on the I^2C protocol specification, refer to the I^2C-bus specification available on the NXP website.

28.5.1 Features

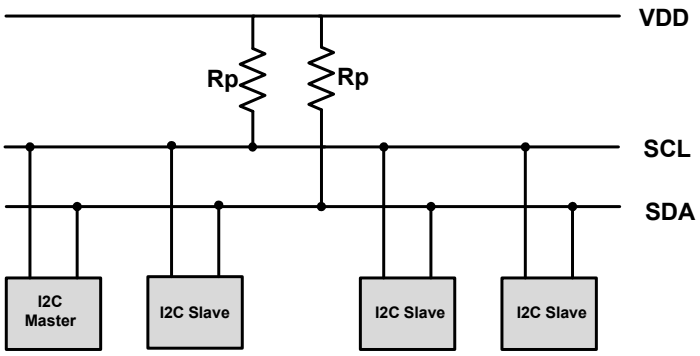
This block supports the following features:

- Master, slave, and master/slave mode
- Standard-mode (100 kbps), fast-mode (400 kbps), and fast-mode plus (1000 kbps) data-rates
- 7-bit slave addressing
- Clock stretching
- Collision detection
- Programmable oversampling of I^2C clock signal (SCL)
- Auto ACK when RX FIFO not full, including address
- General address detection
- FIFO Mode
- EZ and CMD_RESP modes

28.5.2 General Description

Figure 28-36 illustrates an example of an I^2C communication network.

Figure 28-36. I^2C Interface Block Diagram



The standard I^2C bus is a two wire interface with the following lines:

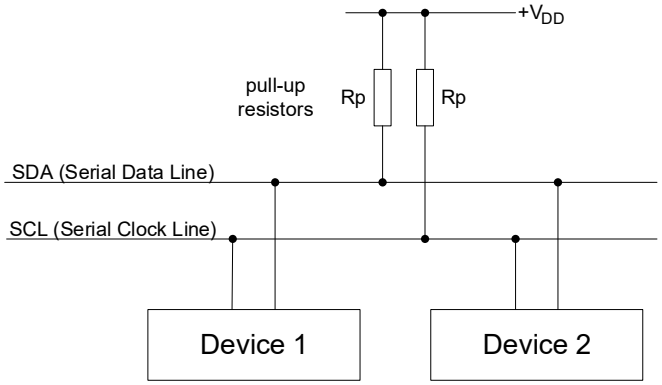
- Serial Data (SDA)
- Serial Clock (SCL)

I^2C devices are connected to these lines using open collector or open-drain output stages, with pull-up resistors (Rp). A simple master/slave relationship exists between devices. Masters and slaves can operate as either transmitter or receiver. Each slave device connected to the bus is software addressable by a unique 7-bit address.

28.5.3 External Electrical Connections

As shown in Figure 28-37, the I²C bus requires external pull-up resistors. The pull-up resistors (R_P) are primarily determined by the supply voltage, bus speed, and bus capacitance. For detailed information on how to calculate the optimum pull-up resistor value for your design Cypress recommends using the UM10204 I²C-bus specification and user manual Rev. 6, available from the NXP website at www.nxp.com.

Figure 28-37. Connection of Devices to the I2C Bus



For most designs, the default values shown inTable 28-12 provide excellent performance without any calculations. The default values were chosen to use standard resistor values between the minimum and maximum limits.

Table 28-12. Recommended Default Pull-up Resistor Values

Standard Mode (0 – 100 kbps)	Fast Mode (0 – 400 kbps)	Fast Mode Plus (0 – 1000 kbps)	Units
4.7 k, 5%	1.74 k, 1%	620, 5%	Ω

These values work for designs with 1.8 V to 5.0 V V_{DD}, less than 200 pF bus capacitance (CB), up to 25 µA of total input leakage (I_{IL}), up to 0.4 V output voltage level (V_{OL}), and a max V_{IH} of 0.7 * V_{DD}. Calculation of custom pull-up resistor values is required if your design does not meet the default assumptions, you use series resistors (R_S) to limit injected noise, or you want to maximize the resistor value for low power consumption. Calculation of the ideal pull-up resistor value involves finding a value between the limits set by three equations detailed in the NXP I²C specification. These equations are:

$$R_{PMIN} = (V_{DD}(max) - V_{OL}(max)) / I_{OL}(min)$$

Equation 28-4

$$R_{PMAX} = T_R(max) / 0.8473 \times C_B(max)$$

Equation 28-5

$$R_{PMAX} = V_{DD}(min) - (VIH(min) + VNH(min)) / IIH(max)$$

Equation 28-6

Equation parameters:

- V_{DD} = Nominal supply voltage for I²C bus
- V_{OL} = Maximum output low voltage of bus devices
- I_{OL}= Low-level output current from I²C specification
- T_R = Rise time of bus from I²C specification
- C_B = Capacitance of each bus line including pins and PCB traces
- V_{IH} = Minimum high-level input voltage of all bus devices
- V_{NH} = Minimum high-level input noise margin from I²C specification
- I_{IH} = Total input leakage current of all devices on the bus

The supply voltage (V_{DD}) limits the minimum pull-up resistor value due to bus devices maximum low output voltage (V_{OL}) specifications. Lower pull-up resistance increases current through the pins and can therefore exceed the spec conditions of V_{OH}. Equation 26-4 is derived using Ohm's law to determine the minimum resistance that will still meet the V_{OL} specification at 3 mA for standard and fast modes, and 20 mA for fast mode plus at the given V_{DD}.

Equation 26-5 determines the maximum pull-up resistance due to bus capacitance. Total bus capacitance is comprised of all pin, wire, and trace capacitance on the bus. The higher the bus capacitance the lower the pull-up resistance required to meet the specified bus speeds rise time due to RC delays. Choosing a pull-up resistance higher than allowed can result in failing timing requirements resulting in communication errors. Most designs with five of fewer I^2C devices and up to 20 centimeters of bus trace length have less than 100 pF of bus capacitance.

A secondary effect that limits the maximum pull-up resistor value is total bus leakage calculated in Equation 26-6. The primary source of leakage is I/O pins connected to the bus. If leakage is too high, the pull-ups will have difficulty maintaining an acceptable V_{IH} level causing communication errors. Most designs with five or fewer I^2C devices on the bus have less than 10 µA of total leakage current.

28.5.4 Terms and Definitions

Table 28-13 explains the commonly used terms in an I^2C communication network.

Table 28-13. Definition of I^2C Bus Terminology

Term	Description
Transmitter	The device that sends data to the bus
Receiver	The device that receives data from the bus
Master	The device that initiates a transfer, generates clock signals, and terminates a transfer
Slave	The device addressed by a master
Multi-master	More than one master can attempt to control the bus at the same time
Arbitration	Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted
Synchronization	Procedure to synchronize the clock signals of two or more devices

28.5.4.1 Clock Stretching

When a slave device is not yet ready to process data, it may drive a '0' on the SCL line to hold it down. Due to the implementation of the I/O signal interface, the SCL line value will be '0', independent of the values that any other master or slave may be driving on the SCL line. This is known as clock stretching and is the only situation in which a slave drives the SCL line. The master device monitors the SCL line and detects it when it cannot generate a positive clock pulse ('1') on the SCL line. It then reacts by delaying the generation of a positive edge on the SCL line, effectively synchronizing with the slave device that is stretching the clock. The SCB on the PSoC 6 MCU can and will stretch the clock.

28.5.4.2 Bus Arbitration

The I^2C protocol is a multi-master, multi-slave interface. Bus arbitration is implemented on master devices by monitoring the SDA line. Bus collisions are detected when the master observes an SDA line value that is not the same as the value it is driving on the SDA line. For example, when master 1 is driving the value '1' on the SDA line and master 2 is driving the value '0' on the SDA line, the actual line value will be '0' due to the implementation of the I/O signal interface. Master 1 detects the inconsistency and loses control of the bus. Master 2 does not detect any inconsistency and keeps control of the bus.

28.5.5 I^2C Modes of Operation

I^2C is a synchronous single master, multi-master, multi-slave serial interface. Devices operate in either master mode, slave mode, or master/slave mode. In master/slave mode, the device switches from master to slave mode when it is addressed. Only a single master may be active during a data transfer. The active master is responsible for driving the clock on the SCL line. Table 28-14 illustrates the I^2C modes of operation.

Table 28-14. I^2C Modes

Mode	Description
Slave	Slave only operation (default)
Master	Master only operation
Multi-master	Supports more than one master on the bus

Table 28-15 lists some common bus events that are part of an I²C data transfer. The Write Transfer and Read Transfer sections explain the I²C bus bit format during data transfer.

Table 28-15. I²C Bus Events Terminology

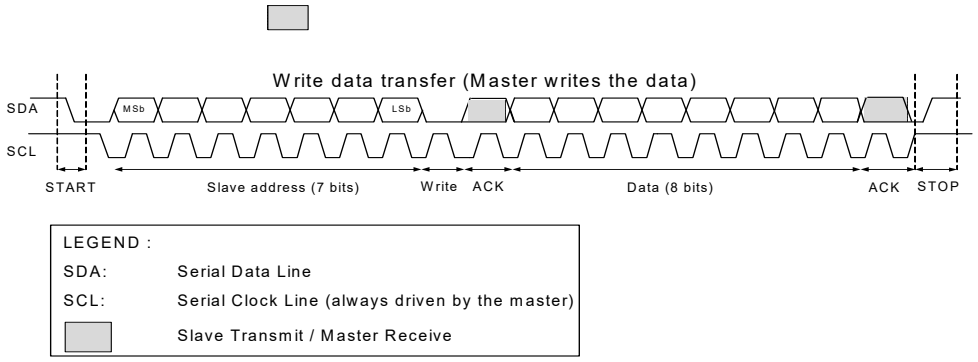
Bus Event	Description
START	A HIGH to LOW transition on the SDA line while SCL is HIGH
STOP	A LOW to HIGH transition on the SDA line while SCL is HIGH
ACK	The receiver pulls the SDA line LOW and it remains LOW during the HIGH period of the clock pulse, after the transmitter transmits each byte. This indicates to the transmitter that the receiver received the byte properly.
NACK	The receiver does not pull the SDA line LOW and it remains HIGH during the HIGH period of clock pulse after the transmitter transmits each byte. This indicates to the transmitter that the receiver did not receive the byte properly.
Repeated START	START condition generated by master at the end of a transfer instead of a STOP condition
DATA	SDA status change while SCL is LOW (data changing), and no change while SCL is HIGH (data valid)

With all of these modes, there are two types of transfer - read and write. In write transfer, the master sends data to slave; in read transfer, the master receives data from slave.

28.5.5.1 Write Transfer

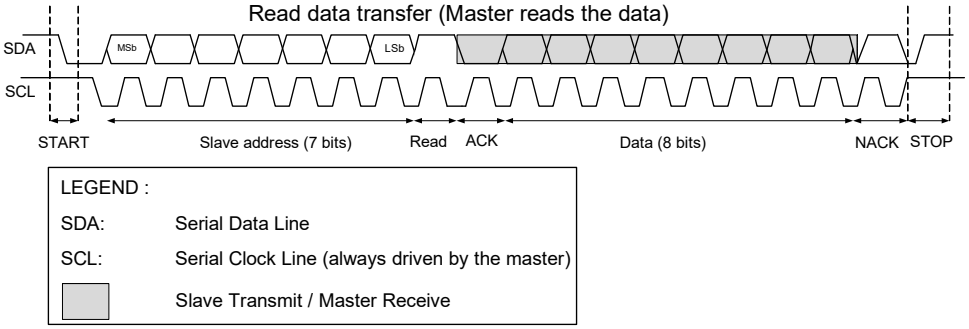
- A typical write transfer begins with the master generating a START condition on the I²C bus. The master then writes a 7-bit I²C slave address and a write indicator (‘0’) after the START condition. The addressed slave transmits an acknowledgment byte by pulling the data line low during the ninth bit time.
- If the slave address does not match any of the slave devices or if the addressed device does not want to acknowledge the request, it transmits a no acknowledgment (NACK) by not pulling the SDA line low. The absence of an acknowledgement, results in an SDA line value of ‘1’ due to the pull-up resistor implementation.
- If no acknowledgment is transmitted by the slave, the master may end the write transfer with a STOP event. The master can also generate a repeated START condition for a retry attempt.
- The master may transmit data to the bus if it receives an acknowledgment. The addressed slave transmits an acknowledgment to confirm the receipt of every byte of data written. Upon receipt of this acknowledgment, the master may transmit another data byte.
- When the transfer is complete, the master generates a STOP condition.

Figure 28-38. Master Write Data Transfer



28.5.5.2 Read Transfer

Figure 28-39. Master Read Data Transfer



- A typical read transfer begins with the master generating a START condition on the I^{2}C bus. The master then writes a 7-bit I^{2}C slave address and a read indicator (‘1’) after the START condition. The addressed slave transmits an acknowledgment by pulling the data line low during the ninth bit time.
- If the slave address does not match with that of the connected slave device or if the addressed device does not want to acknowledge the request, a no acknowledgment (NACK) is transmitted by not pulling the SDA line low. The absence of an acknowledgment, results in an SDA line value of ‘1’ due to the pull-up resistor implementation.
- If no acknowledgment is transmitted by the slave, the master may end the read transfer with a STOP event. The master can also generate a repeated START condition for a retry attempt.
- If the slave acknowledges the address, it starts transmitting data after the acknowledgment signal. The master transmits an acknowledgment to confirm the receipt of each data byte sent by the slave. Upon receipt of this acknowledgment, the addressed slave may transmit another data byte.
- The master can send a NACK signal to the slave to stop the slave from sending data bytes. This completes the read transfer.
- When the transfer is complete, the master generates a STOP condition.
- When the slave transmits a NACK, even if a new byte is written into the TX FIFO it will wait for the NACK to complete.

28.5.6 I^{2}C Buffer Modes

I^{2}C can operate in three different buffered modes – FIFO, EZ, and CMD_RESP modes. The buffer is used in different ways in each of the modes. The following subsections explain each of these buffered modes in detail.

28.5.6.1 FIFO Mode

The FIFO mode has a TX FIFO for the data being transmitted and an RX FIFO for the data being received. Each FIFO is constructed out of the SRAM buffer. The FIFOs are either 64 elements deep with 16-bit data elements or 128 elements deep with 8-bit data elements. The width of the data elements are configured using the CTRL.BYTE_MODE bitfield of the SCB. For I^{2}C, put the FIFO in BYTE mode because all transactions are a byte wide.

The FIFO mode operation is available only in Active and Sleep power modes, not in the Deep Sleep power mode. However, on the Deep Sleep-capable SCB the slave address can be used to wake the device from sleep.

A write access to the transmit FIFO uses register TX_FIFO_WR. A read access from the receive FIFO uses register RX_FIFO_RD.

Transmit and receive FIFO status information is available through status registers TX_FIFO_STATUS and RX_FIFO_STATUS. When in debug mode, a read from this register behaves as a read from the SCB_RX_FIFO_RD_SILENT register; that is, data will not be removed from the FIFO.

Each FIFO has a trigger output. This trigger output can be routed through the trigger mux to various other peripheral on the device such as DMA or TCPWMs. The trigger output of the SCB is controlled through the TRIGGER_LEVEL field in the RX_CTRL and TX_CTRL registers.

- For a TX FIFO a trigger is generated when the number of entries in the transmit FIFO is less than TX_FIFO_CTRL.TRIGGER_LEVEL.
- For the RX FIFO a trigger is generated when the number of entries in the FIFO is greater than the RX_FIFO_CTRL.TRIGGER_LEVEL.

Note that the DMA has a trigger deactivation setting. For the SCB this should be set to 16.

Active to Deep Sleep Transition

Before going to deep sleep ensure that all active communication is complete. This can be done by checking the BUS_BUSY bit in the I2C_Status register.

Ensure that the TX and RX FIFOs are empty as any data will be lost during deep sleep.

Before going to deep sleep the clock to the SCB needs to be disabled. This can be done by setting the SDA_IN_FILT_TRIM[1] bit in the I2C_CFG register to '0'.

Deep Sleep to Active Transition

EC_AM = 1, EC_OP = 0, FIFO Mode.

The following descriptions only apply to slave mode.

Master Write:

- I2C_CTRL.S_NOT_READY_ADDR_NACK = 0, I2C_CTRL.S_READY_ADDR_ACK = 1. The clock is stretched until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the clock stretch will be released
- I2C_CTRL.S_NOT_READY_ADDR_NACK = 0, I2C_CTRL.S_READY_ADDR_ACK = 0. The clock is stretched until SDA_IN_FILT_TRIM[1] is set to '1', and S_ACK or S_NACK is set in the I2C_S_CMD register.
- I2C_CTRL.S_NOT_READY_ADDR_NACK = 1, I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is NACK'd until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the slave will respond with an ACK to a master address.

Master Read:

- I2C_CTRL.S_NOT_READY_ADDR_NACK = 0, I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is stretched until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the clock stretch will be released.
- I2C_CTRL.S_NOT_READY_ADDR_NACK = 1, I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is NACK'd until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the slave will ACK.

  Note: When doing a repeated start after a write, wait until the UNDERFLOW interrupt status is asserted before setting the I2C_M_CMD.START bit and writing the new address into the TX_FIFO. Otherwise, the address in the FIFO will be sent as data and not as an address.

28.5.6.2 EZI2C Mode

The Easy I^2C (EZI2C) protocol is a unique communication scheme built on top of the I^2C protocol by Cypress. It uses a meta protocol around the standard I^2C protocol to communicate to an I^2C slave using indexed memory transfers. This removes the need for CPU intervention.

The EZI2C protocol defines a single memory buffer with an 8-bit address that indexes the buffer (256-entry array of 8-bit per entry is supported) located on the slave device. The EZ address is used to address these 256 locations. The CPU writes and reads to the memory buffer through the EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

The slave interface accesses the memory buffer using the current address. At the start of a transfer (I^2C START/RESTART), the base address is copied to the current address. A data element write or read operation is to the current address location. After the access, the current address is incremented by '1'.

If the current address equals the last memory buffer address (255), the current address is not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in EZ mode.

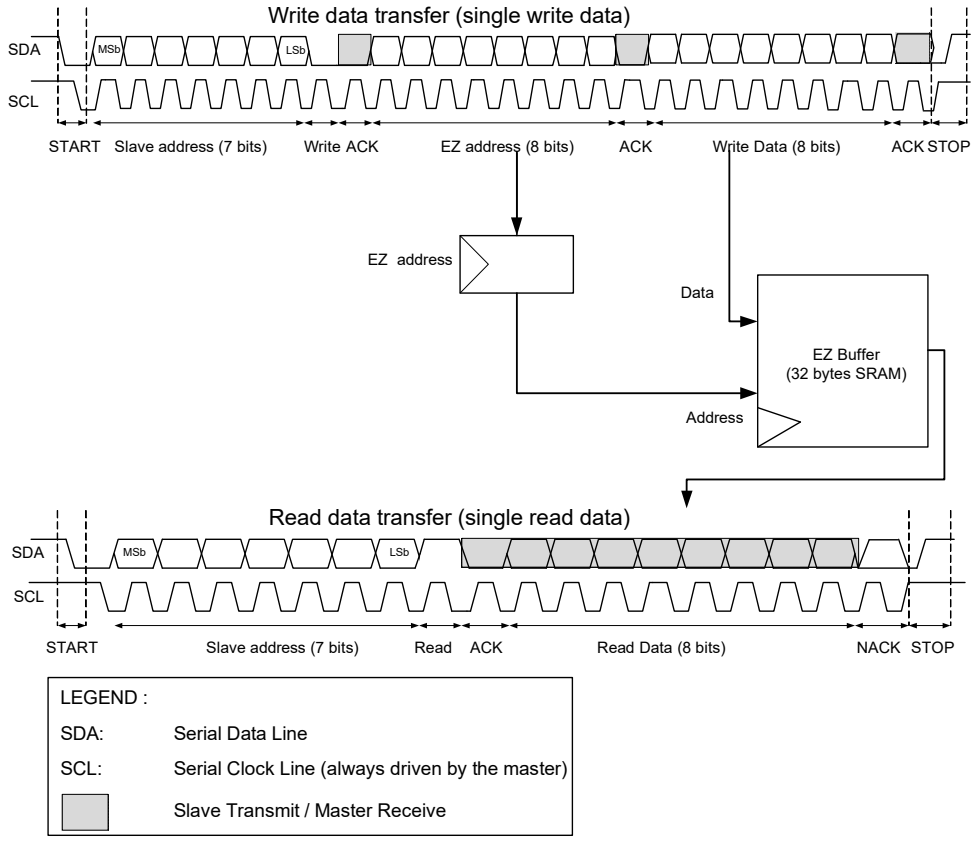
The I^2C base and current addresses are provided through I2C_STATUS. At the end of a transfer, the difference between the base and current addresses indicates how many read or write accesses were performed. The block provides interrupt cause fields to identify the end of a transfer. EZI2C can be implemented through firmware or hardware. All SCBs can implement EZI2C through a firmware implementation in both Active and Sleep power modes. The Deep Sleep SCB can implement a hardware- and firmware-based EZI2C with a Deep Sleep power mode. This document focuses on hardware-implemented EZI2C; for more information on software implementation, see the PDL.

EZI2C distinguishes three operation phases:

- Address phase: The master transmits an 8-bit address to the slave. This address is used as the slave base and current address.
- Write phase: The master writes 8-bit data element(s) to the slave's memory buffer. The slave's current address is set to the slave's base address. Received data elements are written to the current address memory location. After each memory write, the current address is incremented.
- Read phase: The master reads 8-bit data elements from the slave's memory buffer. The slave's current address is set to the slave's base address. Transmitted data elements are read from the current address memory location. After each memory read, the current address is incremented.

Note that a slave's base address is updated by the master and not by the CPU.

Figure 28-40. EZI2C Write and Read Data Transfer



Active to Deep Sleep Transition

Before going to deep sleep ensure that all active communication is complete. This can be done by checking the BUS_BUSY bit in the I2C_Status register.

Ensure that the TX and RX FIFOs are empty as any data will be lost during deep sleep.

Before going to deep sleep the clock to the SCB needs to be disabled. This can be done by setting the SDA_IN_FILT_TRIM[1] bit in the I2C_CFG register to ‘0’.

Deep Sleep to Active Transition

EC_AM = 1, EC_OP = 0, EZ Mode.

- S_NOT_READY_ADDR_NACK = 0, S_READY_ADDR_ACK = 1. The clock is stretched until SDA_IN_FILT_TRIM[1] is set to ‘1’. After that bit is set to 1, the clock stretch will be released.
- S_NOT_READY_ADDR_NACK = 1, S_READY_ADDR_ACK = x. The incoming address is NACK’d until SDA_IN_FILT_TRIM[1] is set to ‘1’. After that bit is set to 1, the slave will ACK.

28.5.6.3 Command-Response Mode

This mode has a single memory buffer, a base read address, a current read address, a base write address, and a current write address that are used to index the memory buffer. The base addresses are provided by the CPU. The current addresses are used by the slave to index the memory buffer for sequential accesses of the memory buffer. The memory buffer holds 256 8-bit data elements. The base and current addresses are in the range [0 to 255].

The CPU writes and reads to the memory buffer through the SCB_EZ_DATA registers. These are word accesses, but only the least significant byte of the word is used.

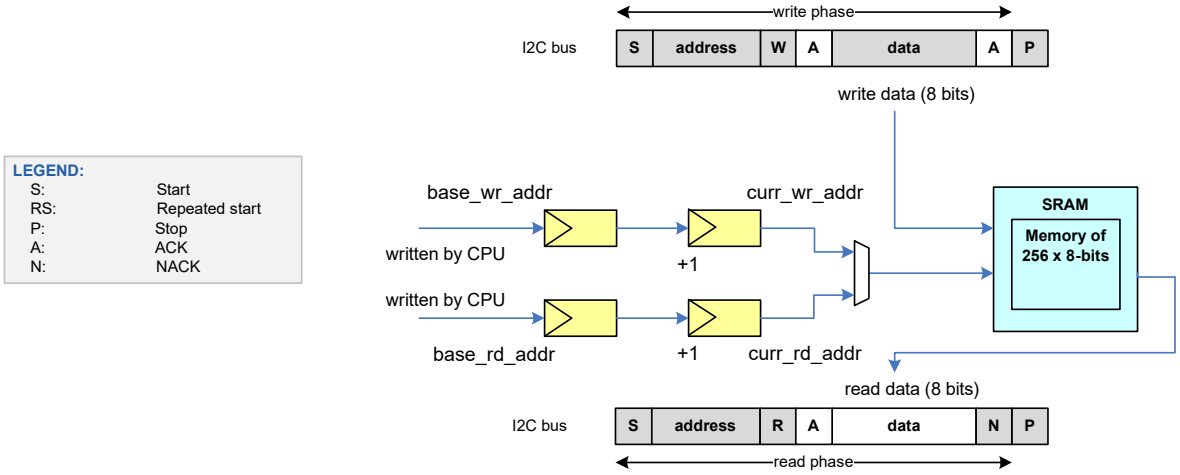
The slave interface accesses the memory buffer using the current addresses. At the start of a write transfer (I^2C START/RESTART), the base write address is copied to the current write address. A data element write is to the current write address location. After the write access, the current address is incremented by '1'. At the start of a read transfer, the base read address is copied to the current read address. A data element read is to the current read address location. After the read data element is transmitted, the current read address is incremented by '1'.

If the current addresses equal the last memory buffer address (255), the current addresses are not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in command-response mode.

The base addresses are provided through CMD_RESP_CTRL. The current addresses can be viewed in CMD_RESP_STATUS. At the end of a transfer (I^2C stop), the difference between a base and current address indicates how many read/write accesses were performed. This block provides interrupts to identify the end of a transfer, which can be found in SCB8_INTR_I2C_EC and SCB8_INTR_SPI_EC register sections. Command-response mode operation is available in Active, Sleep, and Deep Sleep power modes. The command-response mode has two phases of operation:

- Write phase - The write phase begins with a START/RESTART followed by the slave address with read/write bit set to '0' indicating a write. The slave's current write address is set to the slave's base write address. Received data elements are written to the current write address memory location. After each memory write, the current write address is incremented.
- Read phase - The read phase begins with a START/RESTART followed by the slave address with read/write bit set to '1' indicating a read. The slave's current read address is set to the slave's base read address. Transmitted data elements are read from the current address memory location. After each read data element is transferred, the current read address is incremented.

Figure 28-41. I^2C Command-Response Mode



Note: A slave's base addresses are updated by the CPU and not by the master.

28.5.7 Clocking and Oversampling

The SCB I^{2}C supports both internally and externally clocked operation modes. Two bitfields (EC_AM_MODE and EC_OP MODE) in the SCB_CTRL register determine the SCB clock mode. EC_AM_MODE indicates whether I^{2}C address matching is internally (0) or externally (1) clocked. I^{2}C address matching comprises the first part of the I^{2}C protocol. EC_OP_MODE indicates whether the rest of the protocol operation (besides I^{2}C address matching) is internally (0) or externally (1) clocked. The externally clocked mode of operation is supported only in the I^{2}C slave mode.

An internally-clocked operation uses the programmable clock dividers. For I^{2}C, an integer clock divider must be used for both master and slave. For more information on system clocking, see the Clocking System chapter on page 242. The internally-clocked mode does not support the command-response mode.

The SCB_CTRL bitfields EC_AM_MODE and EC_OP_MODE can be configured in the following ways.

- EC_AM_MODE is ‘0’ and EC_OP_MODE is ‘0’: Use this configuration when only Active mode functionality is required.
  - FIFO mode: Supported.
  - EZ mode: Supported.
  - Command-response mode: Not supported. The slave NACKs every slave address.
- EC_AM_MODE is ‘1’ and EC_OP_MODE is ‘0’: Use this configuration when both Active and Deep Sleep functionality are required. This configuration relies on the externally clocked functionality for the I^{2}C address matching and relies on the internally clocked functionality to access the memory buffer. The “hand over” from external to internal functionality relies either on an ACK/NACK or clock stretching scheme. The former may result in termination of the current transfer and relies on a master retry. The latter stretches the current transfer after a matching address is received. This mode requires the master to support either NACK generation (and retry) or clock stretching. When the I^{2}C address is matched, INTR_I2C_EC.WAKE_UP is set to ‘1’. The associated Deep Sleep functionality interrupt brings the system into Active power mode.
  - FIFO mode: See “Deep Sleep to Active Transition” on page 357
  - EZ mode: See “Deep Sleep to Active Transition” on page 358.
  - CMD_RESP mode: Not supported. The slave NACKs every slave address.
- EC_AM_MODE is ‘1’ and EC_OP_MODE is ‘1’. Use this mode when both Active and Deep Sleep functionality are required. When the slave is selected, INTR_I2C_EC.WAKE_UP is set to ‘1’. The associated Deep Sleep functionality interrupt brings the system into Active power mode. When the slave is deselected, INTR_I2C_EC.EZ_STOP and/or INTR_I2C_EC.EZ_WRITE_STOP are set to ‘1’.
  - FIFO mode: Not supported.
  - EZ mode: Supported.
  - CMD_RESP mode: Supported.

An externally-clocked operation uses a clock provided by the serial interface. The externally clocked mode does not support FIFO mode. If EC_OP_MODE is ‘1’, the external interface logic accesses the memory buffer on the external interface clock (I^{2}C SCL). This allows for EZ and CMD_RESP mode functionality in Active and Deep Sleep power modes.

In Active system power mode, the memory buffer requires arbitration between external interface logic (on I^{2}C SCL) and the CPU interface logic (on system peripheral clock). This arbitration always gives the highest priority to the external interface logic (host accesses). The external interface logic takes one serial interface clock/bit periods for the I^{2}C. During this period, the internal logic is denied service to the memory buffer. The PSoC 6 MCU provides two programmable options to address this “denial of service”:

- If the BLOCK bitfield of SCB_CTRL is ‘1’: An internal logic access to the memory buffer is blocked until the memory buffer is granted and the external interface logic has completed access. For a 100-kHz I^{2}C interface, the maximum blocking period of one serial interface bit period measures 10 µs (approximately 208 clock cycles on a 48 MHz SCB input clock). This option provides normal SCB register functionality, but the blocking time introduces additional internal bus wait states.
- If the BLOCK bitfield of SCB_CTRL is ‘0’: An internal logic access to the memory buffer is not blocked, but fails when it conflicts with an external interface logic access. A read access returns the value 0xFFFF:FFFF and a write access is ignored. This option does not introduce additional internal bus wait states, but an access to the memory buffer may not take effect. In this case, following failures are detected:
  - Read Failure: A read failure is easily detected, as the returned value is 0xFFFF:FFFF. This value is unique as non-failing memory buffer read accesses return an unsigned byte value in the range 0x0000:0000-0x0000:00FF.
  - Write Failure: A write failure is detected by reading back the written memory buffer location, and confirming that the read value is the same as the written value.

For both options, a conflicting internal logic access to the memory buffer sets INTR_TX.BLOCKED field to ‘1’ (for write access-es) and INTR_RX.BLOCKED field to ‘1’ (for read accesses). These fields can be used as either status fields

or as interrupt cause fields (when their associated mask fields are enabled).

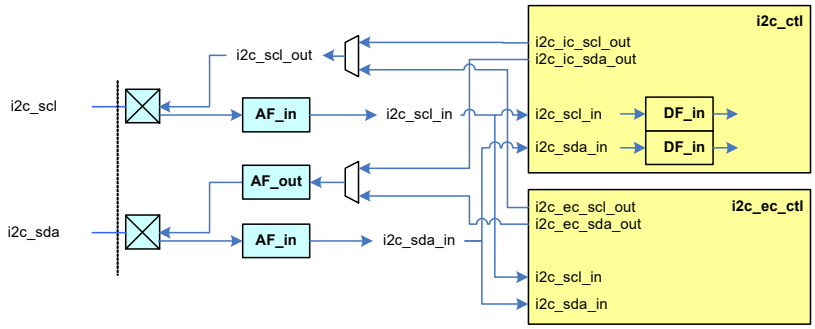
If a series of read or write accesses is performed and CTRL.BLOCKED is '0', a failure is detected by comparing the logical OR of all read values to 0xFFFF:FFFF and checking the INTR_TX.BLOCKED and INTR_RX.BLOCKED fields to determine whether a failure occurred for a (series of) write or read operation(s).

28.5.7.1 Glitch Filtering

The PSoC 6 MCU SCB I^{2}C has analog and digital glitch filters. Analog glitch filters are applied on the i2c_scl_in and i2c_sda_in input signals (AF_in) to filter glitches of up to 50 ns. An analog glitch filter is also applied on the i2c_sda_out output signal (AF_out). Analog glitch filters are enabled and disabled in the SCB.I2C_CFG register. Do not change the _TRIM bitfields; only change the _SEL bitfields in this register.

Digital glitch filters are applied on the i2c_scl_in and i2c_sda_in input signals (DF_in). The digital glitch filter is enabled in the SCB.RX_CTRL register via the MEDIAN bitfield.

Figure 28-42. I^{2}C Glitch Filtering Connection



The following table lists the useful combinations of glitch filters.

Table 28-16. Glitch Filter Combinations

AF_in	AF_out	DF_in	Comments
0	0	1	Used when operating in internally-clocked mode and in master in fast-mode plus (1-MHz speed mode)
1	0	0	Used when operating in internally-clocked mode (EC_OP_MODE is '0')
1	1	0	Used when operating in externally-clocked mode (EC_OP_MODE is '1'). Only slave mode.

When operating in EC_OP_MODE = 1, the 100-kHz, 400-kHz, and 1000-kHz modes require the following settings for AF_out:

AF_in	AF_out	DF_in	
1	1	0	100-kHz mode: I2C_CFG.SDA_OUT_FILT_SEL = 3 400-kHz mode: I2C_CFG.SDA_OUT_FILT_SEL = 3 1000-kHz mode: I2C_CFG.SDA_OUT_FILT_SEL = 1

28.5.7.2 Oversampling and Bit Rate

Internally-clocked Master

The PSoC 6 MCU implements the I^{2}C clock as an oversampled multiple of the SCB input clock. In master mode, the block determines the I^{2}C frequency. Routing delays on the PCB, on the chip, and the SCB (including analog and digital glitch filters) all contribute to the signal interface timing. In master mode, the block operates off clk_scb and uses programmable oversampling factors for the SCL high (1) and low (0) times. For high and low phase oversampling, see

I2C_CTRL.LOW_PHASE_OVS and I2C_CTRL.HIGH_PHASE_OVS registers. For simple manipulation of the oversampling factor, see the SCB_CTRL.OVS register.

Table 28-17. I²C Frequency and Oversampling Requirements in I²C Master Mode

AF_in	AF_out	DF_in	Mode	Supported Frequency	LOW_PHASE_OVS	HIGH_PHASE_OVS	clk_scb Frequency
0	0	1	100 kHz	[62, 100] kHz	[9, 15]	[9, 15]	[1.98-3.2] MHz
			400 kHz	[264, 400] kHz	[13, 5]	[7, 15]	[8.45-10] MHz
			1000 kHz	[447, 1000] kHz	[8, 15]	[5, 15]	[14.32-25.8] MHz
1	0	0	100 kHz	[48, 100] kHz	[7, 15]	[7, 15]	[1.55-3.2] MHz
			400 kHz	[244, 400] kHz	[12, 15]	[7, 15]	[7.82-10] MHz
			1000 kHz	Not supported			

Table 28-17 assumes worst-case conditions on the I²C bus. The following equations can be used to determine the settings for your own system. This will involve measuring the rise and fall times on SCL and SDA lines in your system.

$t_{CLK\_SCB(Min)} = (t_{LOW} + t_F)/LOW\_PHASE\_OVS$

If clk_scb is any faster than this, the t_{LOW} of the I²C specification will be violated. t_F needs to be measured in your system.

$t_{CLK\_SCB(Max)} = (t_{VD} – t_{RF} – 100\ ns)/3$ (When analog filter is enabled and digital disabled)

$t_{CLK\_SCB(Max)} = (t_{VD} – t_{RF})/4$ (When analog filter is disabled and analog filter is enabled)

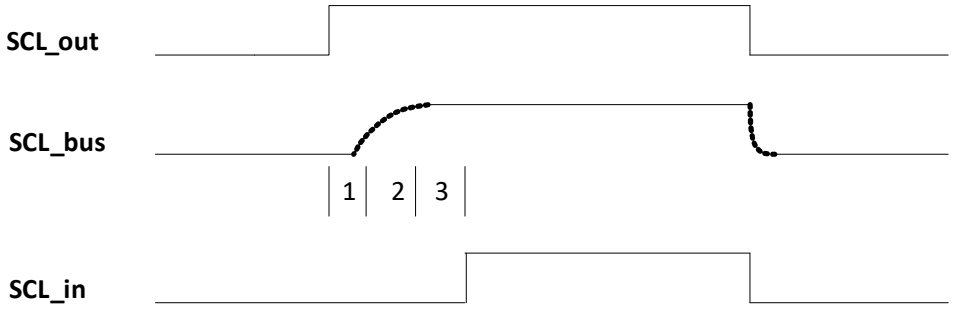
t_{RF} is the maximum of either the rise or fall time. If clk_scb is slower than this frequency, t_{VD} will be violated.

I²C Master Clock Synchronization

The HIGH_PHASE_OVS counter does not start counting until the SCB detects that the SCL line is high. This is not the same as when the SCB sets the SCL high. The differences are explained by three delays:

1. Delay from SCB to I/O pin
2. I²C bus t_R
3. Input delay (filters and synchronization)

Figure 28-43. I²C SCL Turnaround Path



If the above three delays combined are greater than one clk_scb cycle, then the high phase of the SCL will be extended. This may cause the actual data rate on the I²C bus to be slower than expected. This can be avoided by:

- Decreasing the pull-up resistor, or decreasing the bus capacitance to reduce t_R.
- Reducing the I2C_CTRL.HIGH_PHASE_OVS value.

Internally-clocked Slave

In slave mode, the I^2C frequency is determined by the incoming I^2C SCL signal. To ensure proper operation, clk_scb must be significantly higher than the I^2C bus frequency. Unlike master mode, this mode does not use programmable oversampling factors.

Table 28-18. SCB Input Clock Requirements in I^2C Slave Mode

AF_in	AF_out	DF_in	Mode	clk_scb Frequency Range
0	0	1	100 kHz	[1.98-12.8] MHz
			400 kHz	[8.45-17.14] MHz
			1000 kHz	[14.32-44.77] MHz
1	0	0	100 kHz	[1.55-12.8] MHz
			400 kHz	[7.82-15.38] MHz
			1000 kHz	[15.84-89.0] MHz

$t_{CLK\_SCB(Max)} = (t_{VD} – t_{RF} – 100\ ns) / 3$ (When analog filter is enabled and digital disabled)

$t_{CLK\_SCB(Max)} = (t_{VD} – t_{RF}) / 4$ (When analog filter is disabled and analog filter is enabled)

t_{RF} is the maximum of either the rise or fall time. If clk_scb is slower than this frequency, t_{VD} will be violated.

The minimum period of clk_scb is determined by one of the following equations:

$t_{CLK\_SCB(MIN)} = (t_{SU;DAT(min)} + t_{RF}) /16$

or

$t_{CLK\_SCB(Min)} = (0.6 * t_F – 50\ ns)/2$ (When analog filter is enabled and digital disabled)

$t_{CLK\_SCB(Min)} = (0.6 * t_F)/3$ (When analog filter is disabled and digital enabled)

The result that yields the largest period from the two sets of equations above should be used to set the minimum period of clk_scb.

Master-Slave

In this mode, when the SCB is acting as a master device, the block determines the I^2C frequency. When the SCB is acting as a slave device, the block does not determine the I^2C frequency. Instead, the incoming I^2C SCL signal does.

To guarantee operation in both master and slave modes, choose clock frequencies that work for both master and slave using the tables above.

28.5.8 Enabling and Initializing the I^2C

The following section describes the method to configure the I^2C block for standard (non-EZ) mode and EZI2C mode.

28.5.8.1 Configuring for I^2C FIFO Mode

The I^2C interface must be programmed in the following order.

1. Program protocol specific information using the SCB_I2C_CTRL register. This includes selecting master - slave functionality.
2. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers.
3. Set the SCB_CTRL.BYTE_MODE to ‘1’ to enable the byte mode.
4. Program the SCB_CTRL register to enable the I^2C block and select the I^2C mode. For a complete description of the I^2C registers, see the registers TRM.

28.5.8.2 Configuring for EZ and CMD_RESP Modes

To configure the I^2C block for EZ and CMD_RESP modes, set the following I^2C register bits

1a. Select the EZI2C mode by writing '1' to the EZ_MODE bit (bit 10) of the SCB_CTRL register.

1b. Select CMD_RESP mode by writing a 1 to the CMD_RESP bit (bit 12) of the SCB_CTRL register.

2. Set the S_READY_ADDR_ACK (bit 12) and S_READY_DATA_ACK (bit 13) bits of the SCB_I2C_CTRL register.

Note: For all modes clk_scb must also be configured. For information on configuring a peripheral clock and connecting it to the SCB consult the Clocking System chapter on page 242.

The GPIO must also be connected to the SCB; see the following section for more details.

28.5.9 I/O Pad Connections

Figure 28-44. I²C I/O Pad Connections

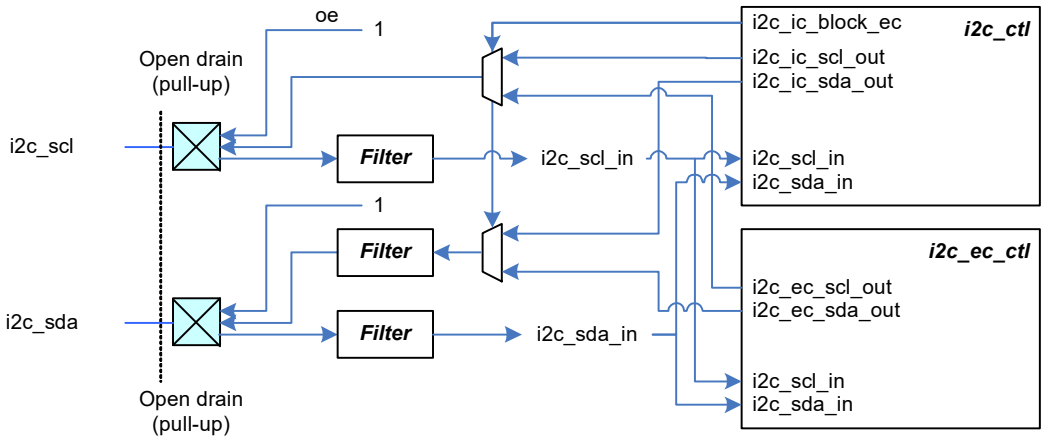


Table 28-19. I²C I/O Pad Descriptions

I/O Pads	Drive Mode	On-chip I/O Signals	Usage
i2c_scl	Open drain with external pull-up	i2c_scl_in	Receive a clock
		i2c_scl_out	Transmit a clock
i2c_sda	Open drain with external pull-up	i2c_sda_in	Receive data
		i2c_sda_out	Transmit data

When configuring the I²C SDA/SCL lines, the following sequence must be followed. If this sequence is not followed, the I²C lines may initially have overshoot and undershoot.

1. Set SCB_CTRL_MODE to '0'.
2. Configure HSIOM for SCL and SDA to connect to the SCB.
3. Set TX_CTRL.OPEN_DRAIN to '1'.
4. Configure I²C pins for high-impedance drive mode.
5. Configure SCB for I²C
6. Enable SCB
7. Configure I²C pins for Open Drain Drives Low.

28.5.10 I²C Registers

The I²C interface is controlled by reading and writing a set of configuration, control, and status registers, as listed in Table 28-20.

Table 28-20. I²C Registers

Register	Function
SCB_CTRL	Enables the SCB block and selects the type of serial interface (SPI, UART, I²C). Also used to select internally and externally clocked operation and EZ and non-EZ modes of operation.
SCB_I2C_CTRL	Selects the mode (master, slave) and sends an ACK or NACK signal based on receiver FIFO status.
SCB_I2C_STATUS	Indicates bus busy status detection, read/write transfer status of the slave/master, and stores the EZ slave address.
SCB_I2C_M_CMD	Enables the master to generate START, STOP, and ACK/NACK signals.
SCB_I2C_S_CMD	Enables the slave to generate ACK/NACK signals.
SCB_STATUS	Indicates whether the externally clocked logic is using the EZ memory. This bit can be used by software to determine whether it is safe to issue a software access to the EZ memory.
SCB_I2C_CFG	Configures filters, which remove glitches from the SDA and SCL lines.
SCB_TX_CTRL	Specifies the data frame width; also used to specify whether MSb or LSb is the first bit in transmission.
SCB_TX_FIFO_CTRL	Specifies the trigger level, clearing of the transmitter FIFO and shift registers, and FREEZE operation of the transmitter FIFO.
SCB_TX_FIFO_STATUS	Indicates the number of bytes stored in the transmitter FIFO, the location from which a data frame is read by the hardware (read pointer), the location from which a new data frame is written (write pointer), and decides if the transmitter FIFO holds the valid data.
SCB_TX_FIFO_WR	Holds the data frame written into the transmitter FIFO. Behavior is similar to that of a PUSH operation.
SCB_RX_CTRL	Performs the same function as that of the SCB_TX_CTRL register, but for the receiver. Also decides whether a median filter is to be used on the input interface lines.
SCB_RX_FIFO_CTRL	Performs the same function as that of the SCB_TX_FIFO_CTRL register, but for the receiver.
SCB_RX_FIFO_STATUS	Performs the same function as that of the SCB_TX_FIFO_STATUS register, but for the receiver.
SCB_RX_FIFO_RD	Holds the data read from the receiver FIFO. Reading a data frame removes the data frame from the FIFO; behavior is similar to that of a POP operation. This register has a side effect when read by software: a data frame is removed from the FIFO.
SCB_RX_FIFO_RD_SILENT	Holds the data read from the receiver FIFO. Reading a data frame does not remove the data frame from the FIFO; behavior is similar to that of a PEEK operation.
SCB_RX_MATCH	Stores slave device address and is also used as slave device address MASK.
SCB_EZ_DATA	Holds the data in an EZ memory location.

Note: Detailed descriptions of the I²C register bits are available in the registers TRM.

28.6 SCB Interrupts

SCB supports interrupt generation on various events. The interrupts generated by the SCB block vary depending on the mode of operation.

Table 28-21. SCB Interrupts

Interrupt	Functionality	Active/Deep Sleep	Registers
interrupt_master	I^2C master and SPI master functionality	Active	INTR_M, INTR_M_SET, INTR_M_MASK, INTR_M_MASKED
interrupt_slave	I^2C slave and SPI slave functionality	Active	INTR_S, INTR_S_SET, INTR_S_MASK, INTR_S_MASKED
interrupt_tx	UART transmitter and TX FIFO functionality	Active	INTR_TX, INTR_TX_SET, INTR_TX_MASK, INTR_TX_MASKED
interrupt_rx	UART receiver and RX FIFO functionality	Active	INTR_RX, INTR_RX_SET, INTR_RX_MASK, INTR_RX_MASKED
interrupt_i2c_ec	Externally clocked I^2C slave functionality	Deep Sleep	INTR_I2C_EC, INTR_I2C_EC_MASK, INTR_I2C_EC_MASKED
interrupt_spi_ec	Externally clocked SPI slave functionality	Deep Sleep	INTR_ISPI_EC, INTR_SPI_EC_MASK, INTR_SPI_EC_MASKED

Note: To avoid being triggered by events from previous transactions, whenever the firmware enables an interrupt mask register bit, it should clear the interrupt request register in advance.

Note: If the DMA is used to read data out of RX FIFO, the NOT_EMPTY interrupt may never trigger. This can occur when clk_peri (clocking DMA) is running much faster than the clock to the SCB. As a workaround to this issue, set the RX_FIFO_CTRL.TRIGGER_LEVEL to ‘1’ (not 0); this will allow the interrupt to fire.

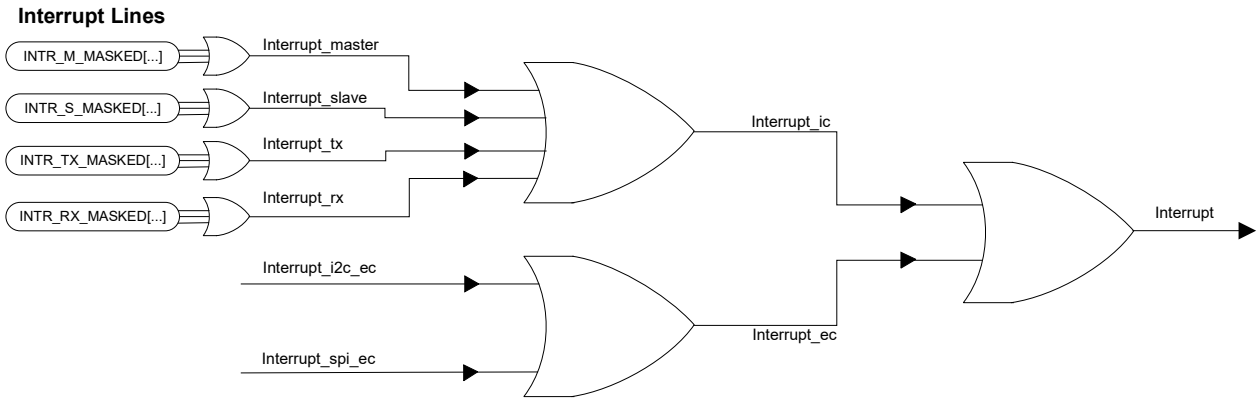
The following register definitions correspond to the SCB interrupts:

- **INTR_M:** This register provides the instantaneous status of the interrupt sources. A write of ‘1’ to a bit will clear the interrupt.
- **INTR_M_SET:** A write of ‘1’ into this register will set the interrupt.
- **INTR_M_MASK:** The bit in this register masks the interrupt sources. Only the interrupt sources with their masks enabled can trigger the interrupt.
- **INTR_M_MASKED:** This register provides the instantaneous value of the interrupts after they are masked. It provides logical and corresponding request and mask bits. This is used to understand which interrupt triggered the event.

Note: While registers corresponding to INTR_M are used here, these definitions can be used for INTR_S, INTR_TX, INTR_RX, INTR_I2C_EC, and INTR_SPI_EC.

Figure 28-45 shows the physical interrupt lines. All the interrupts are OR'd together to make one interrupt source that is the OR of all six individual interrupts. All the externally-clocked interrupts make one interrupt line called interrupt_ec, which is the OR'd signal of interrupt_i2C_ec and interrupt_spi_ec. All the internally-clocked interrupts make one interrupt line called interrupt_ic, which is the OR'd signal of interrupt_master, interrupt_slave, interrupt_tx, and interrupt_rx. The Active functionality interrupts are generated synchronously to clk_peri while the Deep Sleep functionality interrupts are generated asynchronously to clk_peri.

Figure 28-45. Interrupt Lines



28.6.1 SPI Interrupts

The SPI interrupts can be classified as master interrupts, slave interrupts, TX interrupts, RX interrupts, and externally clocked (EC) mode interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCB_INTR_CAUSE) that can be used to determine interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bitfield. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the registers TRM. The SPI supports interrupts on the following events:

- SPI Master Interrupts
  - SPI master transfer done – All data from the TX FIFO are sent. This interrupt source triggers later than TX_FIFO_EMPTY by the amount of time it takes to transmit a single data element. TX_FIFO_EMPTY triggers when the last data element from the TX FIFO goes to the shifter register. However, SPI Done triggers after this data element is transmitted. This means SPI Done will be asserted one SCLK clock cycle earlier than the completion of data element reception.
- SPI Slave Interrupts
  - SPI Bus Error – Slave deselected at an unexpected time in the SPI transfer. The firmware may decide to clear the TX and RX FIFOs for this error.
  - SPI slave deselected after any EZSPI transfer occurred.
  - SPI slave deselected after a write EZSPI transfer occurred.
- SPI TX
  - TX FIFO has less entries than the value specified by TRIGGER_LEVEL in SCB_TX_FIFO_CTRL.
  - TX FIFO not full – At least one data element can be written into the TX FIFO.
  - TX FIFO empty – The TX FIFO is empty.
  - TX FIFO overflow – Firmware attempts to write to a full TX FIFO.
  - TX FIFO underflow – Hardware attempts to read from an empty TX FIFO. This happens when the SCB is ready to transfer data and EMPTY is '1'.
  - TX FIFO trigger – Less entries in the TX FIFO than the value specified by TX_FIFO_CTRL.TRIGGER_LEVEL.
- SPI RX
  - RX FIFO has more entries than the value specified by TRIGGER_LEVEL in SCB_RX_FIFO_CTRL.
  - RX FIFO full - RX FIFO is full.
  - RX FIFO not empty - RX FIFO is not empty. At least one data element is available in the RX FIFO to be read.
  - RX FIFO overflow - Hardware attempt to write to a full RX FIFO.
  - RX FIFO underflow - Firmware attempts to read from and empty RX FIFO.

- RX FIFO trigger - More entries in the RX FIFO than the value specified by RX_FIFO_CTRL.TRIGGER_LEVEL.

- SPI Externally Clocked
  - Wake up request on slave select – Active on incoming slave request (with address match). Only set when EC_AM is '1'.
  - SPI STOP detection at the end of each transfer – Activated at the end of every transfer (I2C STOP). Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is '1'.
  - SPI STOP detection at the end of a write transfer – Activated at the end of a write transfer (I2C STOP). This event is an indication that a buffer memory location has been written to. For EZ mode, a transfer that only writes the base address does not activate this event. Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is '1'.
  - SPI STOP detection at the end of a read transfer – Activated at the end of a read transfer (I2C STOP). This event is an indication that a buffer memory location has been read from. Only set for a slave request with an address match, in EZ and CMD_RESP modes when EC_OP is '1'.

Figure 28-46 and Figure 28-47 show how each of the interrupts are triggered. Figure 28-46 shows the TX buffer and the corresponding interrupts while Figure 28-47 shows all the corresponding interrupts for the RX buffer. The FIFO has 256 split into 128 bytes for TX and 128 bytes for RX instead of the 8 bytes shown in the figures. For more information on how to implement and clear interrupts, see the SPI (SCB_SPI_PDL) datasheet and the PDL.

Figure 28-46. TX Interrupt Source Operation

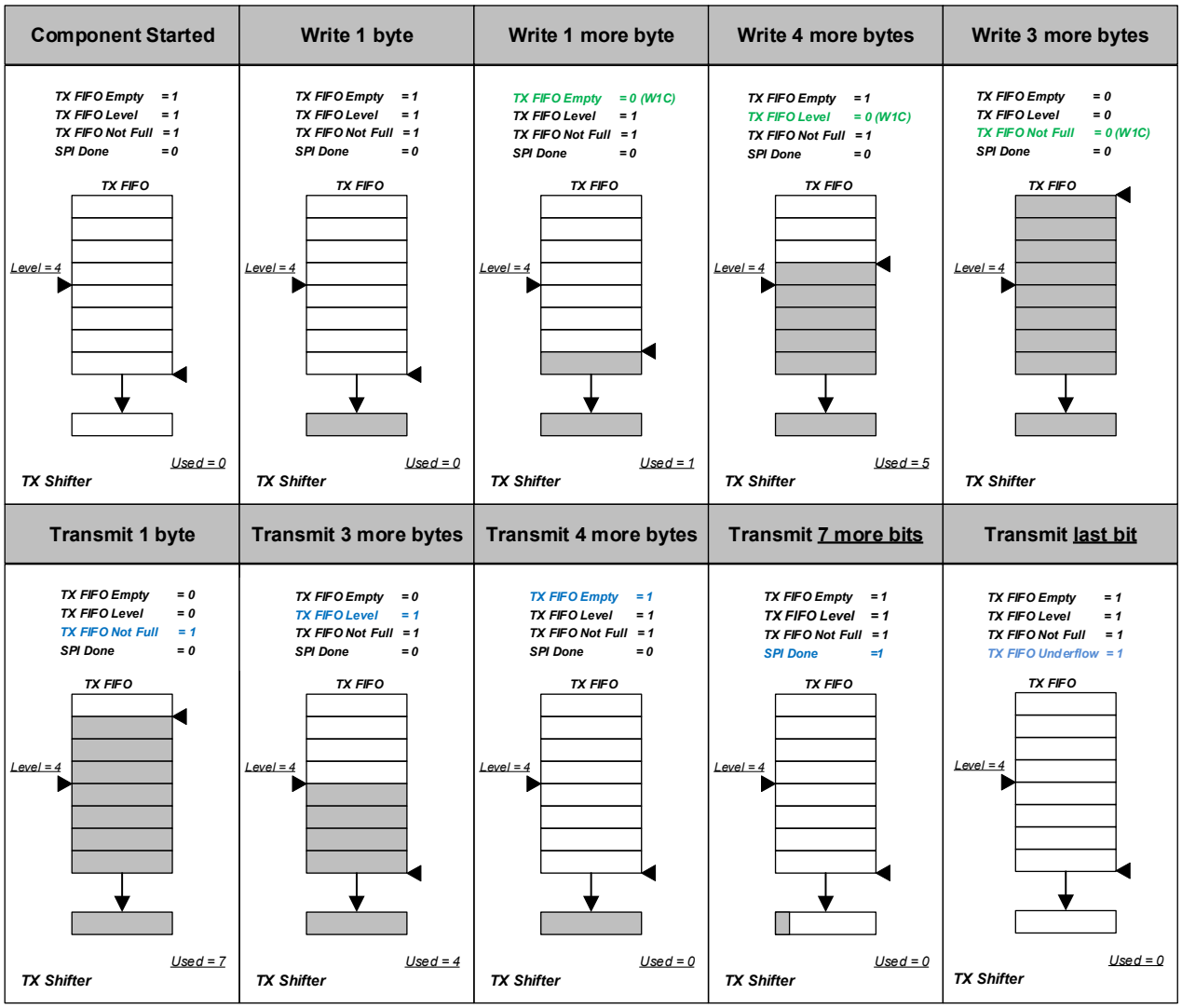
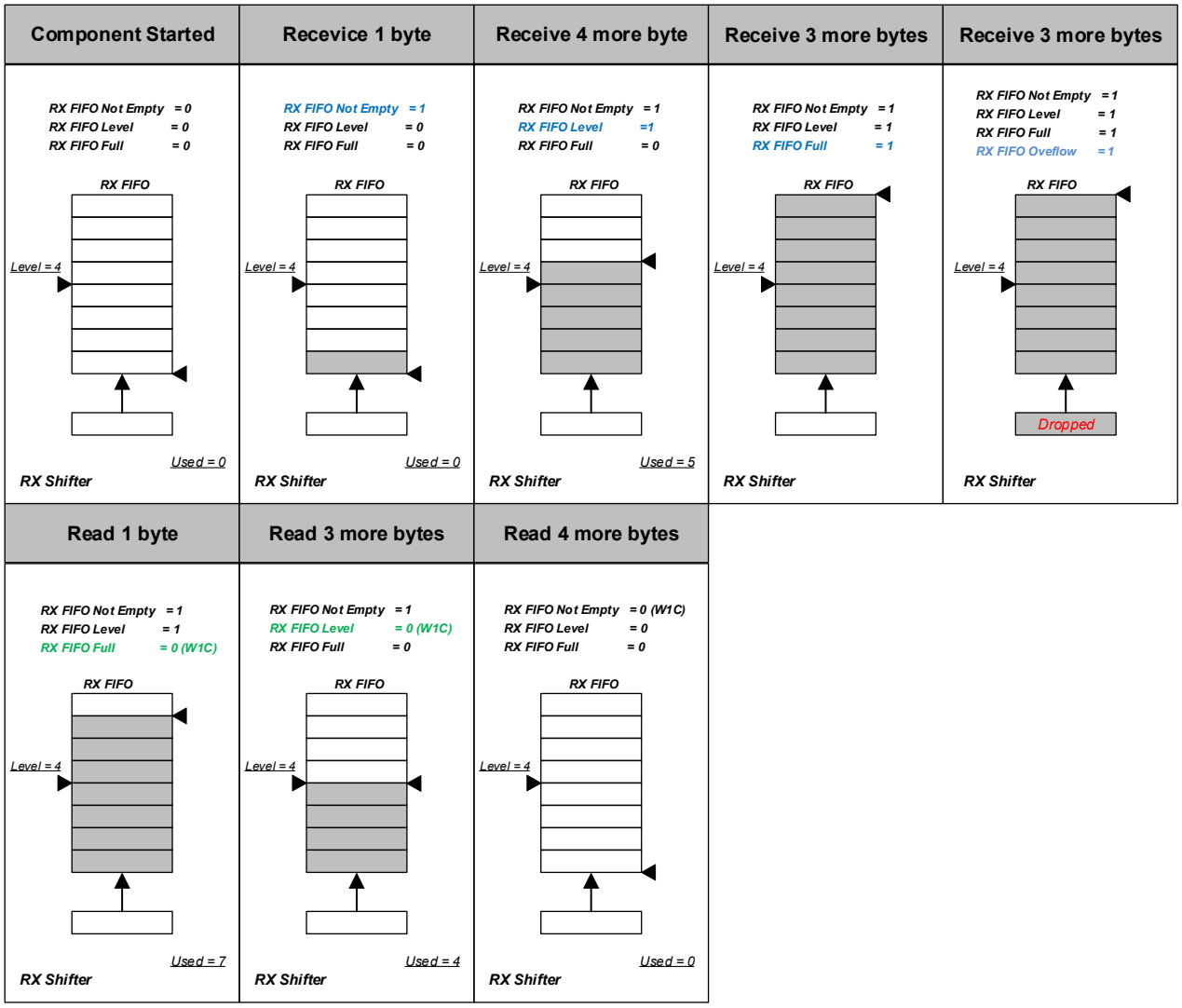


Figure 28-47. RX Interrupt Source Operation



28.6.2 UART Interrupts

The UART interrupts can be classified as TX interrupts and RX interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCB_ INTR_CAUSE) that can be used to determine interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bitfield. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the registers TRM. The UART block generates interrupts on the following events:

- UART TX
  - TX FIFO has fewer entries than the value specified by TRIGGER_LEVEL in SCB_TX_FIFO_CTRL.
  - TX FIFO not full – TX FIFO is not full. At least one data element can be written into the TX FIFO.
  - TX FIFO empty – The TX FIFO is empty.
  - TX FIFO overflow – Firmware attempts to write to a full TX FIFO.

- TX FIFO underflow – Hardware attempts to read from an empty TX FIFO. This happens when the SCB is ready to transfer data and EMPTY is ‘1’.
- TX NACK – UART transmitter receives a negative acknowledgment in SmartCard mode.
- TX done – This happens when the UART completes transferring all data in the TX FIFO and the last stop field is transmitted (both TX FIFO and transmit shifter register are empty).
- TX lost arbitration – The value driven on the TX line is not the same as the value observed on the RX line. This condition event is useful when transmitter and receiver share a TX/RX line. This is the case in LIN or SmartCard modes.

- UART RX
  - RX FIFO has more entries than the value specified by TRIGGER_LEVEL in SCB_RX_FIFO_CTRL.
  - RX FIFO full – RX FIFO is full. Note that received data frames are lost when the RX FIFO is full.
  - RX FIFO not empty – RX FIFO is not empty.
  - RX FIFO overflow – Hardware attempts to write to a full RX FIFO.
  - RX FIFO underflow – Firmware attempts to read from an empty RX FIFO.
  - Frame error in received data frame – UART frame error in received data frame. This can be either a start of stop bit error:
    Start bit error: After the beginning of a start bit period is detected (RX line changes from 1 to 0), the middle of the start bit period is sampled erroneously (RX line is ‘1’). **Note:** A start bit error is detected before a data frame is received.
    Stop bit error: The RX line is sampled as ‘0’, but a ‘1’ was expected. A stop bit error may result in failure to receive successive data frames. **Note:** A stop bit error is detected after a data frame is received.
  - Parity error in received data frame – If UART_RX_CTL.DROP_ON_PARITY_ERROR is ‘1’, the received frame is dropped. If UART_RX_CTL.DROP_ON_PARITY_ERROR is ‘0’, the received frame is sent to the RX FIFO. In SmartCard sub mode, negatively acknowledged data frames generate a parity error. Note that firmware can only identify the erroneous data frame in the RX FIFO if it is fast enough to read the data frame before the hardware writes a next data frame into the RX FIFO.
  - LIN baud rate detection is completed – The receiver software uses the UART_RX_STATUS.BR_COUNTER value to set the clk_scb to guarantee successful receipt of the first LIN data frame (Protected Identifier Field) after the synchronization byte.
  - LIN break detection is successful – The line is ‘0’ for UART_RX_CTRL.BREAK_WIDTH + 1 bit period. Can occur at any time to address unanticipated break fields; that is, “break-in-data” is supported. This feature is supported for the UART standard and LIN submodes. For the UART standard submodes, ongoing receipt of data frames is not affected; firmware is expected to take proper action. For the LIN submode, possible ongoing receipt of a data frame is stopped and the (partially) received data frame is dropped and baud rate detection is started. Set to ‘1’, when event is detected. Write with '1' to clear bit.

Figure 28-48 and Figure 28-49 show how each of the interrupts are triggered. Figure 28-48 shows the TX buffer and the corresponding interrupts while Figure 28-49 shows all the corresponding interrupts for the RX buffer. The FIFO has 256 split into 128 bytes for TX and 128 bytes for RX instead of the 8 bytes shown in the figures. For more information on how to implement and clear interrupts see the UART (SCB_UART_PDL) datasheet and the PDL.

Figure 28-48. TX Interrupt Source Operation

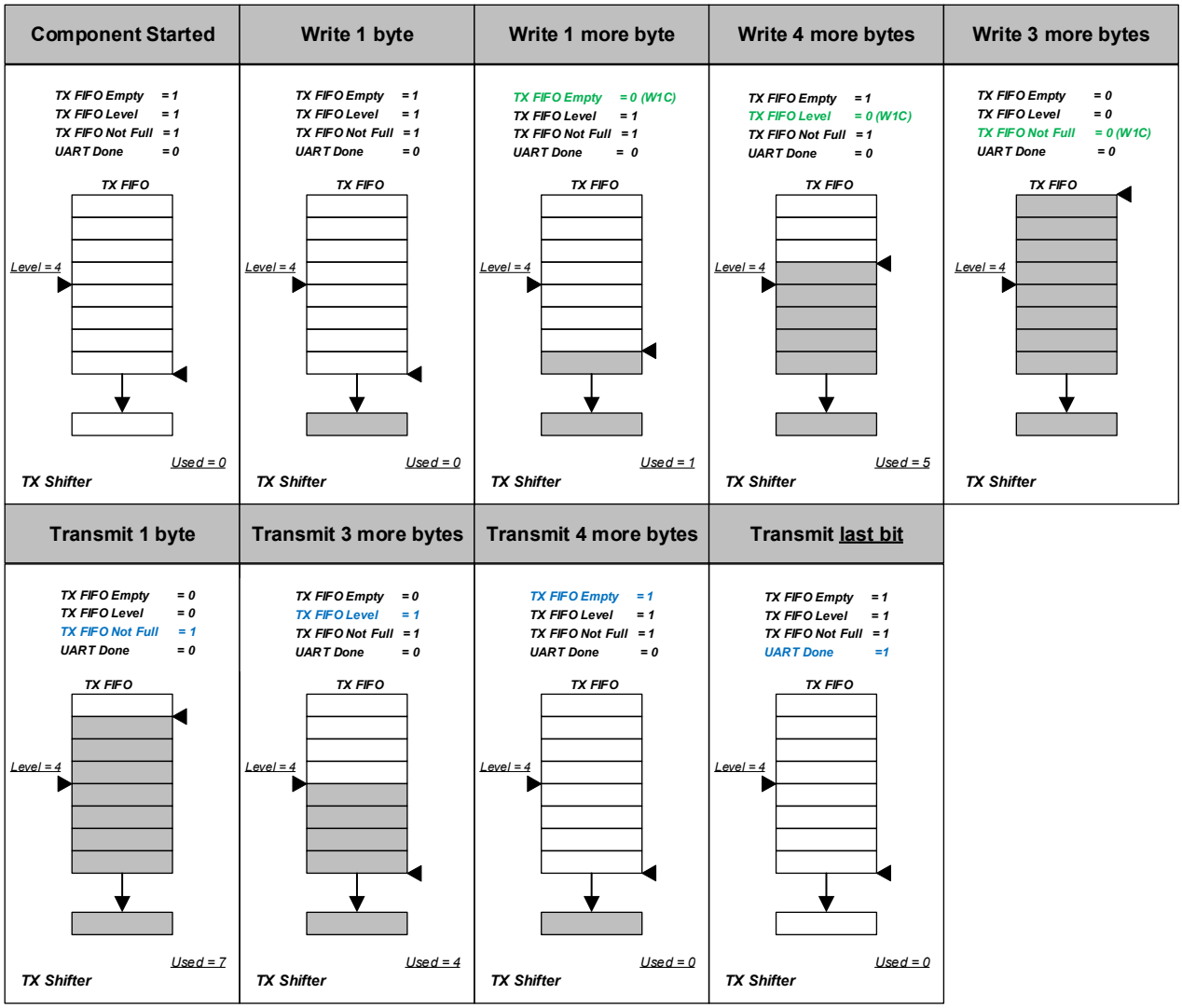
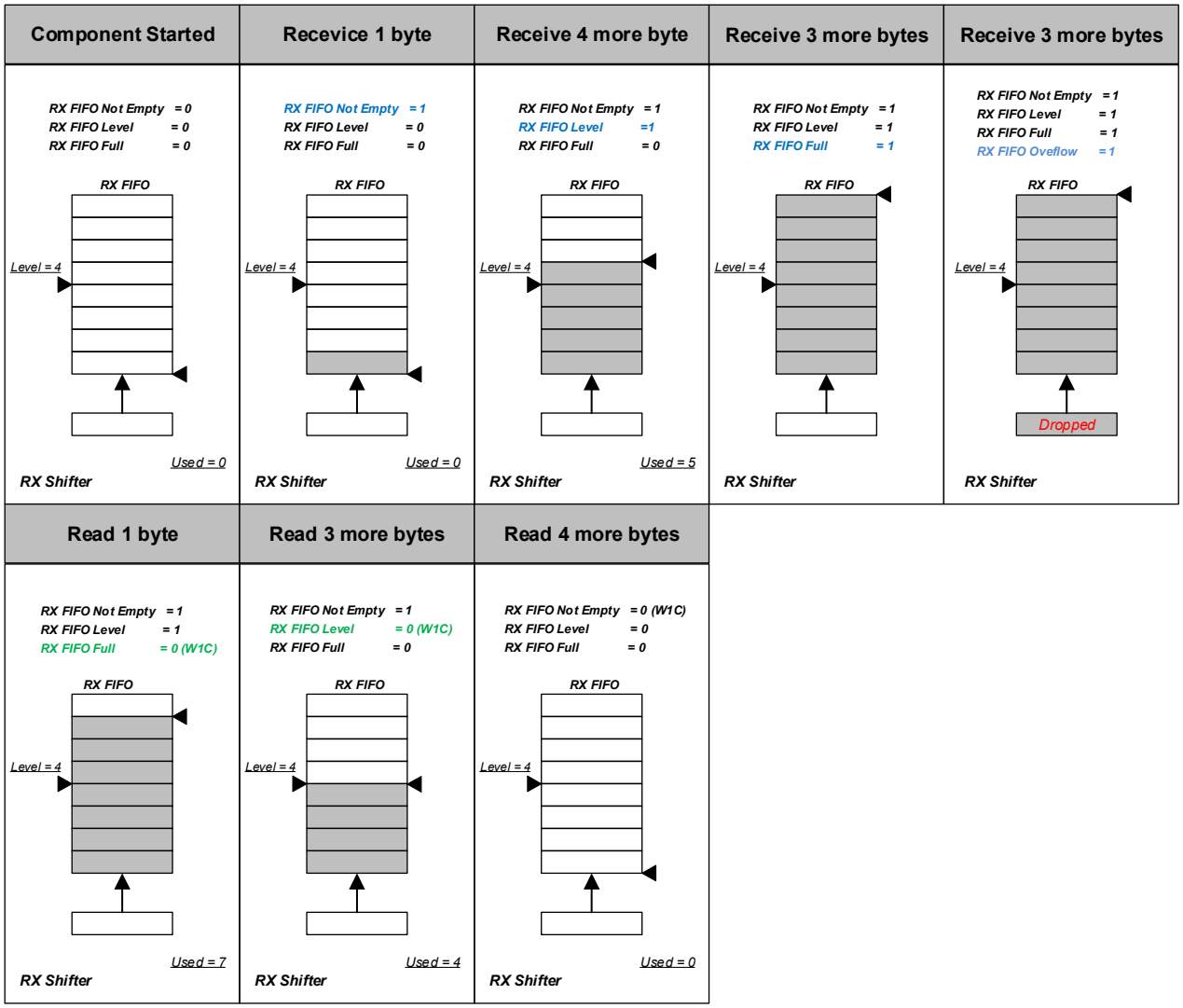


Figure 28-49. RX Interrupt Source Operation



28.6.3 I²C Interrupts

I²C interrupts can be classified as master interrupts, slave interrupts, TX interrupts, RX interrupts, and externally clocked (EC) mode interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCB_INTR_CAUSE) that can be used to determine interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bitfield. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the registers TRM. The I²C block generates interrupts for the following conditions.

- I²C Master
  - I²C master lost arbitration – The value driven by the master on the SDA line is not the same as the value observed on the SDA line.
  - I²C master received NACK – When the master receives a NACK (typically after the master transmitted the slave address or TX data).
  - I²C master received ACK – When the master receives an ACK (typically after the master transmitted the slave address or TX data).
  - I²C master sent STOP – When the master has transmitted a STOP.
  - I²C bus error – Unexpected stop/start condition is detected.
- I²C Slave
  - I²C slave lost arbitration – The value driven on the SDA line is not the same as the value observed on the SDA line (while the SCL line is '1'). This should not occur; it represents erroneous I²C bus behavior. In case of lost arbitration, the I²C slave state machine aborts the ongoing transfer. Software may decide to clear the TX and RX FIFOs in case of this error.
  - I²C slave received NACK – When the slave receives a NACK (typically after the slave transmitted TX data).
  - I²C slave received ACK – When the slave receives an ACK (typically after the slave transmitted TX data).
  - I²C slave received STOP – I²C STOP event for I²C (read or write) transfer intended for this slave (address matching is performed). When STOP or REPEATED START event is detected. The REPEATED START event is included in this interrupt cause such that the I²C transfers separated by a REPEATED START can be distinguished and potentially treated separately by the firmware. Note that the second I²C transfer (after a REPEATED START) may be to a different slave address.
    The event is detected on any I²C transfer intended for this slave. Note that an I²C address intended for the slave (address matches) will result in an I2C_STOP event independent of whether the I²C address is ACK'd or NACK'd.
  - I²C slave received START – When START or REPEATED START event is detected. In the case of an externally-clocked address matching (CTRL.EC_AM_MODE is '1') and clock stretching is performed (until the internally-clocked logic takes over) (I2C_CTRL.S_NOT_READY_ADDR_NACK is '0'), this field is not set. Firmware should use INTR_S_EC.WAKE_UP, INTR_S.I2C_ADDR_MATCH, and INTR_S.I2C_GENERAL.
  - I²C slave address matched – I²C slave matching address received. If CTRL.ADDR_ACCEPT, the received address (including the R/W bit) is available in the RX FIFO. In the case of externally-clocked address matching (CTRL.EC_AM_MODE is '1') and internally-clocked operation (CTRL.EC_OP_MODE is '0'), this field is set when the event is detected.
  - I²C bus error – Unexpected STOP/START condition is detected
- I²C TX
  - TX trigger – TX FIFO has fewer entries than the value specified by TRIGGER_LEVEL in SCB_TX_FIFO_CTRL.
  - TX FIFO not full – At least one data element can be written into the TX FIFO.
  - TX FIFO empty – The TX FIFO is empty.
  - TX FIFO overflow – Firmware attempts to write to a full TX FIFO.
  - TX FIFO underflow – Hardware attempts to read from an empty TX FIFO.
- I²C RX
  - RX FIFO has more entries than the value specified by TRIGGER_LEVEL in SCB_RX_FIFO_CTRL.

- RX FIFO is full – The RX FIFO is full.
- RX FIFO is not empty – At least one data element is available in the RX FIFO to be read.
- RX FIFO overflow – Hardware attempts to write to a full RX FIFO.
- RX FIFO underflow – Firmware attempts to read from an empty RX FIFO.

- I^2C Externally Clocked
  - Wake up request on address match – Active on incoming slave request (with address match). Only set when EC_AM is ‘1’.
  - I^2C STOP detection at the end of each transfer – Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is ‘1’.
  - I^2C STOP detection at the end of a write transfer – Activated at the end of a write transfer (I^2C STOP). This event is an indication that a buffer memory location has been written to. For EZ mode, a transfer that only writes the base address does not activate this event. Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is ‘1’.
  - I^2C STOP detection at the end of a read transfer – Activated at the end of a read transfer (I^2C STOP). This event is an indication that a buffer memory location has been read from. Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is ‘1’.

For more information on how to implement and clear interrupts see the I^2C (SCB_I2C_PDL) datasheet and the PDL.

29. Serial Memory Interface (SMIF)

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The SMIF block implements a single-SPI, dual-SPI, quad-SPI, or octal-SPI communication to interface with external memory chips. The SMIF block's primary use case is to set up the external memory and have it mapped to the PSoC 6 MCU memory space using the hardware. This mode of operation, called the XIP mode, allows the bus masters in the PSoC 6 MCU to directly interact with the SMIF for memory access to an external memory location.

A graphical interface is provided with the ModusToolbox for configuring the QSPI (SMIF) block. For more information, see the ModusToolbox QSPI Configurator Guide.

29.1 Features

The Serial Memory Interface (SMIF) block provides a master interface to serial memory devices that supports the following functionality.

- Interfacing up to four memory devices (slaves) at a time
- SPI protocol
  - SPI mode 0: clock polarity (CPOL) and clock phase (CPHA) are both '0'
  - Support for single, dual, quad, and octal SPI protocols
  - Support for dual-quad SPI mode: the use of two quad SPI memory devices to increase data bandwidth for SPI read and write transfers
  - Support for configurable MISO sampling time and programmable receiver clock
- Support for device capacities in the range of 64 KB to 128 MB
- eXecute In Place (XIP) enables mapping the external memory into an internal memory address
- Command mode enables using the SMIF block as a simple communication hardware
- Supports a 4-KB read cache in memory mapped (XIP) mode
- Supports on-the-fly 128-bit encryption and decryption

29.2 Architecture

Figure 29-1 shows a high-level block diagram of the SMIF hardware in PSoC 6 MCUs. Notice that the block is divided into multiple clock domains. This enables multiple domains to access the SMIF and still enable maintaining an asynchronous clock for the communication interface.

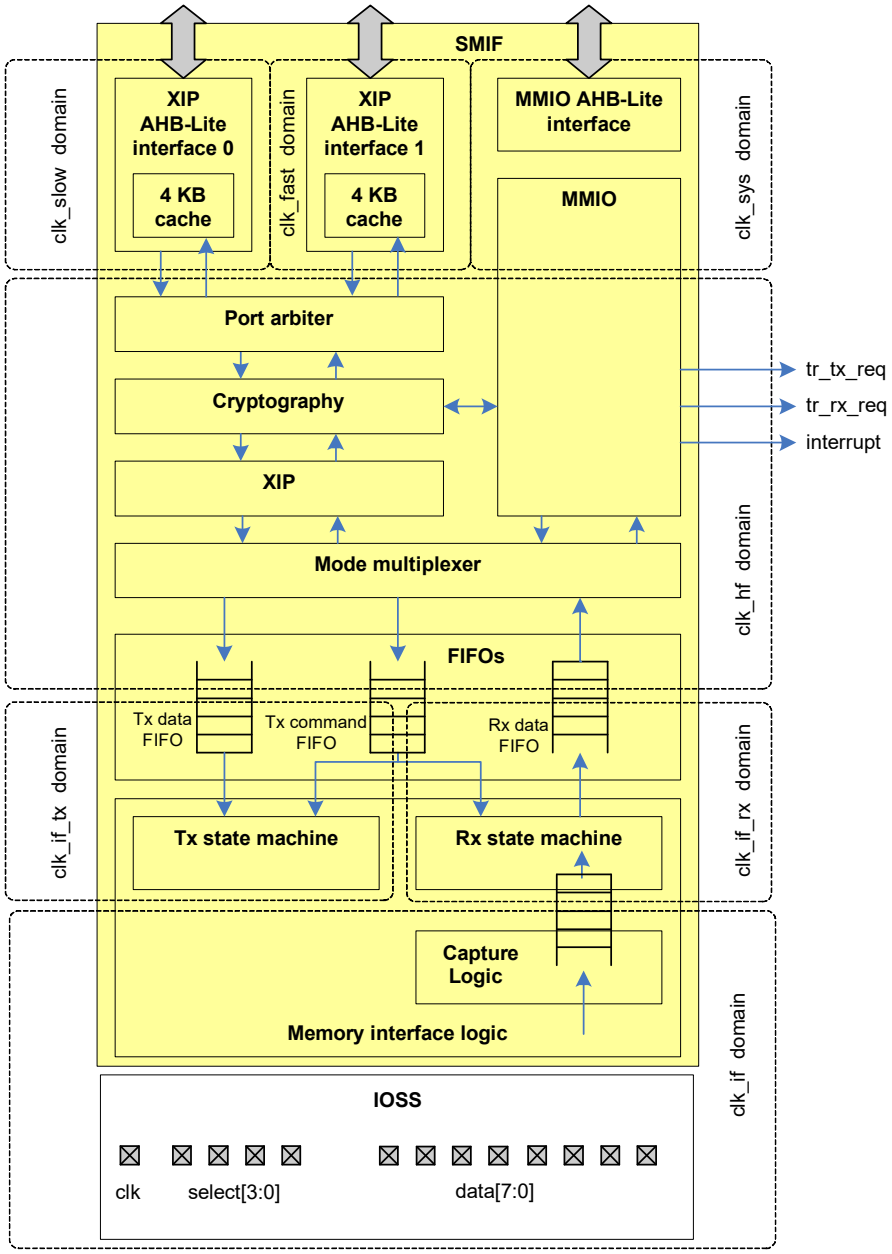
The SMIF block can also generate DMA triggers and interrupt signals. This allows events in the SMIF block to trigger actions in other parts of the system.

The SMIF interface is implemented using eight data lines, four slave select lines, and a clock line.

The access to the SMIF block can be by two modes: Command mode or XIP mode. The Command mode gives access to the SMIF's peripheral registers and the internal FIFOs. This mode is used when the user code is responsible for constructing the command structure for the external memory. Typically, this mode is used when the SMIF writes to an external flash memory. The MMIO interface is also used to configure the SMIF hardware block, including configuring the device registers that set up the XIP operation of the SMIF block.

The XIP mode of operation maps the external memory space to a range of addresses in the PSoC 6 MCU's address space. Refer to the registers TRM for details. When this address range is accessed, the hardware automatically generates the commands required to initiate the associated transfer from the external memory. The typical use case for the XIP mode is to execute code placed in external memory. Thus executing code from external memory is seamless.

Figure 29-1. SMIF Hardware Block Diagram



The SMIF block has three AHB-Lite interfaces:

- An AHB-Lite interface to access the SMIF's MMIO registers.
- Two AHB-Lite interfaces to support execute-in-place (XIP).

All interfaces provide access to external memory devices. At any time, either the MMIO AHB-Lite interface or the two XIP AHB-Lite interfaces have access to the memory interface logic and external memory devices. The operation mode is specified by SMIFn_CTL.XIP_MODE. The operation mode should not be modified when the SMIF is busy (STATUS.BUSY is '1').

In the MMIO AHB-Lite interface, access is supported through software writes to transmit (Tx) FIFOs and software reads from receive (Rx) FIFOs. The FIFOs are mapped on SMIF registers. This interface provides the flexibility to implement any memory device transfer. For example, memory device transfers to setup, program, or erase the external memory devices.

In an XIP AHB-Lite interface, access is supported through XIP: AHB-Lite read and write transfers are automatically (by the hardware) translated in memory device read and write transfers. This interface provides efficient implementation of memory device read and write transfers, but does NOT support other types of memory device transfers. To improve XIP performance, the XIP AHB-Lite interface has a 4-KB read cache.

As mentioned, Command mode and XIP mode are mutually exclusive. The operation modes share Tx and Rx FIFOs and memory interface logic. In Command mode, the Tx and Rx FIFOs are accessed through the SMIF registers and under software control. In XIP mode, the Tx and Rx FIFOs are under hardware control. The memory interface logic is controlled through the Tx and Rx FIFOs and is agnostic of the operation mode.

29.2.1 Tx and Rx FIFOs

The SMIF block has two Tx FIFOs and one Rx FIFO. These FIFOs provide an asynchronous clock domain transfer between clk_hf logic and clk_if_tx/clk_if_rx memory interface logic. The memory interface logic is completely controlled through the Tx and Rx FIFOs.

- The Tx command FIFO transmits memory commands to the memory interface logic.
- The Tx data FIFO transmits write data to the memory interface transmit logic.
- The Rx data FIFO receives read data from the memory interface receive logic.

29.2.1.1 Tx Command FIFO

The Tx command FIFO consists of four 20-bit entries. Each entry holds a command. A memory transfer consists of a series of commands. In other words, a command specifies a phase of a memory transfer. Four different types of commands are supported:

- Tx command. A memory transfer must start with a Tx command. The Tx command includes a byte that is to be transmitted over the memory interface. The Tx command specifies the width of the data transfer (single, dual, quad, or octal data transfer). The Tx command specifies whether the command is for the last phase of the memory transfer (explicit “last command” indication). The Tx command specifies which of the four external devices are selected (multiple devices can be selected simultaneously); that is, the device selection as encoded by the Tx command is used for the complete memory transfer. The Tx command asserts the corresponding slave select lines. This is the reason every memory transfer should start with this command.
- TX_COUNT command. The TX_COUNT command specifies the number of bytes to be transmitted from the Tx data FIFO. This command relies on the Tx data FIFO to provide the bytes that are to be transmitted over the memory interface. The TX_COUNT command specifies the width of the data transfer and always constitutes the last phase of the memory transfer (implicit “last command” indication - de-asserts the slave select). Note that the TX_COUNT command does not assert the slave select lines. This must be done by a Tx command preceding it.
- RX_COUNT command. The RX_COUNT command specifies the number of bytes to be received from the Rx data FIFO. This command relies on the Rx data FIFO to accept the bytes that are received over the memory interface. The RX_COUNT command specifies the width of the data transfer and always constitutes the last phase of the memory transfer (implicit “last command” indication - de-asserts the slave select). Note that the RX_COUNT command does not assert the slave select lines. This must be done by a Tx command preceding it.
- DUMMY_COUNT command. The DUMMY_COUNT command specifies a number of dummy cycles. Dummy cycles are used to implement a turn-around (TAR) time in which the memory master changes from a transmitter driving the data lines to a receiver receiving on the same data lines. The DUMMY_COUNT command never constitutes the last phase of the memory transfer (implicit NOT “last command” indication - de-asserts the slave select); that is, it must be followed by another command. Note that the DUMMY COUNT command does not assert the slave select lines. This must be done by a Tx command preceding it.

Together, the four command types can be used to construct any SPI transfer. The Tx command FIFO is used by both the memory interface transmit and receive logic. This ensures lockstep operation. The Tx command is a representation of a queue of commands that are to be processed.

The software will write the sequence of commands into the Tx command FIFO to generate a sequence responsible for the communication with slave device. The software can read the number of used Tx command FIFO entries through the TX_CMD_FIFO_STATUS.USED[2:0] register field.

The software can write to the Tx command FIFO through the MMIO TX_CMD_FIFO_WR register. If software attempts to write to a full Tx command FIFO, the MMIO CTL.BLOCK field specifies the behavior:

- If CTL.BLOCK is ‘0’, an AHB-Lite bus error is generated.
- If CTL.BLOCK is ‘1’, the AHB-Lite write transfer is extended until an entry is available. This increases latency.

29.2.1.2 Tx Data FIFO

The Tx data FIFO consists of eight 8-bit entries. A Tx command FIFO TX_COUNT command specifies the number of bytes to be transmitted; that is, specifies the number of Tx data FIFO entries used. The Tx data FIFO is used by the memory interface transmit logic.

Software can read the number of used Tx data FIFO entries through the TX_DATA_FIFO_STATUS.USED[3:0] register field.

Software can write to the Tx data FIFO through the TX_DATA_FIFO_WR1, TX_DATA_FIFO_WR2, and TX_DATA_FIFO_WR4 registers:

- The TX_DATA_FIFO_WR1 register supports a write of a single byte to the FIFO.
- The TX_DATA_FIFO_WR2 register supports a write of two bytes to the FIFO.
- The TX_DATA_FIFO_WR4 register supports a write of four bytes to the FIFO. If software attempts to write more bytes than available entries in the Tx data FIFO, the MMIO CTL.BLOCK field specifies the behavior:
- If CTL.BLOCK is '0', an AHB-Lite bus error is generated.
- If CTL.BLOCK is '1', the AHB-Lite write transfer is extended until the required entries are available.

29.2.1.3 Rx Data FIFO

The Rx data FIFO consists of eight 8-bit entries. A Tx command FIFO RX_COUNT command specifies the number of bytes to be received; that is, specifies the number of Rx data FIFO entries used. The memory interface transmit logic will stop generating the SPI clock when the Rx data FIFO is full. This is how flow control is achieved.

Software can read the number of used Rx data FIFO entries through the RX_DATA_FIFO_STATUS.USED[3:0] register field.

Software can read from the Rx data FIFO through the MMIO RX_DATA_FIFO_RD1, RX_DATA_FIFO_RD2, and RX_DATA_FIFO_RD4 registers:

- The RX_DATA_FIFO_RD1 register supports a read of a single byte from the FIFO.
- The RX_DATA_FIFO_RD2 register supports a read of two bytes from the FIFO.
- The RX_DATA_FIFO_RD4 register supports a read of four bytes from the FIFO. If software attempts to read more bytes than available in the Rx data FIFO, the MMIO CTL.BLOCK field specifies the behavior:
- If BLOCK is '0', an AHB-Lite bus error is generated and hard fault occurs.

If BLOCK is '1', the AHB-Lite read transfer is extended until the bytes are available.

Software can also read the first byte of the RX data FIFO without changing the status of the FIFO through the RX_DATA_FIFO_RD1_SILENT register.

29.2.2 Command Mode

If CTL.XIP_MODE is '0', the SMIF is in Command mode. Software generates SPI transfers by accessing the Tx FIFOs and Rx FIFO. Software writes to the Tx FIFOs and reads from the Rx FIFO. The Tx command FIFO has formatted commands (Tx, TX_COUNT, RX_COUNT, and DUMMY_COUNT) that are described in the registers TRM.

Software should ensure that it generates correct memory transfers and accesses the FIFOs correctly. For example, if a memory transfer is generated to read four bytes from a memory device, software should read the four bytes from the Rx data FIFO. Similarly, if a memory transfer is generated to write four bytes to a memory device, software should write the four bytes to the Tx command FIFO or Tx data FIFO.

Incorrect software behavior can lock up the memory interface. For example, a memory transfer to read 32 bytes from a memory device, without software reading the Rx data FIFO will lock up the memory transfer as the memory interface cannot provide more than eight bytes to the Rx data FIFO (the Rx data FIFO has eight entries). This will prevent any successive memory transfers from taking place. Hence, the software should make sure that it read the FIFOs to avoid congestion. Note that a locked up memory transfer due to Tx or Rx FIFO states is still compliant to the memory bus protocol (but undesirable): the SPI protocol allows shutting down the interface clock in the middle of a memory transfer.

29.2.3 XIP Mode

If CTL.XIP_MODE is '1', the SMIF is in XIP mode. Hardware automatically (without software intervention) generates memory transfers by accessing the Tx FIFOs and Rx FIFO. Hardware supports only memory read and write transfers. Other functionality such as status reads are not supported. This means operations such as writing into a flash device may not be supported by XIP mode. This is because the writing operation into a flash memory involves not only a write command transfer, but also a status check to verify the status of the operation.

- Hardware generates a memory read transfer for an AHB-Lite read transfer (to be precise: only for AHB-Lite read transfers that miss in the cache).
- Hardware generates a memory write transfer for an AHB-Lite write transfer.

Each slave device slot has a set of associated device configuration registers. To access a memory device in XIP mode, the corresponding device configuration registers (SMIFn_DEVICEn) should be initialized. The device configuration register sets up the following parameters for memory:

- Write enable (WR_EN): Used to disable writes in XIP mode.
- Crypto enable (CRYPTO_EN): When enabled, all read access to the memory is decrypted and write access encrypted automatically.
- Data Select (DATA_SEL): Selects the data lines to be used (Connecting SPI Memory Devices on page 382).

- Base address and size: Sets up the mapped memory space. Any access in this space is converted to the access to the external memory automatically.
- Read and write commands: Used to communicate with the external memory device. These commands are defined by multiple settings.

As different memory devices support different types of memory read and write commands, you must provide the hardware with device specifics, such that it can perform the automatic translations. To this end, each memory device has a set of MMIO registers that specify its memory read and write transfers. This specification includes:

- Presence and value of the SPI command byte.
- Number of address bytes.
- Presence and value of the mode byte.
- Number of dummy cycles.
- Specified data transfer widths.

The XIP interface logic produces an AHB-Lite bus error under the following conditions:

- The SMIF is disabled (SMIFn_CTL.ENABLED is '0').
- The SMIF is not in XIP_MODE (SMIFn_CTL.XIP_MODE is '0').
- The transfer request is not in a memory region.
- The transfer is a write and the identified memory region does not support writes (SMIFn_DEVICEn_CTL.WR_EN is '0').
- In XIP mode (CTL.XIP_MODE is '1') and dual quad SPI mode (ADDR_CTL.DIV2 is '1') or the transfer address is not a multiple of 2.
- In XIP mode (CTL.XIP_MODE is '1') and dual quad SPI mode (ADDR_CTL.DIV2 is '1'), the transfer size is not a multiple of 2.

29.2.4 Cache

To improve XIP performance, the XIP AHB-Lite interface has a cache. The cache is defined as follows:

- 4 KB capacity.
- Read-only cache. Write transfers bypass the cache. A write to an address, which is prefetched in the cache, invalidates the associated cache subsector. If there is a write to a memory in Command mode then you must invalidate the cache while switching back to XIP mode.
- Four-way set associative, with a least recently used (LRU) replacement scheme.

Each XIP interface implements a 4-KB cache memory, enabled by default. Any XIP access can be cached if a cache is enabled. There are separate cache registers for the slow cache (in the clk_slow domain) and fast cache (in the clk_fast domain). The cache can be enabled using the SLOW_CA_CTL[ENABLED] or FAST_CA_CTL[ENABLED] registers. Read transfers that "hit" are processed by the cache. Read transfers that "miss" result in a XIP memory read transfer.

If CA_CTL.PREF_EN is '1', prefetching is enabled and if CA_CTL.PREF_EN is '0', prefetching is disabled. If prefetch is enabled, a cache miss results in a 16 B (subsector) refill for the missing data AND a 16 B prefetch for the next sequential data (independent of whether this data is already in the cache or not).

Cache coherency is not supported by the hardware. For example, an XIP interface 0 write to an address in the XIP interface 0 cache invalidates (clears) the associated cache subsector in the XIP interface 0 cache, but not in the XIP interface 1 cache. This means XIP interface 1 cache now has outdated data. The user code can manually invalidate cache by using the SLOW_CA_CMD[INV] or FAST_CA_CMD[INV] register.

Caches should also be invalidated upon mode transitions. For example, in Command mode, a write to an address in the cache interface will cause the data in the cache interface to be outdated. The cache should be invalidated when transitioning to XIP mode to ensure that only valid data is used.

29.2.5 Arbitration

The SMIF provides two AHB-Lite slave interfaces to CPUSS (one fast interface and one slow interface). Both interfaces have a cache (as described in 29.2.4 Cache) and can generate XIP requests to the external memory devices.

An arbitration component (as shown in Figure 29-1) arbitrates between the two interfaces. Arbitration is based on the master identifier of the AHB-Lite transfer. The arbitration priority is specified by a system wide priority. Each master identifier has a 2-bit priority level ("0" is the highest priority level and "3" is the lowest priority level). Master identifiers with the same priority level are within the same priority group. Within a priority group, round-robin arbitration is performed.

29.2.6 Deselect Delay

The SMIF supports configuration of deselect delay between transfers. The SMIFn_CTL.DESELECT_DELAY field controls the minimum number of interface cycles to hold the chip select line inactive.

29.2.7 Cryptography

In XIP mode, a cryptography component supports on-the-fly encryption for write data and on-the-fly decryption for read data. The use of on-the-fly cryptography is determined by a device's MMIO CTL.CRYPTO_EN field. In Command mode, the cryptography component is accessible through a register interface to support offline encryption and decryption.

The usage scenario for cryptography is: data is encrypted in the external memory devices. Therefore, memory read and write data transfers require decryption and encryption functionality respectively. By storing data encrypted in the external memory devices (nonvolatile devices), leakage of sensitive data is avoided.

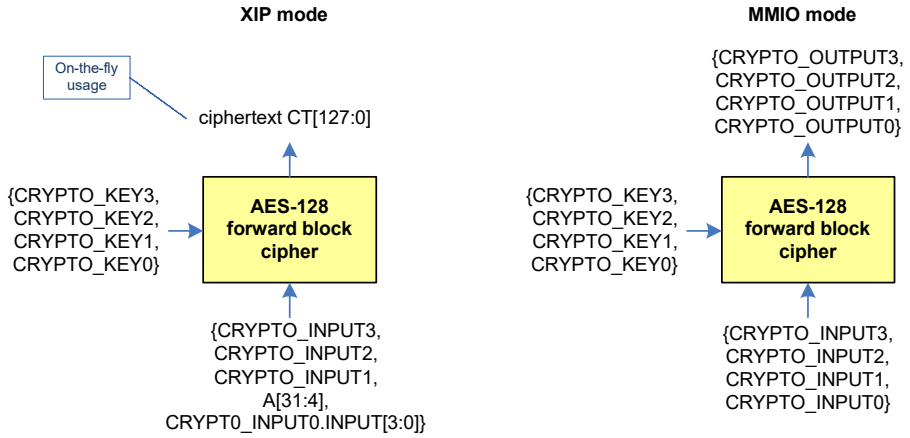
Encryption and decryption are based on the AES-128 forward block cipher: advanced encryption standard block cipher with a 128-bit key. KEY[127:0] is a secret (private) key programmed into the CRYPTO_KEY3, …, CRYPTO_KEY0 registers. These registers are software write-only: a software read returns “0”. In the SMIF hardware, by applying AES-128 with KEY[127:0] on a plaintext PT[127:0], we get a ciphertext CT[127:0].

In XIP mode, the XIP address is used as the plaintext PT[]. The resulting ciphertext CT[] is used on-the-fly and not software accessible. The XIP address is extended with the CRYPTO_INPUT3, …, CRYPTO_INPUT0 registers.

In Command mode, the MMIO CRYPTO_INPUT3, …, CRYPTO_INPUT0 registers provide the plaintext PT[]. The resulting ciphertext CT[] is provided through the MMIO CRYPTO_OUTPUT3, …, CRYPTO_OUTPUT0 registers.

Figure 29-2 illustrates the functionality in XIP and Command modes.

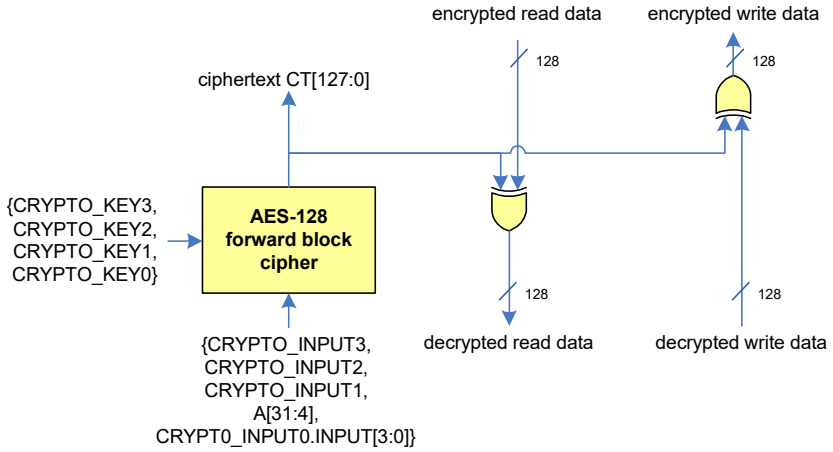
Figure 29-2. Cryptography in XIP and Command Modes



In XIP mode, the resulting ciphertext CT[] (of the encrypted address) is XOR’d with the memory transfer’s read data or write data. Note that the AES-128 block cipher is on the address of the data and not on the data itself. For memory read transfers, this means that as long as the latency of the memory transfer’s read data is longer than the AES-128 block cipher latency, the on-the-fly decryption does not add any delay. Figure 29-3 illustrates the complete XIP mode functionality.

The XIP mode only encrypts the address and XORs with the data; to implement the same in Command mode, you must provide the address as the PT[] into the crypto_INPUTx registers.

Figure 29-3. XIP Mode Functionality



29.3 Memory Device Signal Interface

The SMIF acts as a master for SPI applications. SPI requires the definition of clock polarity and phase. In SPI mode, the SMIF supports a single clock polarity and phase configuration:

- Clock polarity (CPOL) is '0': the base value of the clock is 0.
- Clock phase (CPHA) is '0': driving of data is on the falling edge of the clock; capturing of data is specified by CTL.CLOCK_IF_RX_SEL.

The above configuration is also known as SPI configuration 0 and is supported by SPI memory devices.

29.3.1 Specifying Memory Devices

The SMIF requires that the memory devices are defined for their operation in XIP mode. The SMIF supports up to four memory devices. Each memory device is defined by a set of registers. The memory device specific register structure includes:

- The device base address and capacity. The ADDR register specifies the memory device's base address in the PSoC 6 MCU address space and the MASK register specifies the memory device's size/capacity. If a memory device is not present or is disabled, the ADDR and MASK registers specify a memory device with 0 B capacity. Typically, the devices' address regions in the PSoC 6 MCU address space are non-overlapping (except for dual-quad SPI mode) to ensure that the activation of select signals is mutually exclusive.
- The device data signal connections (as described in Connecting SPI Memory Devices on page 382).
- The definition of a read transfer to support XIP mode.
- The definition of a write transfer to support XIP mode.

Each memory device uses a dedicated device select signal: memory device 0 uses select[0], memory device 1 uses select[1], and so on. In other words, there is a fixed, one-to-one connection between memory device, register set, and select signal connection.

In XIP mode, the XIP AHB-Lite bus transfer address is compared with the device region. If the address is within the device region, the device select signal is activated. If an XIP AHB-Lite bus transfer address is within multiple regions (this is possible if the device regions overlap), all associated device select signals are activated. This overlap enables XIP in dual-quad SPI mode: the command, address, and mode bytes can be driven to two quad SPI devices simultaneously.

In XIP mode, dual quad SPI mode requires the ADDR_CTL.DIV2 field of the selected memory devices to be set to '1'. When this field is '1', the transfer address is divided by 2 and the divided by 2 address is provided to the memory devices.

In dual quad SPI mode, each memory device contributes a 4-bit nibble for each 8-bit byte. However, both memory devices are quad SPI memories with a byte interface. Therefore, the transfer size must be a multiple of 2.

The XIP_ALIGNMENT_ERROR interrupt cause is set under the following conditions (in XIP mode and when ADDR_CTL.DIV2 is '1'):

- The transfer address is not a multiple of 2. In this case the divided by 2 address for the memory devices is incorrect.
- The transfer size is not a multiple of 2. In this case, the memory devices contribute only a nibble of a byte. This is not supported as the memory devices have a byte interface.

29.3.2 Connecting SPI Memory Devices

Memory device I/O signals (SCK, $\overline{CS}$, SI/IO0, SO/IO1, IO2, IO3, IO4, IO5, IO6, IO7) are connected to the SMIF I/O signals (clk, select[3:0], and spi_data[7:0]). Not all memory devices provide the same number of I/O signals.

Table 29-1. Memory Device I/O Signals

Memory Device	IO Signals
Single SPI memory	SCK, $\overline{CS}$, SI, SO. This memory device has two data signals (SI and SO).
Dual SPI memory	SCK, $\overline{CS}$, IO0, IO1. This memory device has two data signals (IO0, IO1).
Quad SPI memory	SCK, $\overline{CS}$, IO0, IO1, IO2, IO3. This memory device has four data signals (IO0, IO1, IO2, IO3).
Octal SPI memory	SCK, $\overline{CS}$, IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7. This memory device has eight data signals (IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7).

Table 29-1 illustrates that each memory has a single clock signal SCK, a single (low active) select signal ($\overline{CS}$), and multiple data signals (IO0, IO1, …).

Each memory device has a fixed select signal connection (to select[3:0]).

Each memory device has programmable data signal connections (to data[7:0]): the CTL.DATA_SEL[1:0] field specifies how a device's data signals are connected. The CTL.DATA_SEL[1:0] is responsible for configuring the selection of data lines to be used by a slave. This is not to be confused with the select lines that are used for addressing the four slaves of the SMIF master. This information is used by the SMIF interface to drive out data on the correct spi_data[] outputs and capture data from the correct spi_data[] inputs. If multiple device select signals are activated, the same data is driven to all selected devices simultaneously.

Not all data signal connections are legal/supported. Supported connections are dependent on the type of memory device.

Table 29-2. Data Signal Connections

DATA_SEL[1:0]	Single SPI Device	Dual SPI Device	Quad SPI Device	Octal SPI Device
0	spi_data[0] = SI spi_data[1] = SO	spi_data[0] = IO0 spi_data[1] = IO1	spi_data[0] = IO0 … spi_data[3] = IO3	spi_data[0] = IO0 … spi_data[7] = IO7
1	spi_data[2] = SI spi_data[3] = SO	spi_data[2] = IO0 spi_data[3] = IO1	Illegal	Illegal
2	spi_data[4] = SI spi_data[5] = SO	spi_data[4] = IO0 spi_data[5] = IO1	spi_data[4] = IO0 … spi_data[7] = IO3	Illegal
3	spi_data[6] = SI spi_data[7] = SO	spi_data[6] = IO0 spi_data[7] = IO1	Illegal	Illegal

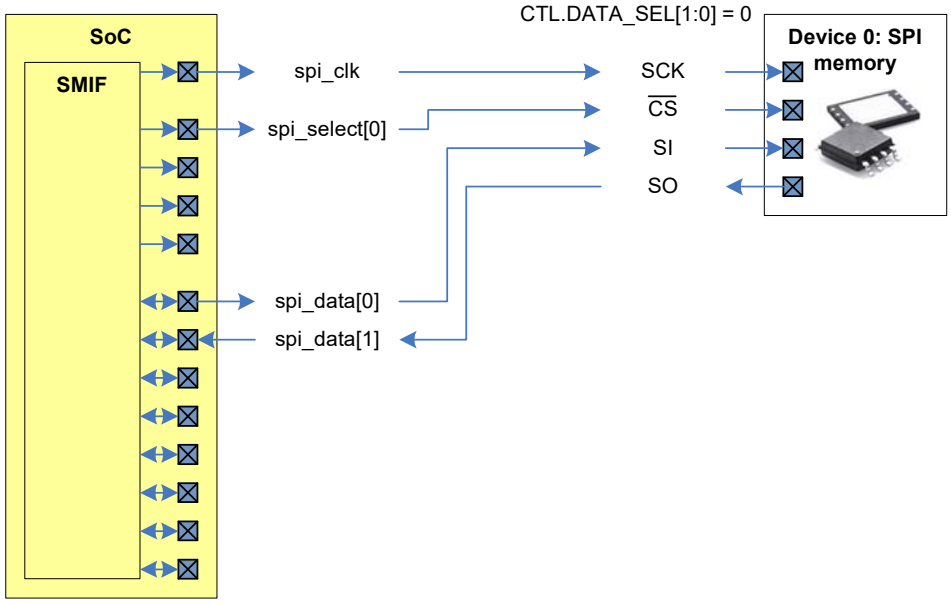
Memory devices can:

- Use shared data signal connections.
- Use dedicated data signal connections. This reduces the load on the data lines allowing faster signal level changes, which in turn allows for a faster I/O interface.

Note that dual-quad SPI mode requires dedicated data signals to enable read and/or write data transfer from and to two quad SPI devices simultaneously.

Figure 29-4 illustrates memory device 0, which is a single SPI memory with data signals connections to spi_data[1:0].

Figure 29-4. Single SPI Memory Device 0 Connected to spi_data[1:0]



Because of the pin layout, you might want to connect a memory device to specific data lines. Figure 29-5 illustrates memory device 0, which is a single SPI memory with data signals connections to spi_data[7:6].

Figure 29-5. Single SPI Memory Device 0 Connected to spi_data[7:6]

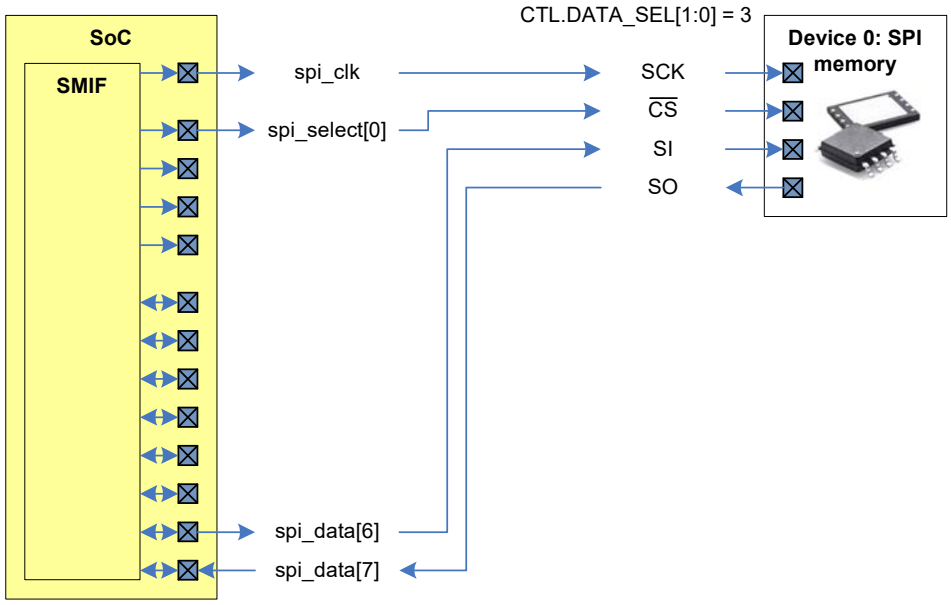


Figure 29-6 illustrates memory devices 0 and 1, both of which are single SPI memories. Each device uses dedicated data signal connections. The device address regions in the PSoC 6 MCU address space must be non-overlapping to ensure that the activation of select[0] and select[1] are mutually exclusive.

Figure 29-6. Single SPI Memory Devices 0 and 1 - Dedicated Data Signal

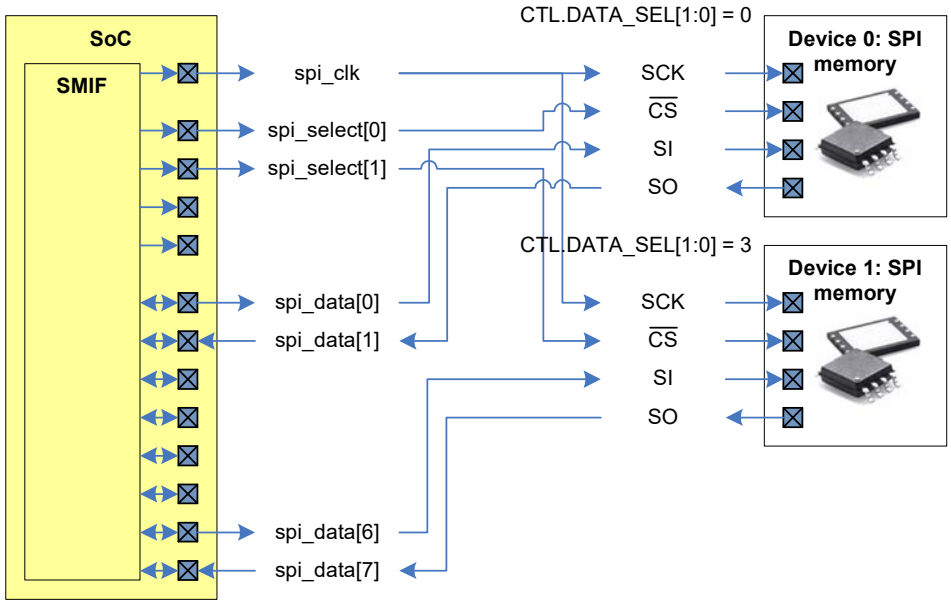


Figure 29-7 illustrates memory devices 0 and 1, both of which are single SPI memories. Both devices use shared data signal connections. The devices' address regions in the PSoC 6 MCU address space must be non-overlapping to ensure that the activation of select[0] and select[1] are mutually exclusive. Note that this solution increases the load on the data lines, which may result in a slower I/O interface.

Figure 29-7. Single SPI Memory Devices 0 and 1 - Shared Data Signal

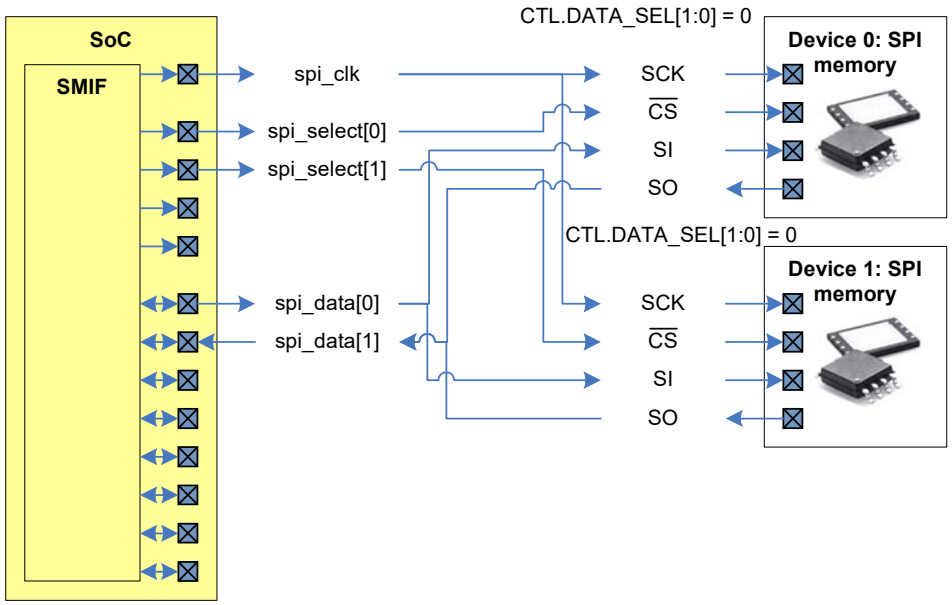


Figure 29-8 illustrates memory device 0, which is a quad SPI memory with data signals connections to spi_data[7:4].

Figure 29-8. Quad SPI Memory Device 0

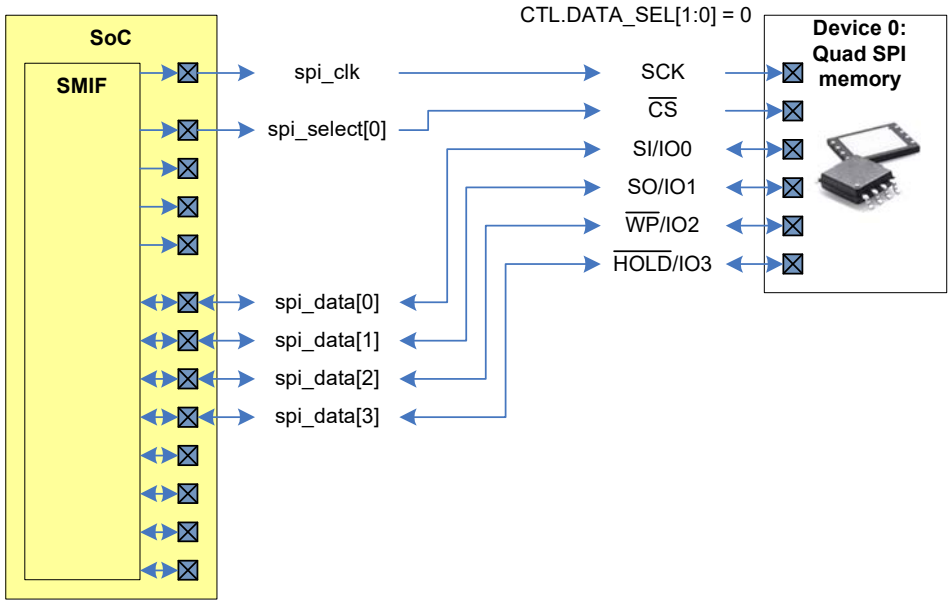


Figure 29-9 illustrates memory devices 0 and 1, device 0 is a single SPI memory and device 1 is a quad SPI memory. Each device uses dedicated data signal connections. The device address regions in the PSoC 6 MCU address space must be non-overlapping to ensure that the activation of select[0] and select[1] are mutually exclusive.

Figure 29-9. Single SPI Memory 0 and Quad SPI Memory 1 - Dedicated Data Signal

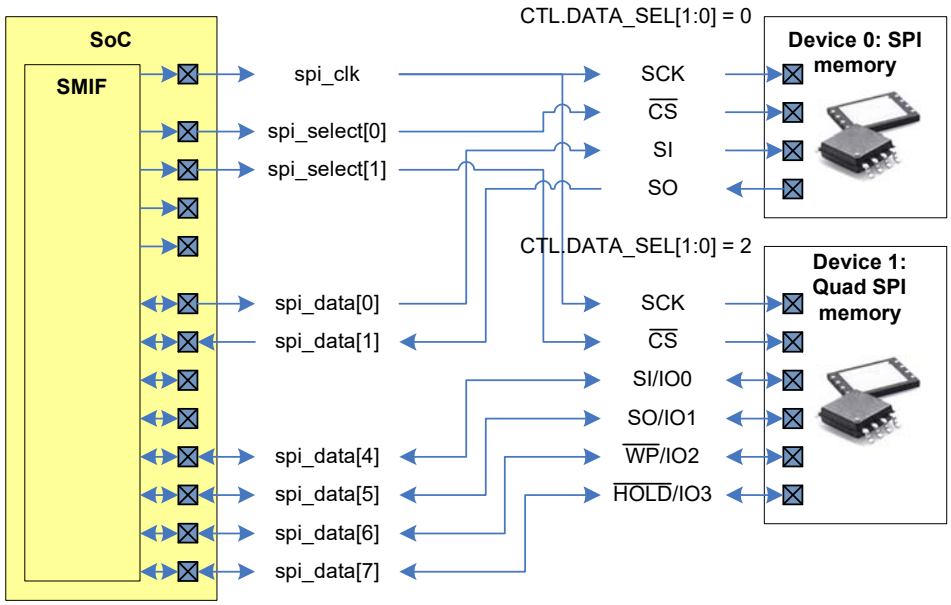


Figure 29-11 illustrates memory devices 0 and 1, device 0 is a single SPI memory and device 1 is a quad SPI memory. Both devices use shared data signal connections. The device address regions in the PSoC 6 MCU address space must be non-overlapping to ensure that the activation of select[0] and select[1] are mutually exclusive.

Figure 29-10. Single SPI Memory Device 0 and Quad SPI Memory Device 1 - Shared Data Signal

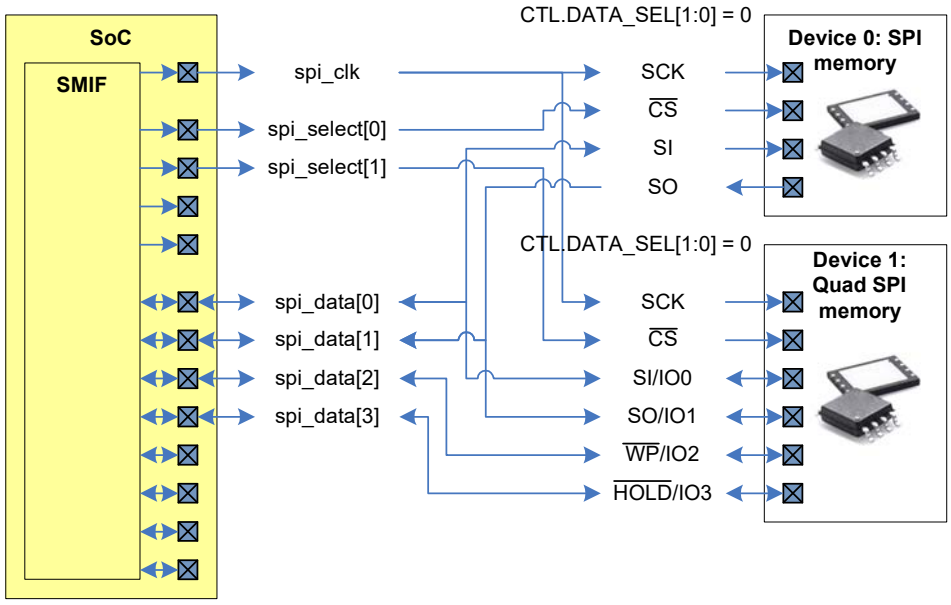


Figure 29-12 illustrates memory devices 0 and 1, both of which are quad SPI memories. Each device uses dedicated data signal connections. The device address regions in the PSoC 6 MCU address space are the same to ensure that the activation of select[0] and select[1] are the same (in XIP mode). This is known as a dual-quad configuration: during SPI read and write transfers, each device provides a nibble of a byte.

Figure 29-11. Quad SPI Memory Devices 0 and 1

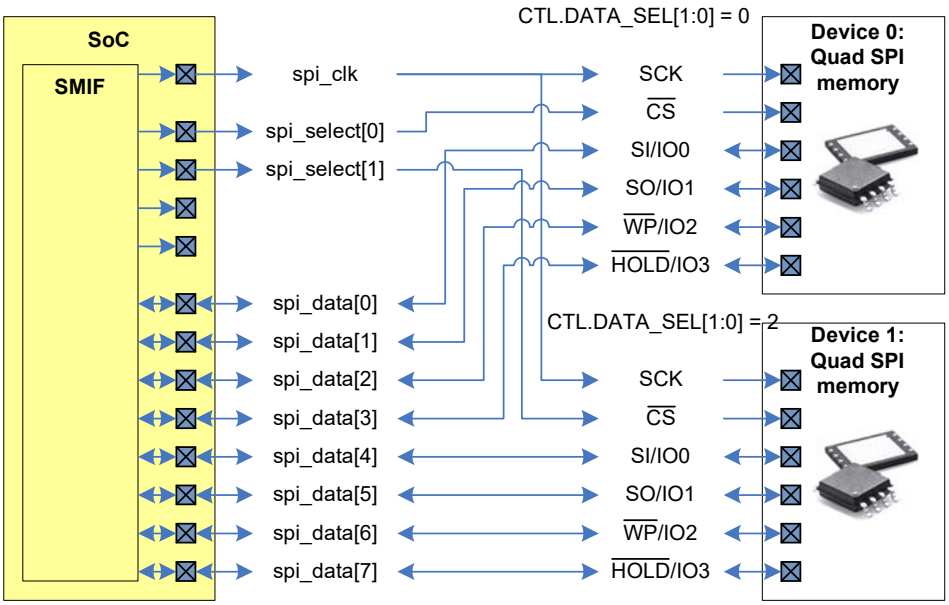
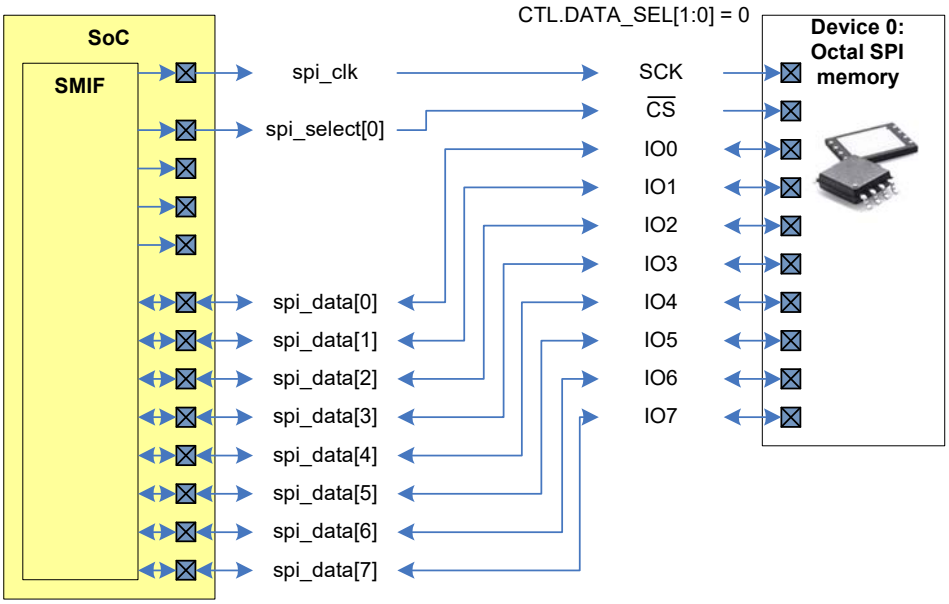


Figure 29-12 illustrates memory device 0, which is a octal SPI memory with data signals connections to spi_data[7:0].

Figure 29-12. Octal SPI Memory Device 0



29.3.3 SPI Data Transfer

SPI data transfer uses most-significant-bit (MSb) for the first data transfer. This means that for a byte B, consisting of bits b7, b6, …, b0, bit b7 is transferred first, followed by bit b6, and so on. For dual, quad, dual quad, and octal SPI transfers, multiple

bits are transferred per cycle. For a single SPI device and device data signal connections to spi_data[1:0] (DATA_SEL is “0”), Table 29-3 summarizes the transfer of byte B.

Table 29-3. Single Data Transfer

Cycle	Data Transfer
0	For a write transfer: b7 is transferred on data[0] and SI/IO0. For a read transfer: b7 is transferred on data[1] and SO/IO1.
1	For a write transfer: b6 is transferred on data[0] and SI/IO0. For a read transfer: b6 is transferred on data[1] and SO/IO1.
2	For a write transfer: b5 is transferred on data[0] and SI/IO0. For a read transfer: b5 is transferred on data[1] and SO/IO1.
3	For a write transfer: b4 is transferred on data[0] and SI/IO0. For a read transfer: b4 is transferred on data[1] and SO/IO1.
4	For a write transfer: b3 is transferred on data[0] and SI/IO0. For a read transfer: b3 is transferred on data[1] and SO/IO1.
5	For a write transfer: b2 is transferred on data[0] and SI/IO0. For a read transfer: b2 is transferred on data[1] and SO/IO1.
6	For a write transfer: b1 is transferred on data[0] and SI/IO0. For a read transfer: b1 is transferred on data[1] and SO/IO1.
7	For a write transfer: b0 is transferred on data[0] and SI/IO0. For a read transfer: b0 is transferred on data[1] and SO/IO1.

Note that in single SPI data transfer, the data signals are uni-directional: in the table, data[0] is exclusively used for write data connected to the device SI input signal and data[1] is exclusively used for read data connected to the device SO output signal.

For a dual SPI device and device data signal connections to data[1:0] (DATA_SEL is “0”), Table 29-4 summarizes the transfer of byte B.

Table 29-4. Dual Data Transfer

Cycle	Data Transfer
0	b7, b6 are transferred on data[1:0] and IO1, IO0.
1	b5, b4 are transferred on data[1:0] and IO1, IO0.
2	b3, b2 are transferred on data[1:0] and IO1, IO0.
3	b1, b0 are transferred on data[1:0] and IO1, IO0.

For a quad SPI device and device data signal connections to data[3:0] (DATA_SEL is “0”), Table 29-5 summarizes the transfer of byte B.

Table 29-5. Quad Data Transfer

Cycle	Data Transfer
0	b7, b6, b5, b4 are transferred on data[3:0] and IO3, IO2, IO1, IO0.
1	b3, b2, b1, b0 are transferred on data[3:0] and IO3, IO2, IO1, IO0.

For a octal SPI device and device data signal connections to data[7:0] (DATA_SEL is “0”), Table 29-6 summarizes the transfer of byte B.

Table 29-6. Octal Data Transfer

Cycle	Data Transfer
0	b7, b6, b5, b4, b3, b2, b1, b0 are transferred on data[7:0] and IO7, IO6, IO5, IO4, IO3, IO2, IO1, IO0.

In dual-quad SPI mode, two quad SPI devices are used.

- The first device (the device with the lower device structure index) should have device data signal connections to data[3:0] (DATA_SEL is 0).
- The second device (the device with the higher device structure index) should have device data signal connection to data[7:4] (DATA_SEL is 2).

The command and data phases of the SPI transfer use different width data transfers:

- The command, address, and mode bytes use quad SPI data transfer.
- The read data and write data use octal data transfer. Each device provides a nibble of each data byte: the first device provides the lower nibble and the second device provides the higher nibble.

Table 29-7 summarizes the transfer of a read data and write data byte B.

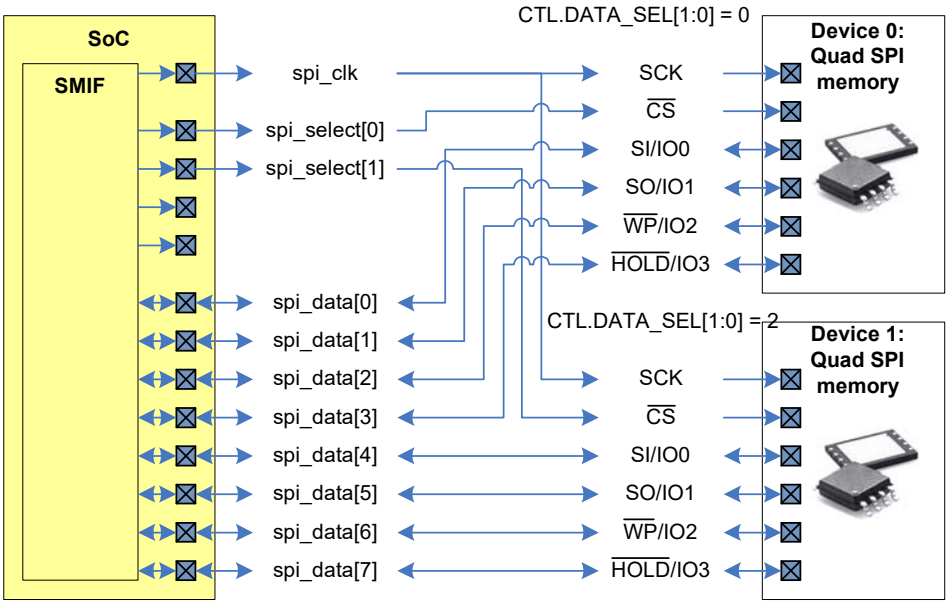
Table 29-7. Dual-quad SPI Mode, Octal Data Transfer

Cycle	Data transfer
0	b7, b6, b5, b4 are transferred on data[7:4] and second device IO3, IO2, IO1, IO0. b3, b2, b1, b0 are transferred on data[3:0] and first device IO3, IO2, IO1, IO0.

29.3.4 Example of Setting up SMIF

Devices 0 and 1 are used to implement the dual-quad SPI mode. Both devices are 1 MB / 8 Mb; the address requires 3 bytes. Device 0 has device data signal connections to data[3:0] and device 1 has device data signal connections to data[7:4].

Figure 29-13. Setting up SMIF



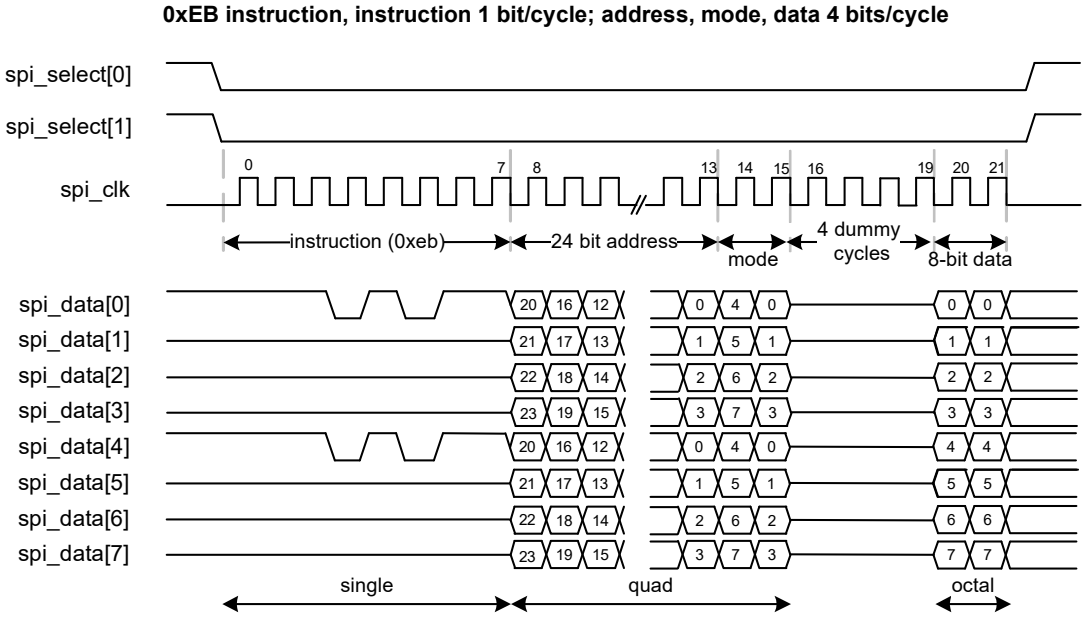
For dual quad SPI mode, the AHB-Lite bus transfer address is divided by two. Cryptography and write functionality are disabled in the following example.

```
#define MASK_1MB 0xfff00000;
DEV0_ADDR     = CPUSS_SMIF_BASE;
DEV0_MASK     = MASK_1MB     // MASK: 1 MB region
DEV0_CTL      =(0 << SMIF_DEVICE_CTL_DATA_SEL_Pos)        // DATA_SEL: data[3:0]
                  | (0 << SMIF_DEVICE_CTL_CRYPTO_EN_Pos)         // CRYPTO_EN
                  | (0 << SMIF_DEVICE_CTL_WR_EN_Pos));       // WR_EN
DEV0_ADDR_CTL =  (1 << SMIF_DEVICE_ADDR_CTL_DIV2_Pos)         // DIV2: enabled
                  | ((3-1) << SMIF_DEVICE_ADDR_CTL_SIZE2_Pos));   // SIZE: 3 B address
```

```
DEV1_ADDR      = CPUSS_SMIF_BASE;
DEV1_MASK      = 0xfff00000;     // MASK: 1 MB region
DEV1_CTL       =   (2 << SMIF_DEVICE_CTL_DATA_SEL_Pos)        // DATA_SEL: data[7:4]
                 | (0 << SMIF_DEVICE_CTL_CRYPTO_EN_Pos)         // CRYPTO_EN
                 | (1 << SMIF_DEVICE_CTL_WR_EN_Pos));       // WR_EN
DEV1_ADDR_CTL =   (1 << SMIF_DEVICE_ADDR_CTL_DIV2_Pos)         // DIV2: enabled
                 | ((3-1) << SMIF_DEVICE_ADDR_CTL_SIZE2_Pos));   // SIZE: 3 B address
```

For XIP read transfers, the 0xEB command/instruction is used (Figure 29-14 illustrates a two-byte transfer from devices 0 and 1 in dual quad SPI mode).

Figure 29-14. Two-Bye Transfer in Dual Quad SPI Mode



The definition of a read transfer is as follows:

```
DEV0_RD_CMD_CTL  = (1UL << SMIF_DEVICE_RD_CMD_CTL_PRESENT_Pos) // PRESENT
                   | (0 << SMIF_DEVICE_RD_CMD_CTL_WIDTH_Pos) // WIDTH: single data transfer
  | 0xeb);                                                   // CODE
DEV0_RD_ADDR_CTL = (2 << SMIF_DEVICE_RD_ADDR_CTL_WIDTH_Pos));  // WIDTH: quad data transfer
DEV0_RD_MODE_CTL = (1UL << SMIF_DEVICE_RD_MODE_CTL_PRESENT_Pos) // PRESENT
                   | (2 << SMIF_DEVICE_RD_MODE_CTL_WIDTH_Pos) // WIDTH: quad data transfer
                   | 0x00);        // CODE
DEV0_RD_DUMMY_CTL=  (1UL << SMIF_DEVICE_RD_DUMMY_CTL_PRESENT_Pos)     // PRESENT
                  | ((4-1) << SMIF_DEVICE_RD_DUMMY_CTL_SIZE5_Pos));  // SIZE: 4 dummy cycles
DEV0_RD_DATA_CTL = (3 << SMIF_DEVICE_RD_DATA_CTL_WIDTH_Pos)); // WIDTH: octal data transfer
```

Note that the command uses single data transfer, the address and mode byte use quad data transfer, and the read data byte uses octal data transfer.

29.4 Triggers

The SMIF has two level-sensitive triggers:

- tr_tx_req is associated with the Tx data FIFO.
- tr_rx_req is associated with the Rx data FIFO.

If the SMIF is enabled (CTL.ENABLED is '1') and Command operation mode is selected (CTL.XIP_MODE is '0'), the trigger functionality is enabled. If the SMIF is disabled (CTL.ENABLED is '0') or the XIP operation mode is selected (CTL.XIP_MODE is '1'), the triggers functionality is disabled. The trigger functionality is defined as follows:

- The MMIO TX_DATA_FIFO_CTL.TRIGGER_LEVEL field specifies a number of FIFO entries. The tr_tx_req trigger is active when the number of used Tx data FIFO entries is smaller or equal than the specified number; that is, TX_DATA_FIFO_STATUS.USED ≤ TRIGGER_LEVEL.
- The MMIO RX_DATA_FIFO_CTL.TRIGGER_LEVEL field specifies a number of FIFO entries. The tr_rx_req trigger is active when the number of used Rx data FIFO entries is greater than the specified number; that is, RX_DATA_FIFO_STATUS.USED > TRIGGER_LEVEL.

29.5 Interrupts

The SMIF has a single interrupt output with six interrupt causes:

- INTR.TR_TX_REQ. This interrupt cause is activated in Command mode when the tr_tx_req trigger is activated.
- INTR.TR_RX_REQ. This interrupt cause is activated in Command mode when the tr_rx_req trigger is activated.
- INTR.XIP_ALIGNMENT_ERROR. This interrupt cause is activated in XIP mode when the selected device's ADDR_CTL.DIV2 field is '1' and the AHB-Lite bus address is not a multiple of 2, or the requested transfer size is not a multiple of 2. This interrupt cause identifies erroneous behavior in dual-quad SPI mode (the selected device ADDR_CTL.DIV2 field is set to '1').
- INTR.TX_CMD_FIFO_OVERFLOW. This interrupt cause is activated in Command mode, on an AHB-Lite write transfer to the Tx command FIFO (TX_CMD_FIFO_WR) with insufficient free entries.
- INTR.TX_DATA_FIFO_OVERFLOW. This interrupt cause is activated in Command mode, on an AHB-Lite write transfer to the Tx data FIFO (TX_DATA_FIFO_WR1, TX_DATA_FIFO_WR2, and TX_DATA_FIFO_WR4) with insufficient free entries.
- INTR.RX_DATA_FIFO_OVERFLOW. This interrupt cause is activated in Command mode, on an AHB-Lite read transfer from the Rx data FIFO (RX_DATA_FIFO_RD1, RX_DATA_FIFO_RD2, and RX_DATA_FIFO_RD4) with insufficient free entries.

29.6 Sleep Operation

The SMIF hardware is operational in the Sleep and Active power modes. In the Sleep power mode, only DMA driven transactions can be performed because the CPU acting as the SPI master is not active.

29.7 Performance

Accesses to the external memory will have some latency, which is dependent upon the mode of SMIF operation, the amount of data being transferred, caching, and cryptography. In Command mode, the number of interface clock cycles per transfer is determined by the equation:

$$\underset{\textit{(in interface clock cycles)}}{\textit{Time}\text{ to Transfer N bytes}} = \left(\left(\frac{\textit{opcode}\text{ size}}{\textit{opcode}\text{ width}}\right) + \left(\frac{\textit{address}\text{ size}}{\textit{address}\text{ width}}\right) + \left(\frac{\textit{mode}\text{ size}}{\textit{mode}\text{ width}}\right) + \textit{dummycycles} + \left(\frac{\textit{data}\text{ size}}{\textit{data}\text{ width}}\right)\right)$$

For example, in Figure 29-14, the equation to calculate the number of cycles would be (22 cycles = [8 bit instruction/1 single width] + [24 bit address/4 width] + [8 bit mode/4 width] + 4 dummy cycles + [16 bit data/8 width]).

In XIP Mode, the performance is affected by the cache. For data reads that hit in the cache, the read will not incur any interface cycles. Read operations that miss in the cache will occur as a normal SMIF read operation and, if prefetching is enabled, will result in two 16 B cache sub-sector refills. Writes to external memory in XIP mode will occur as a normal SMIF write operation.

Enabling cryptography may impact SMIF performance. The AES-128 block cipher has a typical latency of 13 clk_hf cycles. This means that for transfers that take more than 13 cycles, the on-the-fly decryption does not add any delay. If the transfer is less than 13 cycles, the transfer latency will be 13 cycles. When the cache is enabled, the 13-cycle latency for encryption is incurred only once for every 16 B fetched for the cache.

30. Timer, Counter, and PWM (TCPWM)

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

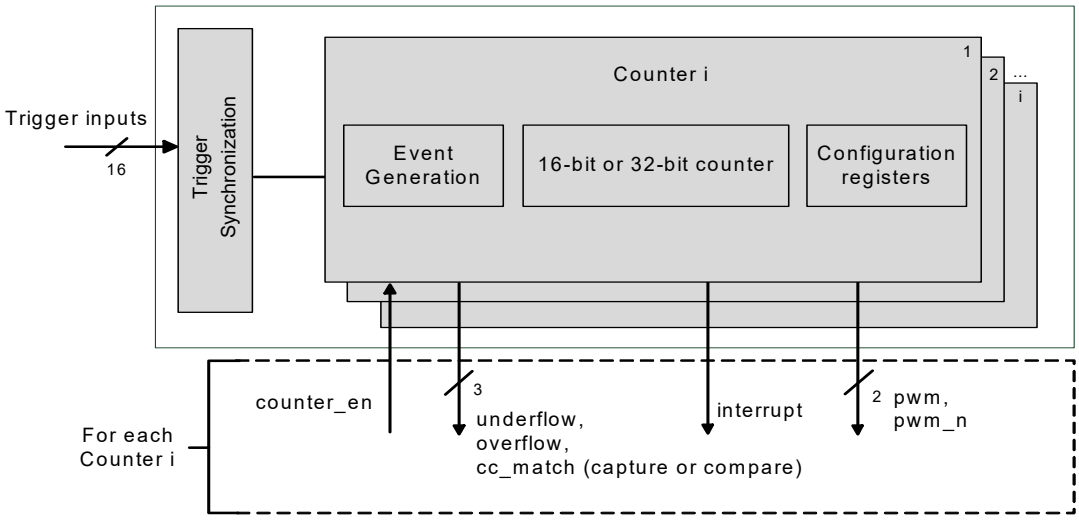
The Timer, Counter, Pulse Width Modulator (TCPWM) block in the PSoC 6 MCU uses a 16- or 32-bit counter, which can be configured as a timer, counter, pulse width modulator (PWM), or quadrature decoder. The block can be used to measure the period and pulse width of an input signal (timer), find the number of times a particular event occurs (counter), generate PWM signals, or decode quadrature signals. This chapter explains the features, implementation, and operational modes of the TCPWM block.

30.1 Features

- The TCPWM block supports the following operational modes:
  - Timer-counter with compare
  - Timer-counter with capture
  - Quadrature decoding
  - Pulse width modulation
  - Pseudo-random PWM
  - PWM with dead time
- Up, Down, and Up/Down counting modes.
- Clock prescaling (division by 1, 2, 4, ... 64, 128)
- 16- or 32-bit counter widths
- Double buffering of compare/capture and period values
- Underflow, overflow, and capture/compare output signals
- Supports interrupt on:
  - Terminal count – Depends on the mode; typically occurs on overflow or underflow
  - Capture/compare – The count is captured to the capture register or the counter value equals the value in the compare register
- Complementary output for PWMs
- Selectable start, reload, stop, count, and capture event signals (events refer to peripheral generated signals that trigger specific functions in each counter in the TCPWM block) for each TCPWM – with rising edge, falling edge, both edges, and level trigger options

30.2 Architecture

Figure 30-1. TCPWM Block Diagram



The TCPWM block can contain up to 32 counters. Each counter can be 16- or 32-bit wide. The three main registers that control the counters are:

- TCPWM_CNT_CC is used to capture the counter value in CAPTURE mode. In all other modes this value is compared to the counter value.
- TCPWM_CNT_COUNTER holds the current counter value.
- TCPWM_CNT_PERIOD holds the upper value of the counter. When the counter counts for n cycles, this field should be set to n–1.

The number of 16- and 32-bit counters are device specific; refer to the PSoC 61 datasheet/PSoC 62 datasheet for details.

In this chapter, a TCPWM refers to the entire block and all the counters inside. A counter refers to the individual counter inside the TCPWM. Within a TCPWM block the width of each counter is the same.

TCPWM has these interfaces:

- I/O signal interface: Consists of input triggers (such as reload, start, stop, count, and capture) and output signals (such as pwm, pwm_n, overflow (OV), underflow (UN), and capture/compare (CC)). All of these input signals are used to trigger an event within the counter, such as a reload trigger generating a reload event. The output signals are generated by internal events (underflow, overflow, and capture/compare) and can be connected to other peripherals to trigger events.
- Interrupts: Provides interrupt request signals from each counter, based on TC or CC conditions.

The TCPWM block can be configured by writing to the TCPWM registers. See “TCPWM Registers” on page 428 for more information on all registers required for this block.

30.2.1 Enabling and Disabling Counters in a TCPWM Block

A counter can be enabled by setting the corresponding bit of the TCPWM_CTRL_SET register. It can be disabled by setting the bit in the TCPWM_CTRL_CLR register. These registers are used to avoid race-conditions on read-modify-write attempts to the TCPWM_CTRL register, which controls the enable/disable fields of the counters.

Note: The counter must be configured before enabling it. Disabling the counter retains the values in the configuration registers.

30.2.2 Clocking

Each TCPWM counter can have its own clock source and the only source for the clock is from the configurable peripheral clock dividers generated by the clocking system; see the Clocking System chapter on page 242 for details. To select a clock divider for a particular counter inside a TCPWM, use the CLOCK_CTL register from the PERI register space.

In this section the clock to the counter will be called clk_counter. Event generation is performed on clk_counter. Another clock, clk_sys is used for the pulse width of the output triggers. clk_sys is synchronous to clk_peri (see “CLK_PERI” on page 253), but can be divided using CLOCK_CTL from the PERI_GROUP_STRUCT register space.

30.2.2.1 Clock Prescaling

clk_counter can be further divided inside each counter, with values of 1, 2, 4, 8…64, 128. This division is called prescaling. The prescaling is set in the GENERIC field of the TCPWM_CNT_CTLR register.

Note: Clock prescaling is not available in quadrature mode and pulse width modulation mode with dead time.

30.2.2.2 Count Input

The counter increments or decrements on a prescaled clock in which the count input is active – “active count”.

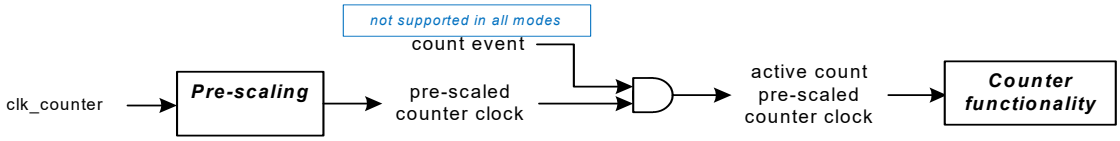
When the count input is configured as level, the count value is changed on each prescaled clk_counter edge in which the count input is high.

When the count input is configured as rising/falling the count value is changed on each prescaled clk_counter edge in which an edge is detected on the count input.

The next section contains additional details on edge detection configuration.

Note: Count events are not supported in quadrature and pulse-width modulation pseudo-random modes; the clk_counter is used in these cases instead of the active count prescaled clock.

Figure 30-2. Counter Clock Generation



Note: The count event and pre-scaled counter clock are AND together, which means that a count event must occur to generate an active count pre-scaled counter clock.

30.2.3 Trigger Inputs

Each TCPWM block has 14 Trigger_In signals, which come from other on-chip resources such as other TCPWMs, SCBs, or DMA. The Trigger_In signals are shared with all counters inside of one TCPWM block. Use the Trigger Mux registers to configure which signals get routed to the Trigger_In for each TCPWM block. See the Trigger Multiplexer Block chapter on page 294 for more details. Two constant trigger inputs ‘0’ and ‘1’ are available in addition to the 14 Trigger_In. For each counter, the trigger input source is selected using the TCPWM_CNT_TR_CTRL0 register.

Each counter can select any of the 16 trigger signals to be the source for any of the following events:

- Reload
- Start
- Stop/Kill
- Count
- Capture/swap

When starting a TCPWM for the first time it is recommended that a reload is used, because a reload will generate an overflow or underflow on startup. If start is used, an overflow or underflow will not be generated on startup.

The TCPWM_CMD_RELOAD, TCPWM_CMD_STOP, TCPWM_CMD_START, and TCPWM_CMD_CAPTURE registers can be used to trigger the reload, stop, start, and capture respectively from software.

The sections describing each TCPWM mode will describe the function of each input in detail.

Typical operation uses the reload input to initialize and start the counter and the stop input to stop the counter. When the counter is stopped, the start input can be used to start the counter with its counter value unmodified from when it was stopped.

If stop, reload, and start coincide, the following precedence relationship holds:

- A stop has higher priority than a reload.
- A reload has higher priority that a start.

As a result, when a reload or start coincides with a stop, the reload or start has no effect.

Before going to the counter each Trigger_IN can pass through a positive edge detector, negative edge detector, both edge detector, or pass straight through to the counter. This is controlled using TCPWM_CNT_TR_CTRL1. In the quadrature mode, edge detection is done using clk_counter. For all other modes, edge detection is done using the clk_peri.

Multiple detected events are treated as follows:

- In the rising edge and falling edge modes, multiple events are effectively reduced to a single event. As a result, events may be lost (see Figure 30-3).
- In the rising/falling edge mode, an even number of events are not detected and an odd number of events are reduced to a single event. This is because the rising/falling edge mode is typically used for capture events to determine the width of a pulse. The current functionality will ensure that the alternating pattern of rising and falling is maintained (see Figure 30-4).

Figure 30-3. Multiple Rising Edge Capture

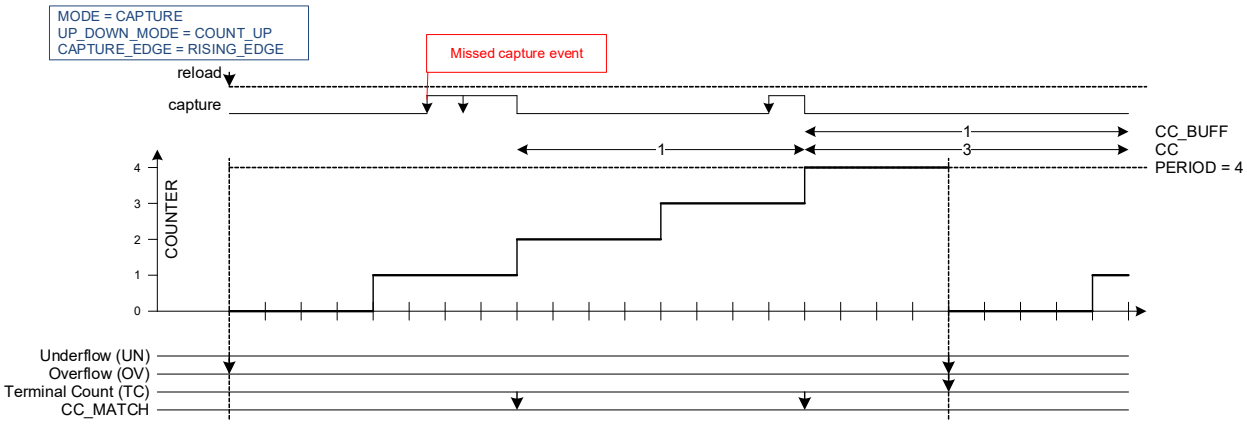


Figure 30-4. Multiple Both Edge Capture

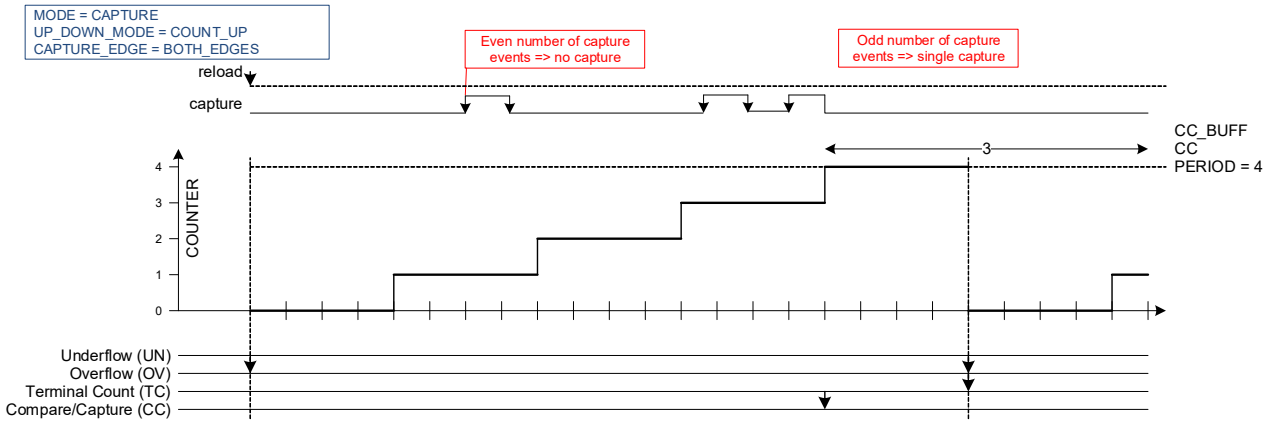
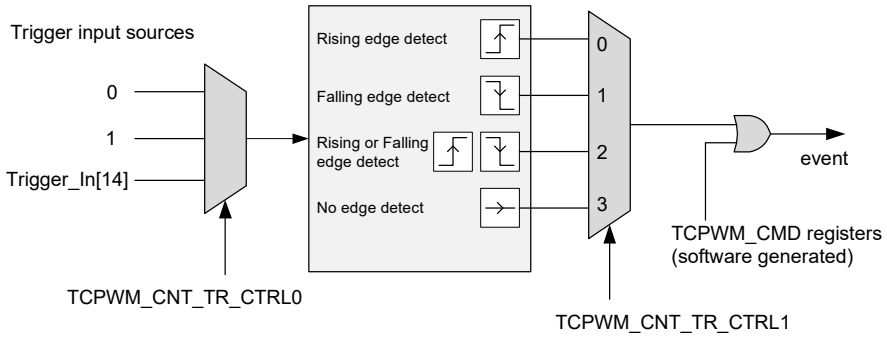


Figure 30-5. TCPWM Input Events



Notes:

- All trigger inputs are synchronized to clk_peri.
- When more than one event occurs in the same clk_counter period, one or more events may be missed. This can happen for high-frequency events (frequencies close to the counter frequency) and a timer configuration in which a pre-scaled (divided) clk_counter is used.

30.2.4 Trigger Outputs

Each counter can generate three trigger output events. These trigger output events can be routed through the trigger mux to other peripherals on the device. The three trigger outputs are:

- Overflow (OV): An overflow event indicates that in up-counting mode, COUNTER equals the PERIOD register, and is changed to a different value.
- Underflow (UN): An underflow event indicates that in down-counting mode, COUNTER equals 0, and is changed to a different value.
- Compare/Capture (CC): This event is generated when the counter is running and one of the following conditions occur:
  - Counter equals the compare value. This event is either generated when the match is about to occur (COUNTER does not equal the CC register and is changed to CC) or when the match is not about to occur (COUNTER equals CC and is changed to a different value).
  - A capture event has occurred and the CC/CC_BUFF registers are updated.

Note: These signals remain high only for two cycles of clk_sys.

30.2.5 Interrupts

The TCPWM block provides a dedicated interrupt output for each counter. This interrupt can be generated for a terminal count (TC) or CC event. A TC is the logical OR of the OV and UN events.

Four registers are used to handle interrupts in this block, as shown in Table 30-1.

Table 30-1. Interrupt Register

Interrupt Registers	Bits	Name	Description
TCPWM_CNT_INTR (Interrupt request register)	0	TC	This bit is set to ‘1’, when a terminal count is detected. Write '1' to clear this bit.
	1	CC_MATCH	This bit is set to ‘1’ when the counter value matches capture/compare register value. Write '1' to clear this bit.
TCPWM_CNT_INTR_SET (Interrupt set request register)	0	TC	Write '1' to set the corresponding bit in the interrupt request register. When read, this register reflects the interrupt request register status.
	1	CC_MATCH	Write '1' to set the corresponding bit in the interrupt request register. When read, this register reflects the interrupt request register status.
TCPWM_CNT_INTR_MASK (Interrupt mask register)	0	TC	Mask bit for the corresponding TC bit in the interrupt request register.
	1	CC_MATCH	Mask bit for the corresponding CC_MATCH bit in the interrupt request register.
TCPWM_CNT_INTR_MASKED (Interrupt masked request register)	0	TC	Logical AND of the corresponding TC request and mask bits.
	1	CC_MATCH	Logical AND of the corresponding CC_MATCH request and mask bits.

30.2.6 PWM Outputs

Each counter has two outputs, pwm (line_out) and pwm_n (line_compl_out) (complementary of pwm). Note that the OV, UN, and CC conditions are used to drive pwm and pwm_n, by configuring the TCPWM_CNT_TR_CTRL2 register (Table 30-2)..

Table 30-2. Configuring Output for OV, UN, and CC Conditions

Field	Bit	Value	Event	Description
CC_MATCH_MODE Default Value = 3	1:0	0	Set pwm to '1	Configures output line on a compare match (CC) event
		1	Clear pwm to '0	
		2	Invert pwm	
		3	No change	
OVERFLOW_MODE Default Value = 3	3:2	0	Set pwm to '1	Configures output line on a overflow (OV) event
		1	Clear pwm to '0	
		2	Invert pwm	
		3	No change	
UNDERFLOW_MODE Default Value = 3	5:4	0	Set pwm to '1	Configures output line on a underflow (UN) event
		1	Clear pwm to '0	
		2	Invert pwm	
		3	No change	

30.2.7 Power Modes

The TCPWM block works in Active and Sleep modes. The TCPWM block is powered from V_{CCD}. The configuration registers and other logic are powered in Deep Sleep mode to keep the states of configuration registers. See Table 30-3 for details.

Table 30-3. Power Modes in TCPWM Block

Power Mode	Block Status
CPU Active	This block is fully operational in this mode with clock running and power switched on.
CPU Sleep	The CPU is in sleep but the block is still functional in this mode. All counter clocks are on.
CPU Deep Sleep	Both power and clocks to the block are turned off, but configuration registers retain their states.
System Hibernate	In this mode, the power to this block is switched off. Configuration registers will lose their state.

30.3 Operation Modes

The counter block can function in six operational modes, as shown in Table 30-4. The MODE [26:24] field of the counter control register (TCPWM_CNTx_CTRL) configures the counter in the specific operational mode.

Table 30-4. Operational Mode Configuration

Mode	MODE Field [26:24]	Description
Timer	000	The counter increments or decrements by '1' at every clk_counter cycle in which a count event is detected. The Compare/Capture register is used to compare the count.
Capture	010	The counter increments or decrements by '1' at every clk_counter cycle in which a count event is detected. A capture event copies the counter value into the capture register.
Quadrature	011	Quadrature decoding. The counter is decremented or incremented based on two phase inputs according to an X1, X2, or X4 decoding scheme.
PWM	100	Pulse width modulation.
PWM_DT	101	Pulse width modulation with dead time insertion.
PWM_PR	110	Pseudo-random PWM using a 16- or 32-bit linear feedback shift register (LFSR) to generate pseudo-random noise.

The counter can be configured to count up, down, and up/down by setting the UP_DOWN_MODE[17:16] field in the TCPWM_CNT_CTRL register, as shown in Table 30-5.

Table 30-5. Counting Mode Configuration

Counting Modes	UP_DOWN_ MODE[17:16]	Description
UP Counting Mode	00	Increments the counter until the period value is reached. A Terminal Count (TC) and Overflow (OV) condition is generated when the counter changes from the period value.
DOWN Counting Mode	01	Decrements the counter from the period value until 0 is reached. A TC and Underflow (UN) condition is generated when the counter changes from a value of ‘0’.
UP/DOWN Counting Mode 1	10	Increments the counter until the period value is reached, and then decrements the counter until ‘0’ is reached. TC and UN conditions are generated only when the counter changes from a value of ‘0’.
UP/DOWN Counting Mode 2	11	Similar to up/down counting mode 1 but a TC condition is generated when the counter changes from ‘0’ and when the counter value changes from the period value. OV and UN conditions are generated similar to how they are generated in UP and DOWN counting modes respectively.

30.3.1 Timer Mode

The timer mode can be used to measure how long an event takes or the time difference between two events. The timer functionality increments/decrements a counter between 0 and the value stored in the PERIOD register. When the counter is running, the count value stored in the COUNTER register is compared with the compare/capture register (CC). When the counter changes from a state in which COUNTER equals CC, the cc_match event is generated.

Timer functionality is typically used for one of the following:

- Timing a specific delay – the count event is a constant '1'.
- Counting the occurrence of a specific event – the event should be connected as an input trigger and selected for the count event.

Table 30-6. Timer Mode Trigger Input Description

Trigger Inputs	Usage
Reload	Sets the counter value and starts the counter. Behavior is dependent on UP_DOWN_MODE: ■ COUNT_UP: The counter is set to "0" and count direction is set to "up". ■ COUNT_DOWN: The counter is set to PERIOD and count direction is set to "down". ■ COUNT_UPDN1/2: The counter is set to "1" and count direction is set to "up". Can be used when the counter is running or not running.
Start	Starts the counter. The counter is not initialized by hardware. The current counter value is used. Behavior is dependent on UP_DOWN_MODE. When the counter is not running: ■ COUNT_UP: The count direction is set to "up". ■ COUNT_DOWN: The count direction is set to "down". ■ COUNT_UPDN1/2: The count direction is not modified. Note that when the counter is running, the start event has no effect. Can be used when the counter is running or not running.
Stop	Stops the counter.
Count	Count event increments/decrements the counter.
Capture	Not used.

Incrementing and decrementing the counter is controlled by the count event and the counter clock clk_counter. Typical operation will use a constant '1' count event and clk_counter without pre-scaling. Advanced operations are also possible; for example, the counter event configuration can decide to count the rising edges of a synchronized input trigger.

Table 30-7. Timer Mode Supported Features

Supported Features	Description
Clock pre-scaling	Pre-scales the counter clock clk_counter.
One-shot	Counter is stopped by hardware, after a single period of the counter: ■ COUNT_UP: on an overflow event. ■ COUNT_DOWN, COUNT_UPDN1/2: on an underflow event.
Auto reload CC	CC and CC_BUFF are exchanged on a cc_match event (when specified by CTRL.AUTO_RELOAD_CC)
Up/down modes	Specified by UP_DOWN_MODE: ■ COUNT_UP: The counter counts from 0 to PERIOD. ■ COUNT_DOWN: The counter counts from PERIOD to 0. ■ COUNT_UPDN1/2: The counter counts from 1 to PERIOD and back to 0.

Table 30-6 lists the trigger outputs and the conditions when they are triggered.

Table 30-8. Timer Mode Trigger Outputs

Trigger Outputs	Description
cc_match (CC)	Counter changes from a state in which COUNTER equals CC.
Underflow (UN)	Counter is decrementing and changes from a state in which COUNTER equals “0”.
Overflow (OV)	Counter is incrementing and changes from a state in which COUNTER equals PERIOD.

Note: Each output is only two clk_sys wide and is represented by an arrow in the timing diagrams in this chapter, for example see Figure 30-7.

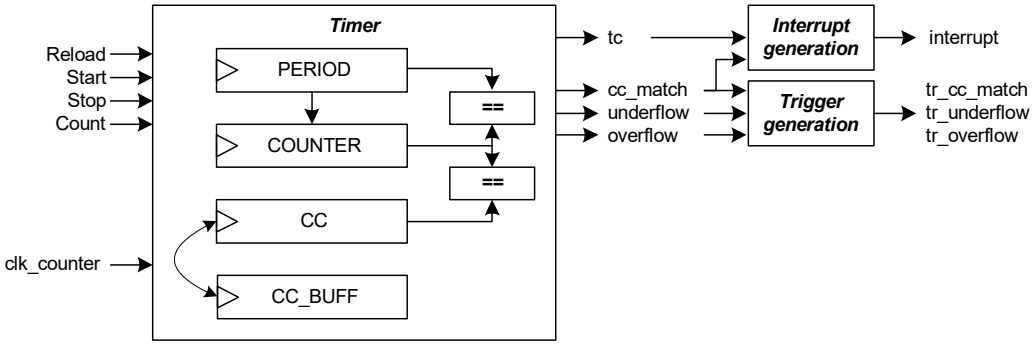
Table 30-9. Timer Mode Interrupt Outputs

Interrupt Outputs	Description
tc	Specified by UP_DOWN_MODE: ■ COUNT_UP: The tc event is the same as the overflow event. ■ COUNT_DOWN: The tc event is the same as the underflow event. ■ COUNT_UPDN1: The tc event is the same as the underflow event. ■ COUNT_UPDN2: The tc event is the same as the logical OR of the overflow and underflow events.
cc_match (CC)	Counter changes from a state in which COUNTER equals CC.

Table 30-10. Timer Mode PWM Outputs

PWM Outputs	Description
pwm	Not used.
pwm_n	Not used.

Figure 30-6. Timer Functionality



Notes:

- The timer functionality uses only PERIOD (and not PERIOD_BUFF).
- Do not write to COUNTER when the counter is running.

Figure 30-7 illustrates a timer in up-counting mode. The counter is initialized (to 0) and started with a software-based reload event.

Notes:

- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 clk_counter periods. The CC register is 2, and sets the condition for a cc_match event.
- When the counter changes from a state in which COUNTER is 4, overflow and tc events are generated.
- When the counter changes from a state in which COUNTER is 2, a cc_match event is generated.

- A constant count event of ‘1’ and clk_counter without prescaling is used in the following scenarios. If the count event is ‘0’ and a reload event is triggered, the reload will be registered only on the first clock edge when the count event is ‘1’. This means that the first clock edge when the count event is ‘1’ will not be used for counting. It will be used for reload.

Figure 30-7. Timer in Up-counting Mode

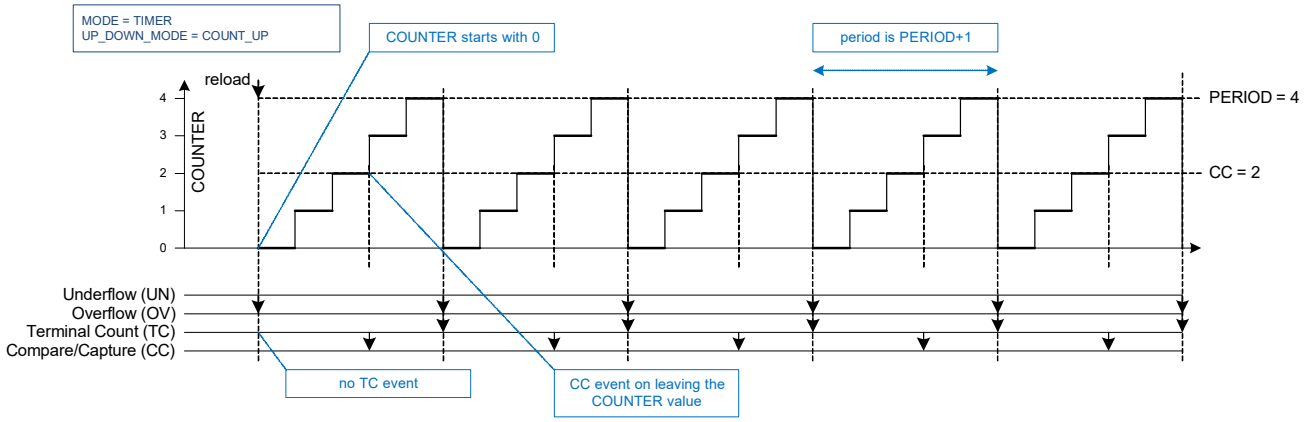


Figure 30-8 illustrates a timer in “one-shot” operation mode. Note that the counter is stopped on a tc event.

Figure 30-8. Timer in One-shot Mode

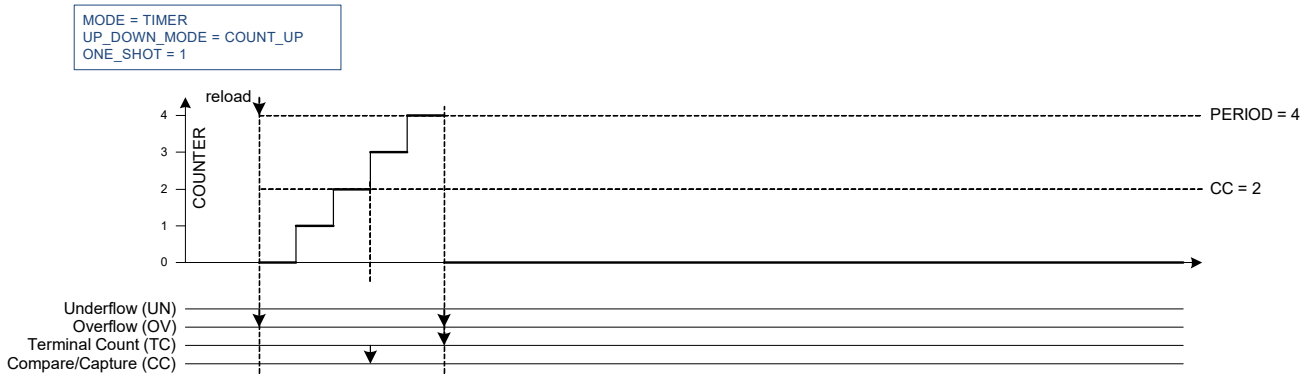


Figure 30-9 illustrates clock pre-scaling. Note that the counter is only incremented every other counter cycle.

Figure 30-9. Timer Clock Pre-scaling

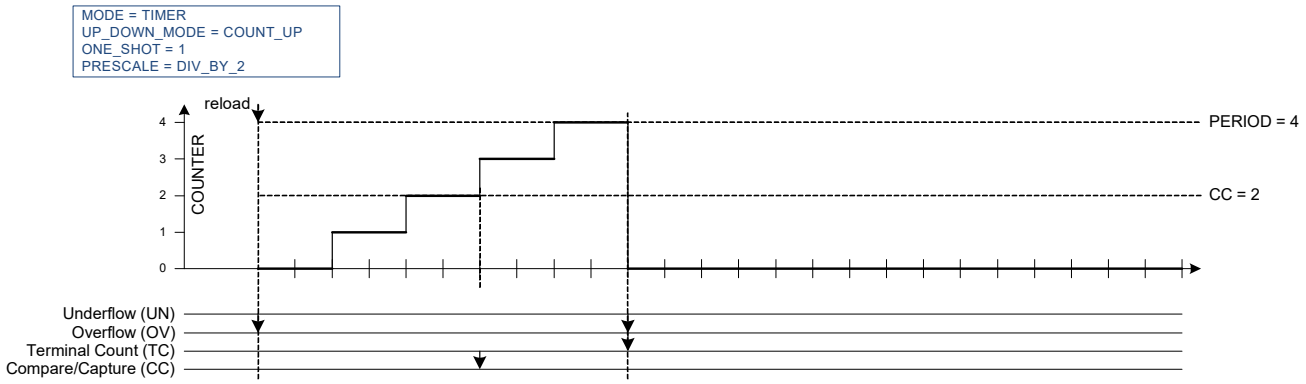


Figure 30-10 illustrates a counter that is initialized and started (reload event), stopped (stop event), and continued/started (start event). Note that the counter does not change value when it is not running (STATUS.RUNNING).

Figure 30-10. Counter Start/Stopped/Continued

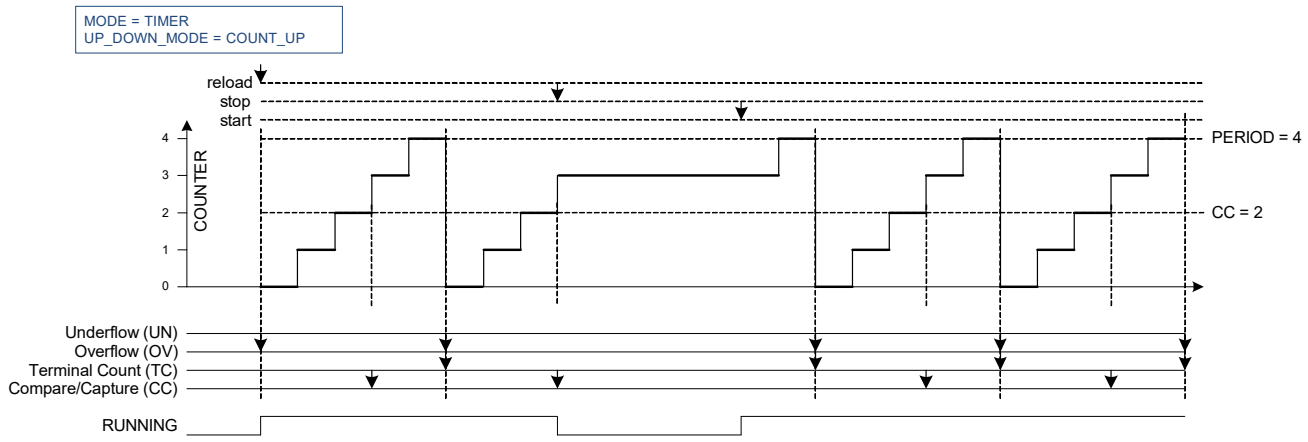


Figure 30-11 illustrates a timer that uses both CC and CC_BUFF registers. Note that CC and CC_BUFF are exchanged on a cc_match event.

Figure 30-11. Use of CC and CC_BUFF Register Bits

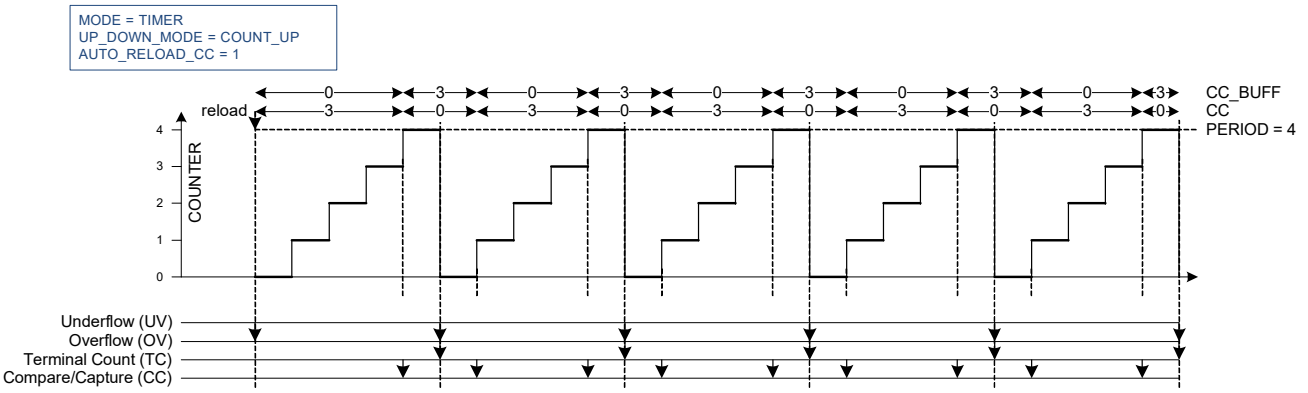


Figure 30-12 illustrates a timer in down counting mode. The counter is initialized (to PERIOD) and started with a software-based reload event.

Notes:

- When the counter changes from a state in which COUNTER is 0, a UN and TC events are generated.
- When the counter changes from a state in which COUNTER is 2, a cc_match event is generated.
- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods.

Figure 30-12. Timer in Down-counting Mode

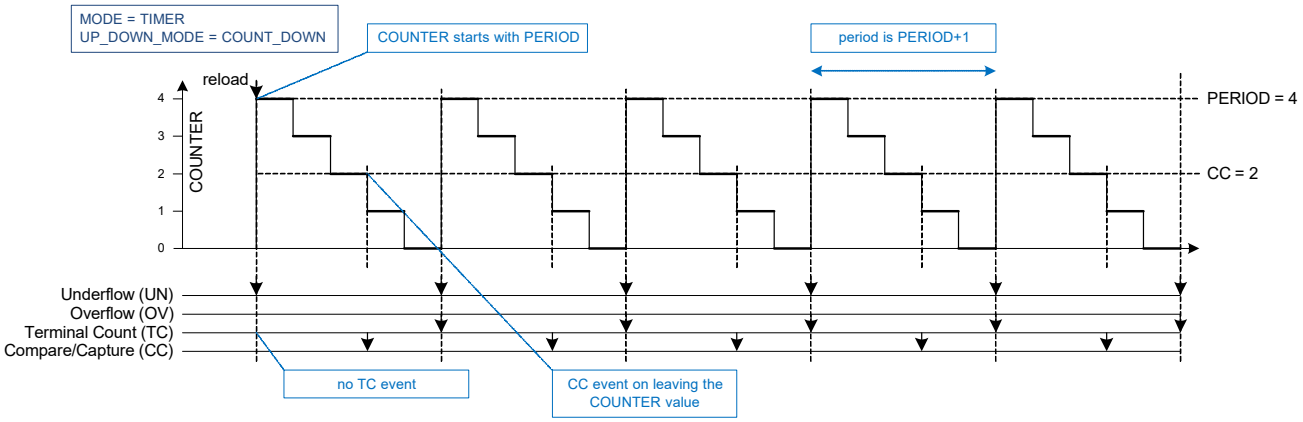


Figure 30-13 illustrates a timer in up/down counting mode 1. The counter is initialized (to 1) and started with a software-based reload event.

Notes:

- When the counter changes from a state in which COUNTER is 4, an overflow is generated.
- When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated.
- When the counter changes from a state in which COUNTER is 2, a cc_match event is generated.
- PERIOD is 4, resulting in an effective repeating counter pattern of 2*4 = 8 counter clock periods.

Figure 30-13. Timer in Up/Down Counting Mode 1

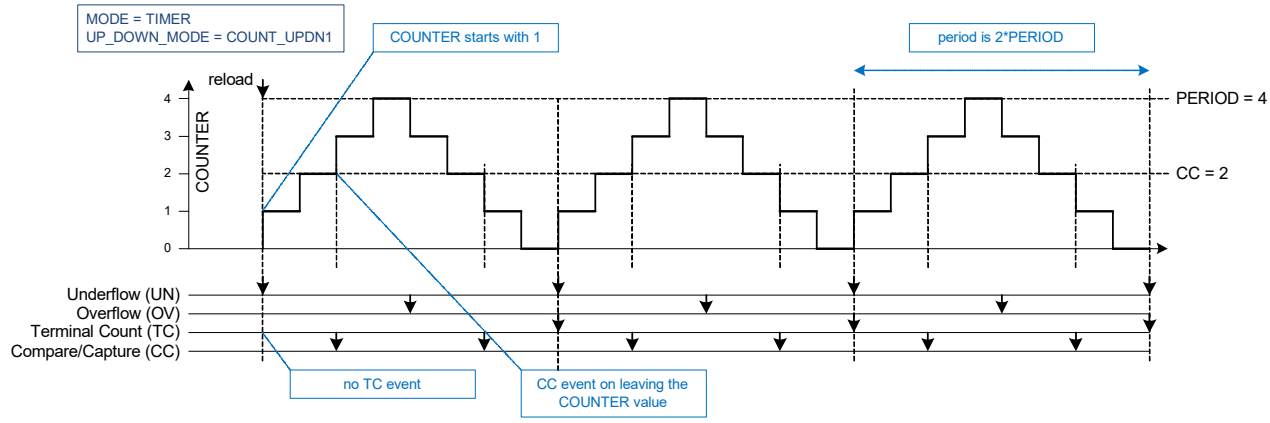


Figure 30-14 illustrates a timer in up/down counting mode 1, with different CC values.

Notes:

- When CC is 0, the cc_match event is generated at the start of the period (when the counter changes from a state in which COUNTER is 0).
- When CC is PERIOD, the cc_match event is generated at the middle of the period (when the counter changes from a state in which COUNTER is PERIOD).

Figure 30-14. Up/Down Counting Mode with Different CC Values

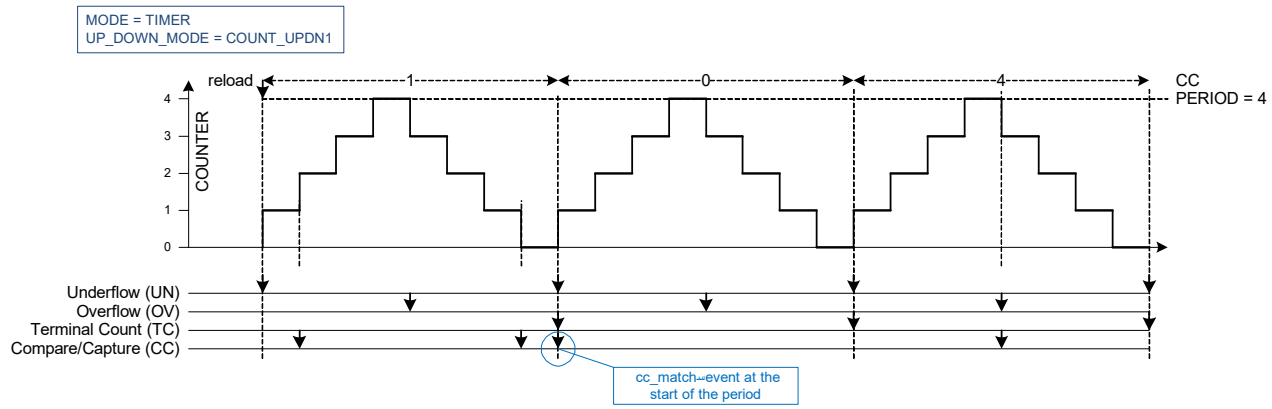
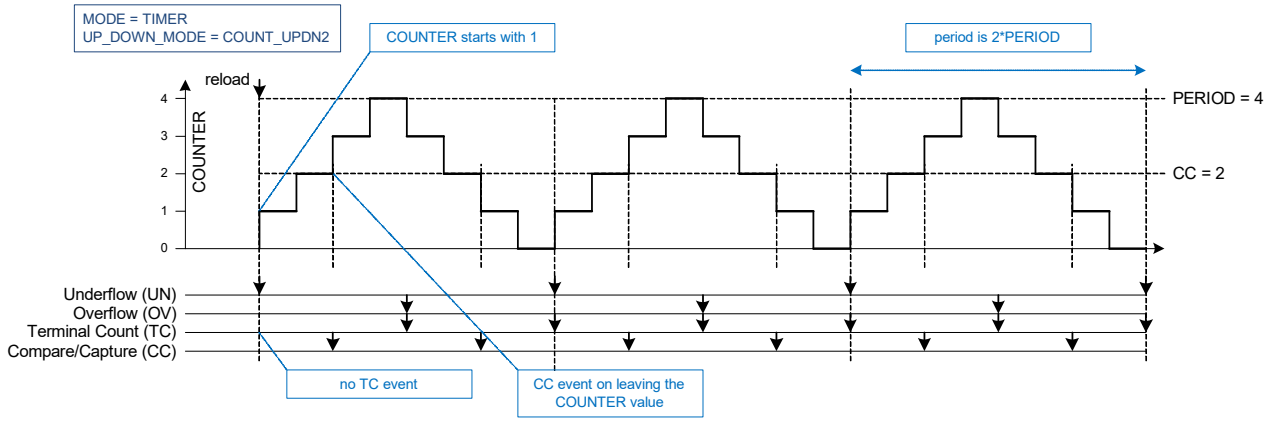


Figure 30-15 illustrates a timer in up/down counting mode 2. This mode is same as up/down counting mode 1, except for the TC event, which is generated when either underflow or overflow event occurs.

Figure 30-15. Up/Down Counting Mode 2



30.3.1.1 Configuring Counter for Timer Mode

The steps to configure the counter for Timer mode of operation and the affected register bits are as follows.

1. Disable the counter by writing ‘1’ to the TCPWM_CTRL_CLR register.
2. Select Timer mode by writing ‘000’ to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required 16- or 32-bit period in the TCPWM_CNT_PERIOD register.
4. Set the 16- or 32-bit compare value in the TCPWM_CNT_CC register and the buffer compare value in the TCPWM_CNT_CC_BUFF register.
5. Set AUTO_RELOAD_CC field of the TCPWM_CNT_CTRL register, if required to swap values at every CC condition.
6. Set clock prescaling by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register.
7. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register.
8. The timer can be configured to run either in continuous mode or one-shot mode by writing 0 or 1, respectively to the ONE_SHOT[18] field of the TCPWM_CNT_CTRL register.
9. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, stop, capture, and count).
10. Set the TCPWM_CNT_TR_CTRL1 register to select the edge of the trigger that causes the event (reload, start, stop, capture, and count).
11. If required, set the interrupt upon TC or CC condition.
12. Enable the counter by writing ‘1’ to the TCPWM_CTRL_SET register. A reload trigger must be provided through firmware (TCPWM_CMD_RELOAD register) to start the counter if the hardware reload signal is not enabled.

30.3.2 Capture Mode

The capture functionality increments and decrements a counter between 0 and PERIOD. When the capture event is activated the counter value COUNTER is copied to CC (and CC is copied to CC_BUFF).

The capture functionality can be used to measure the width of a pulse (connected as one of the input triggers and used as capture event).

The capture event can be triggered through the capture trigger input or through a firmware write to command register (TCPWM_CMD_CAPTURE).

Table 30-11. Capture Mode Trigger Input Description

Trigger Inputs	Usage
reload	Sets the counter value and starts the counter. Behavior is dependent on UP_DOWN_MODE: ■ COUNT_UP: The counter is set to “0” and count direction is set to “up”. ■ COUNT_DOWN: The counter is set to PERIOD and count direction is set to “down”. ■ COUNT_UPDN1/2: The counter is set to “1” and count direction is set to “up”. Can be used only when the counter is not running.
start	Starts the counter. The counter is not initialized by hardware. The current counter value is used. Behavior is dependent on UP_DOWN_MODE: ■ COUNT_UP: The count direction is set to “up”. ■ COUNT_DOWN: The count direction is set to “down”. ■ COUNT_UPDN1/2: The count direction is not modified. Can be used only when the counter is not running.
stop	Stops the counter.
count	Count event increments/decrements the counter.
capture	Copies the counter value to CC and copies CC to CC_BUFF.

Table 30-12. Capture Mode Supported Features

Supported Features	Description
Clock pre-scaling	Pre-scales the counter clock clk_counter.
One-shot	Counter is stopped by hardware, after a single period of the counter: ■ COUNT_UP: on an overflow event. ■ COUNT_DOWN, COUNT_UPDN1/2: on an underflow event.
Up/down modes	Specified by UP_DOWN_MODE: ■ COUNT_UP: The counter counts from 0 to PERIOD. ■ COUNT_DOWN: The counter counts from PERIOD to 0. ■ COUNT_UPDN1/2: The counter counts from 1 to PERIOD and back to 0.

Table 30-13. Capture Mode Trigger Output Description

Trigger Outputs	Description
cc_match (CC)	CC is copied to CC_BUFF and counter value is copied to CC (cc_match equals capture event).
Underflow (UN)	Counter is decrementing and changes from a state in which COUNTER equals “0”.
Overflow (OV)	Counter is incrementing and changes from a state in which COUNTER equals PERIOD.

Table 30-14. Capture Mode Interrupt Outputs

Interrupt Outputs	Description
tc	Specified by UP_DOWN_MODE: ■ COUNT_UP: tc event is the same as the overflow event. ■ COUNT_DOWN: tc event is the same as the underflow event. ■ COUNT_UPDN1: tc event is the same as the underflow event. ■ COUNT_UPDN2: tc event is the same as the logical OR of the overflow and underflow events.
cc_match (CC)	CC is copied to CC_BUFF and counter value is copied to CC (cc_match equals capture event).

Table 30-15. Capture Mode PWM Outputs

PWM Outputs	Description
pwm	Not used.
pwm_n	Not used.

Figure 30-16. Capture Functionality

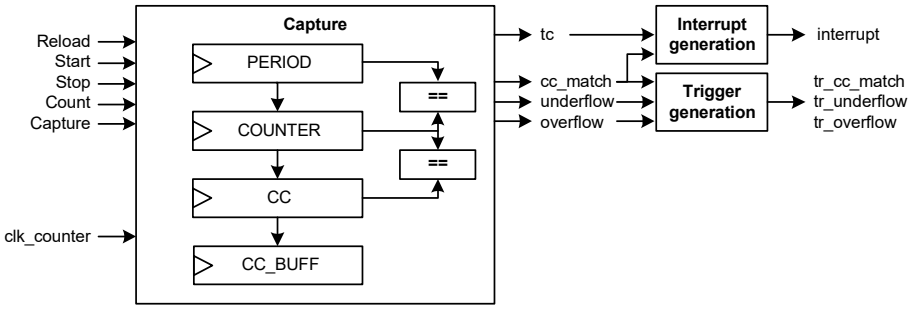
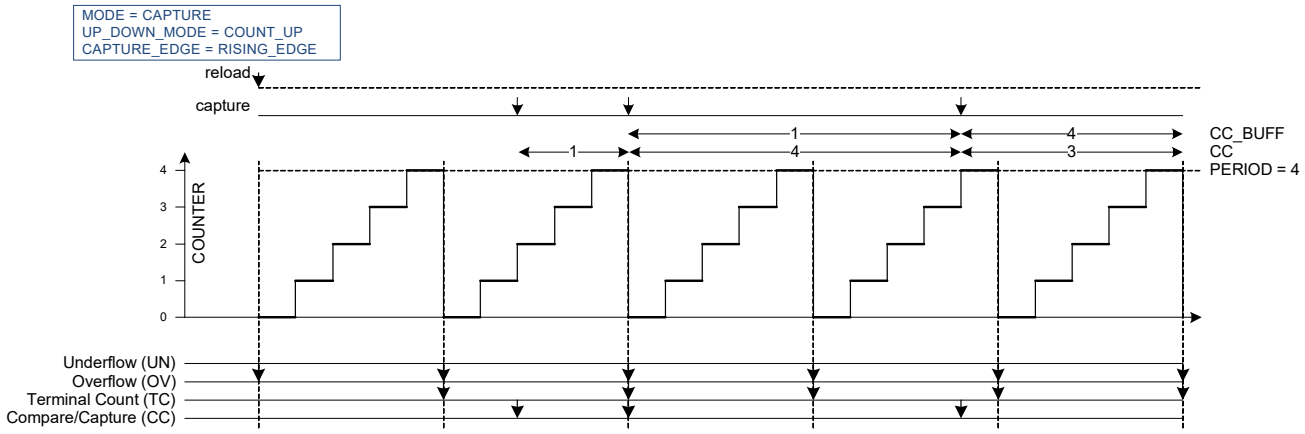


Figure 30-17 illustrates capture behavior in the up counting mode.

Notes:

- The capture event detection uses rising edge detection. As a result, the capture event is remembered until the next “active count” pre-scaled counter clock.
- When a capture event occurs, COUNTER is copied into CC. CC is copied to CC_BUFF.
- A cc_match event is generated when the counter value is captured.

Figure 30-17. Capture in Up Counting Mode

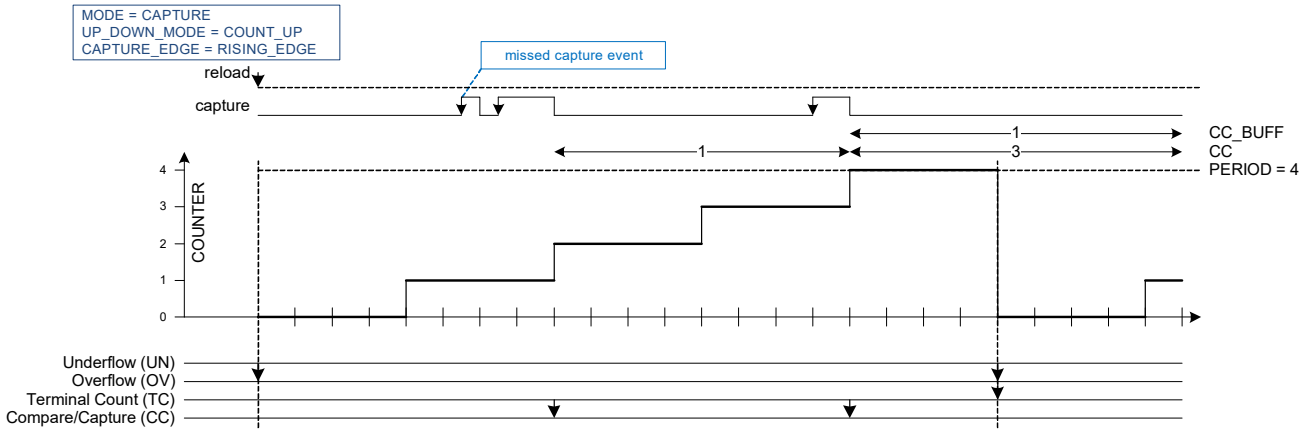


When multiple capture events are detected before the next “active count” pre-scaled counter clock, capture events are treated as follows:

- In the rising edge and falling edge modes, multiple events are effectively reduced to a single event.
- In the rising/falling edge mode, an even number of events is not detected and an odd number of events is reduced to a single event.

This behavior is illustrated by Figure 30-18, in which a pre-scaler by a factor of 4 is used.

Figure 30-18. Multiple Events Detected before Active-Count



30.3.2.1 Configuring Counter for Capture Mode

The steps to configure the counter for Capture mode operation and the affected register bits are as follows.

1. Disable the counter by writing ‘1’ to the TCPWM_CTRL_CLR register.
2. Select Capture mode by writing ‘010’ to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required 16-bit period in the TCPWM_CNT_PERIOD register.
4. Set clock prescaling by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register.
5. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register.
6. Counter can be configured to run either in continuous mode or one-shot mode by writing 0 or 1, respectively to the ONE_-SHOT[18] field of the TCPWM_CNT_CTRL register.
7. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, stop, capture, and count).
8. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (reload, start, stop, capture, and count).
9. If required, set the interrupt upon TC or CC condition.
10. Enable the counter by writing ‘1’ to the TCPWM_CTRL_SET register. A reload trigger must be provided through firmware (TCPWM_CMD_RELOAD register) to start the counter if the hardware reload signal is not enabled.

30.3.3 Quadrature Decoder Mode

Quadrature functionality increments and decrements a counter between 0 and 0xFFFF or 0xFFFFFFFF (32-bit mode). Counter updates are under control of quadrature signal inputs: index, phiA, and phiB. The index input is used to indicate an absolute position. The phiA and phiB inputs are used to determine a change in position (the rate of change in position can be used to derive speed). The quadrature inputs are mapped onto triggers (as described in Table 30-16).

Table 30-16. Quadrature Mode Trigger Input Description

Trigger Input	Usage
reload/index	This event acts as a quadrature index input. It initializes the counter to the counter midpoint 0x8000 (16-bit) or 0x8000000 (32-bit mode) and starts the quadrature functionality. Rising edge event detection or falling edge detection mode must be used.
start/phiB	This event acts as a quadrature phiB input. Pass through (no edge detection) event detection mode must be used.
stop	Stops the quadrature functionality.
count/phiA	This event acts as a quadrature phiA input. Pass through (no edge detection) event detection mode must be used.
capture	Not used.

Table 30-17. Quadrature Mode Supported Features

Supported Features	Description
Quadrature encoding	Three encoding schemes for the phiA and phiB inputs are supported (as specified by CTRL.QUADRATURE_MODE): X1 encoding. X2 encoding. X4 encoding.

Note: Clock pre-scaling is not supported and the count event is used as a quadrature input phiA. As a result, the quadrature functionality operates on the counter clock (clk_counter), rather than on an “active count” prescaled counter clock.

Table 30-18. Quadrature Mode Trigger Output Description

Trigger Outputs	Description
cc_match (CC)	Counter value COUNTER equals 0 or 0xFFFF or 0xFFFFFFFF (32-bit mode) or a reload/index event.
Underflow (UN)	Not used.
Overflow (OV)	Not used.

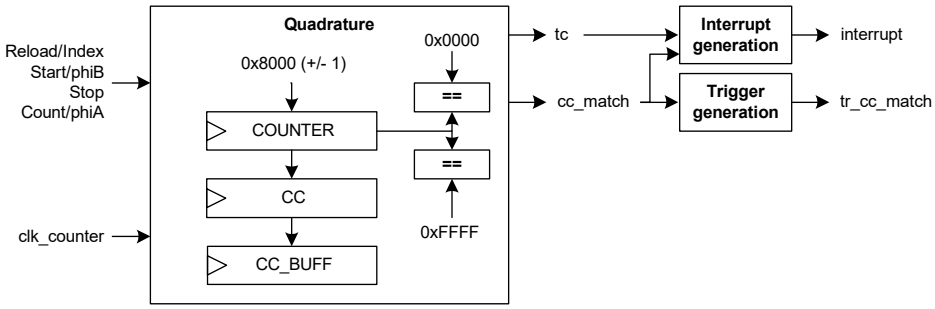
Table 30-19. Quadrature Mode Interrupt Outputs

Interrupt Outputs	Description
cc_match (CC)	Counter value COUNTER equals 0 or 0xFFFF or 0xFFFFFFFF (32-bit mode) or a reload/index event.
tc	Reload/index event.

Table 30-20. Quadrature Mode PWM Outputs

PWM Outputs	Description
pwm	Not used.
pwm_n	Not used.

Figure 30-19. Quadrature Functionality (16-bit example)



Quadrature functionality is described as follows:

- A software-generated reload event starts quadrature operation. As a result, COUNTER is set to 0x8000 (16-bit) or 0x80000000 (32-bit), which is the counter midpoint (the COUNTER is set to 0x7FFF or 0x7FFFFFFF if the reload event coincides with a decrement event; the COUNTER is set to 0x8001 or 0x80000001 if the reload event coincides with an increment event). Note that a software-generated reload event is generated only once, when the counter is not running. All other reload/index events are hardware-generated reload events as a result of the quadrature index signal.
- During quadrature operation:
  - The counter value COUNTER is incremented or decremented based on the specified quadrature encoding scheme.
  - On a reload/index event, CC is copied to CC_BUFF, COUNTER is copied to CC, and COUNTER is set to 0x8000/0x80000000. In addition, the tc and cc_match events are generated.
  - When the counter value COUNTER is 0x0000, CC is copied to CC_BUFF, COUNTER (0x0000) is copied to CC, and COUNTER is set to 0x8000/0x80000000. In addition, the cc_match event is generated.
  - When the counter value COUNTER is 0xFFFF/0xFFFFFFFF, CC is copied to CC_BUFF, COUNTER (0xFFFF/0xFFFFFFFF) is copied to CC, and COUNTER is set to 0x8000/0x80000000. In addition, the cc_match event is generated.

Note: When the counter reaches 0x0000 or 0xFFFF/0xFFFFFFFF, the counter is automatically set to 0x8000/0x80000000 without an increase or decrease event.

The software interrupt handler uses the tc and cc_match interrupt cause fields to distinguish between a reload/index event and a situation in which a minimum/maximum counter value was reached (about to wrap around). The CC and CC_BUFF registers are used to determine when the interrupt causing event occurred.

Note that a counter increment/decrement can coincide with a reload/index/tc event or with a situation cc_match event. Under these circumstances, the counter value set to either 0x8000+1 or 0x80000000+1 (increment) or 0x8000–1 or 0x80000000–1 (decrement).

Counter increments (incr1 event) and decrements (decr1 event) are determined by the quadrature encoding scheme as illustrated by Figure 30-20.

Figure 30-20. Quadrature Mode Waveforms

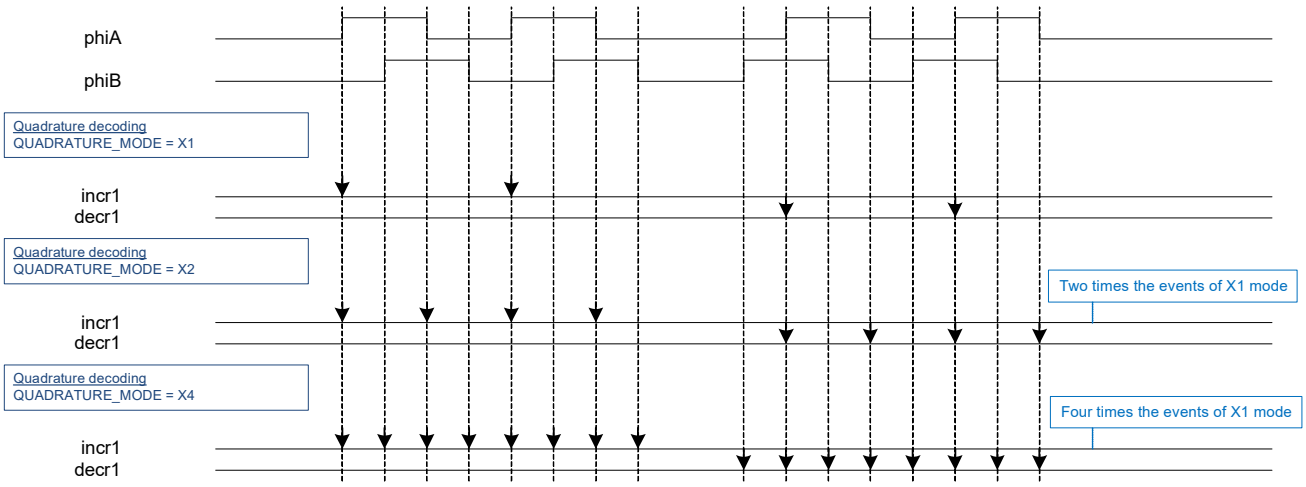


Figure 30-21 illustrates quadrature functionality as a function of the reload/index, incr1, and decr1 events. Note that the first reload/index event copies the counter value COUNTER to CC.

Figure 30-21. Quadrature Mode Reload/Index Timing

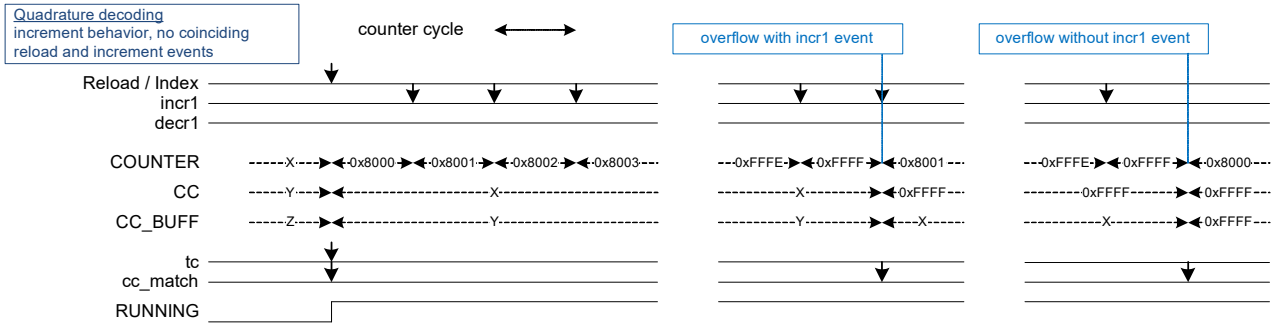
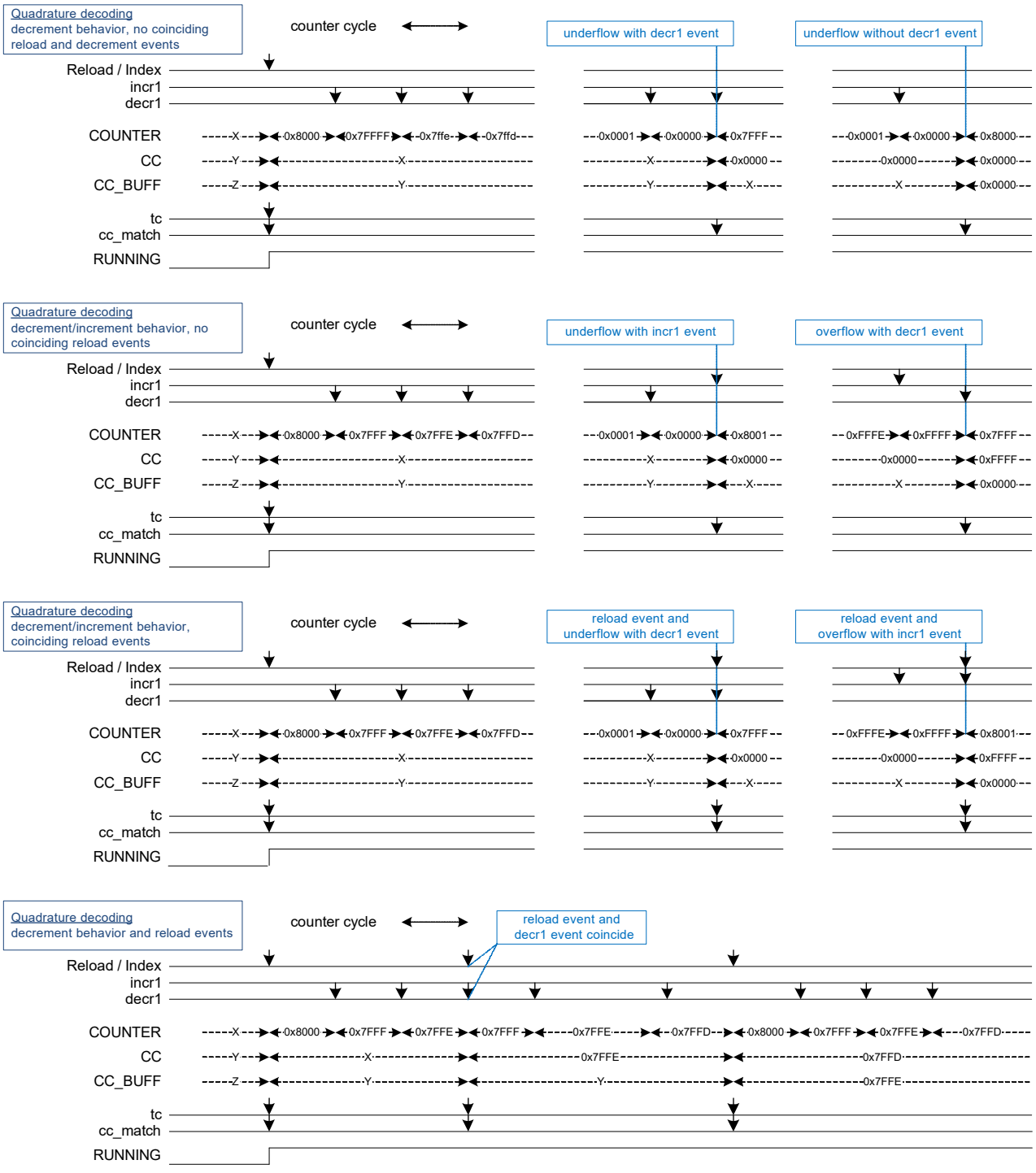


Figure 30-22 illustrate quadrature functionality for different event scenarios (including scenarios with coinciding events). In all scenarios, the first reload/index event is generated by software when the counter is not yet running.

Figure 30-22. Quadrature Mode Timing Cases



30.3.3.1 Configuring Counter for Quadrature Mode

The steps to configure the counter for quadrature mode of operation and the affected register bits are as follows.

1. Disable the counter by writing ‘1’ to the TCPWM_CTRL_CLR register.
2. Select Quadrature mode by writing ‘011’ to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required encoding mode by writing to the QUADRATURE_MODE[21:20] field of the TCPWM_CNT_CTRL register.
4. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (Index and Stop).
5. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (Index and Stop).
6. If required, set the interrupt upon TC or CC condition.
7. Enable the counter by writing ‘1’ to the TCPWM_CTRL_SET register. A reload trigger must be provided through firmware (TCPWM_CMD_RELOAD register) to start the counter if the hardware reload signal is not enabled.

30.3.4 Pulse Width Modulation Mode

The PWM can output left, right, center, or asymmetrically-aligned PWM. The PWM signal is generated by incrementing or decrementing a counter between 0 and PERIOD, and comparing the counter value COUNTER with CC. When COUNTER equals CC, the cc_match event is generated. The pulse-width modulated signal is then generated by using the cc_match event along with overflow and underflow events. Two pulse-width modulated signals “pwm” and “pwm_n” are output from the PWM.

Table 30-21. PWM Mode Trigger Input Description

Trigger Inputs	Usage
reload	Sets the counter value and starts the counter. Behavior is dependent on UP_DOWN_MODE: ■ COUNT_UP: The counter is set to “0” and count direction is set to “up”. ■ COUNT_DOWN: The counter is set to PERIOD and count direction is set to “down”. ■ COUNT_UPDN1/2: The counter is set to “1” and count direction is set to “up”. Can be used only when the counter is not running.
start	Starts the counter. The counter is not initialized by hardware. The current counter value is used. Behavior is dependent on UP_DOWN_MODE: ■ COUNT_UP: The count direction is set to “up”. ■ COUNT_DOWN: The count direction is set to “down”. ■ COUNT_UPDN1/2: The count direction is set to “up”. Can be used only when the counter is not running.
stop/kill	Stops the counter or suppresses the PWM output, depending on PWM_STOP_ON_KILL and PWM_SYNC_KILL.
count	Count event increments/decrements the counter.
capture/swap	This event acts as a swap event. When this event is active, the CC/CC_BUFF and PERIOD/PERIOD_BUFF registers are exchanged on a tc event (when specified by CTRL.AUTO_RELOAD_CC and CTRL.AUTO_RELOAD_PERIOD). A swap event requires rising, falling, or rising/falling edge event detection mode. Pass-through mode is not supported, unless the selected event is a constant '0' or '1'. **Note:** When COUNT_UPDN2 mode exchanges PERIOD and PERIOD_BUFF at a tc event that coincides with an OV event, software should ensure that the PERIOD and PERIOD_BUFF values are the same. When a swap event is detected and the counter is running, the event is kept pending until the next tc event. When a swap event is detected and the counter is not running, the event is cleared by hardware.

Table 30-22. PWM Mode Supported Features

Supported Features	Description
Clock pre-scaling	Pre-scales the counter clock “clk_counter”.
One-shot	Counter is stopped by hardware, after a single period of the counter: ■ COUNT_UP: on an overflow event. ■ COUNT_DOWN and COUNT_UPDN1/2: on an underflow event.
Compare Swap	CC and CC_BUFF are exchanged on a swap event and tc event (when specified by CTRL.AUTO_RELOAD_CC).
Period Swap	PERIOD and PERIOD_BUFF are exchanged on a swap event and tc event (when specified by CTRL.AUTO_RELOAD_PERIOD). **Note:** When COUNT_UPDN2/Asymmetric mode exchanges PERIOD and PERIOD_BUFF at a tc event that coincides with an overflow event, software should ensure that the PERIOD and PERIOD_BUFF values are the same.
Alignment (Up/Down modes)	Specified by UP_DOWN_MODE: ■ COUNT_UP: The counter counts from 0 to PERIOD. Generates a left-aligned PWM output. ■ COUNT_DOWN: The counter counts from PERIOD to 0. Generates a right-aligned PWM output. ■ COUNT_UPDN1/2: The counter counts from 1 to PERIOD and back to 0. Generates a center-aligned/ asymmetric PWM output.
Kill modes	Specified by PWM_STOP_ON_KILL and PWM_SYNC_KILL: ■ PWM_STOP_ON_KILL = ‘1’ (PWM_SYNC_KILL = don’t care): Stop on Kill mode. This mode stops the counter on a stop/kill event. Reload or start event is required to restart counting. ■ PWM_STOP_ON_KILL = ‘0’ and PWM_SYNC_KILL = ‘0’: Asynchronous kill mode. This mode keeps the counter running, but suppresses the PWM output signals and continues to do so for the duration of the stop/kill event. ■ PWM_STOP_ON_KILL = ‘0’ and PWM_SYNC_KILL = ‘1’: Synchronous kill mode. This mode keeps the counter running, but suppresses the PWM output signals and continues to do so until the next tc event without a stop/kill event.

Note that the PWM mode does not support dead time insertion. This functionality is supported by the separate PWM_DT mode.

Table 30-23. PWM Mode Trigger Output Description

Trigger Output	Description
cc_match (CC)	Specified by UP_DOWN_MODE: ■ COUNT_UP and COUNT_DOWN: The counter changes to a state in which COUNTER equals CC. ■ COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC.
Underflow (UN)	Counter is decrementing and changes from a state in which COUNTER equals “0”.
Overflow (OV)	Counter is incrementing and changes from a state in which COUNTER equals PERIOD.

Table 30-24. PWM Mode Interrupt Output Description

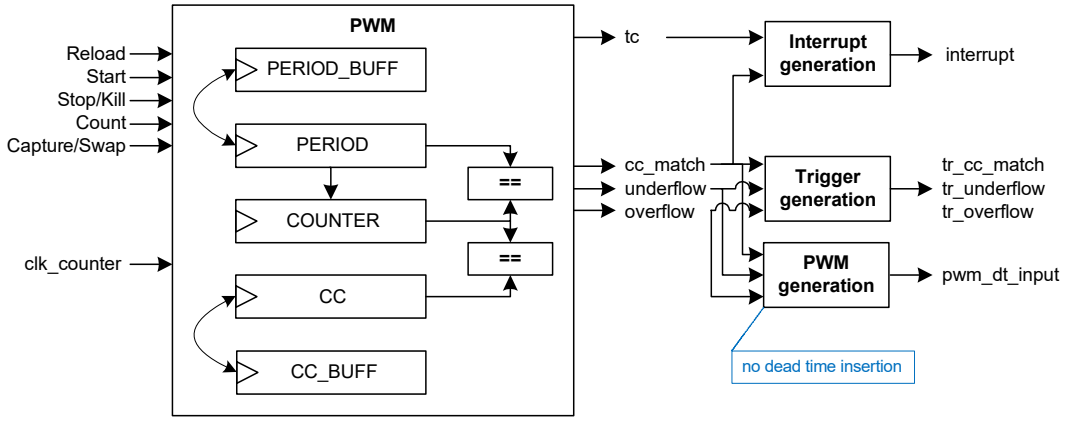
Interrupt Outputs	Description
tc	Specified by UP_DOWN_MODE: ■ COUNT_UP: tc event is the same as the overflow event. ■ COUNT_DOWN: tc event is the same as the underflow event. ■ COUNT_UPDN1: tc event is the same as the underflow event. ■ COUNT_UPDN2: tc event is the same as the logical OR of the overflow and underflow events.
cc_match (CC)	Specified by UP_DOWN_MODE: ■ COUNT_UP and COUNT_DOWN: The counter changes to a state in which COUNTER equals CC. ■ COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC.

Table 30-25. PWM Mode PWM Outputs

PWM Outputs	Description
pwm	PWM output.
pwm_n	Complementary PWM output.

Note that the cc_match event generation in COUNT_UP and COUNT_DOWN modes are different from the generation in other functional modes or counting modes. This is to ensure that 0 percent and 100 percent duty cycles can be generated.

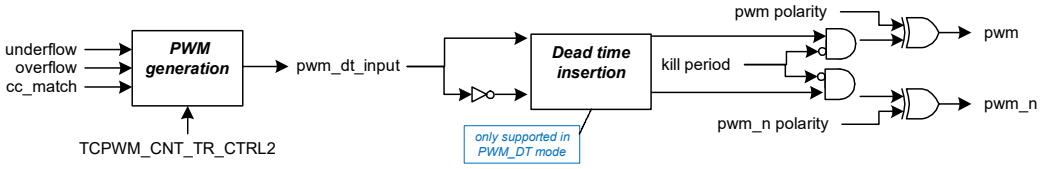
Figure 30-23. PWM Mode Functionality



The generation of PWM output signals is a multi-step process and is illustrated in Figure 30-24. The PWM output signals are generated by using the underflow, overflow, and cc_match events. Each of these events can be individually set to INVERT, SET, or CLEAR pwm_dt_input.

Note: An underflow and cc_match or an overflow and cc_match can occur at the same time. When this happens, underflow and overflow events take priority over cc_match. For example, if overflow = SET and cc_match = CLEAR then pwm_dt_input will be SET to '1' first and then CLEARED to '0' immediately after. This can be seen in Figure 30-26.

Figure 30-24. PWM Output Generation



PWM polarity and PWM_n polarity as seen in Figure 30-24, allow the PWM outputs to be inverted. PWM polarity is controlled through CTRL.QUADRATURE_MODE[0] and PWM_n polarity is controlled through CTRL.QUADRATURE_MODE[1].

PWM behavior depends on the PERIOD and CC registers. The software can update the PERIOD_BUFF and CC_BUFF registers, without affecting the PWM behavior. This is the main rationale for double buffering these registers.

Figure 30-25 illustrates a PWM in up counting mode. The counter is initialized (to 0) and started with a software-based reload event.

Notes:

- When the counter changes from a state in which COUNTER is 4, an overflow and tc event are generated.
- When the counter changes to a state in which COUNTER is 2, a cc_match event is generated.
- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods.

Figure 30-25. PWM in Up Counting Mode

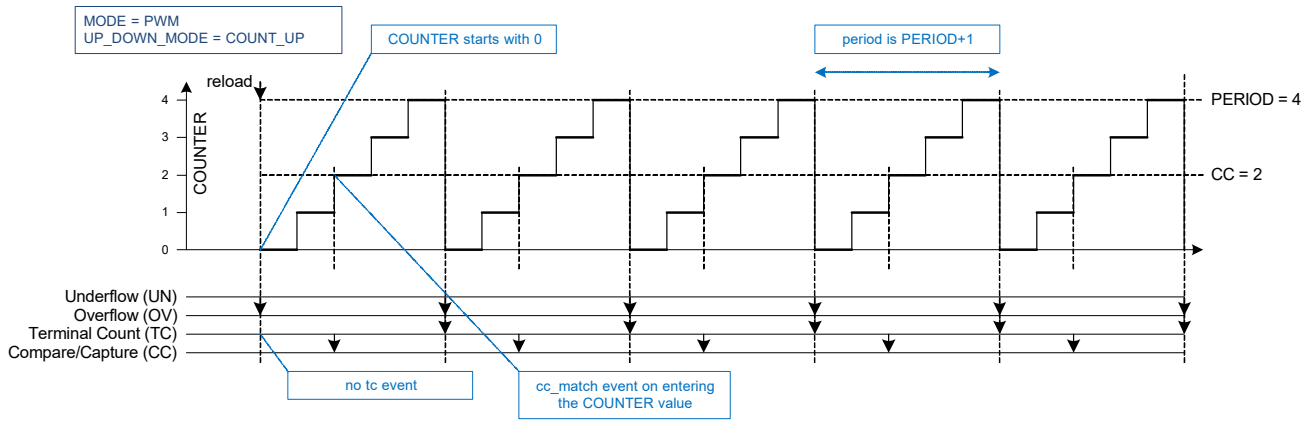


Figure 30-26 illustrates a PWM in up counting mode generating a left-aligned PWM. The figure also illustrates how a right-aligned PWM can be created using the PWM in up counting mode by inverting the OVERFLOW_MODE and CC_MATCH_MODE and using a CC value that is complementary (PERIOD+1 - pulse width) to the one used for left-aligned PWM. Note that CC is changed (to CC_BUFF, which is not depicted) on a tc event. The duty cycle is controlled by setting the CC value. CC = desired duty cycle x (PERIOD+1).

Figure 30-26. PWM Left- and Right-Aligned Outputs

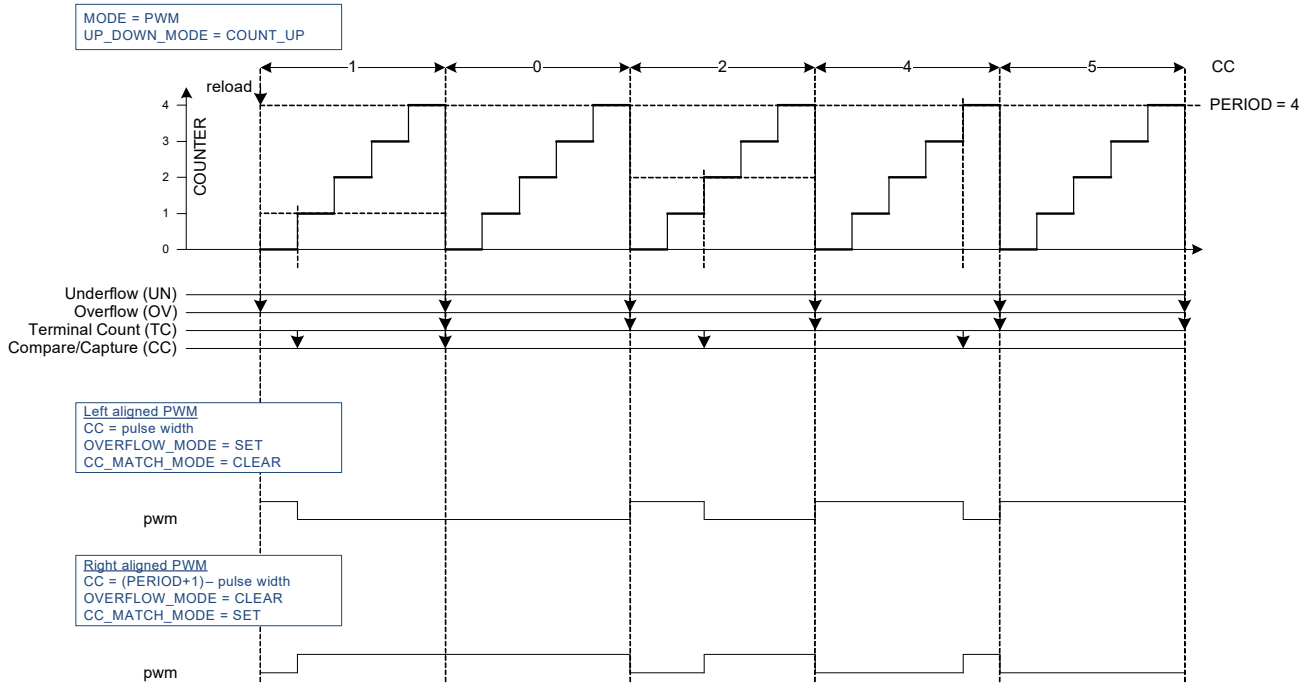


Figure 30-27 illustrates a PWM in down counting mode. The counter is initialized (to PERIOD) and started with a software-based reload event.

Notes:

- When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated.
- When the counter changes to a state in which COUNTER is 2, a cc_match event is generated.
- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods.

Figure 30-27. PWM in Down Counting Mode

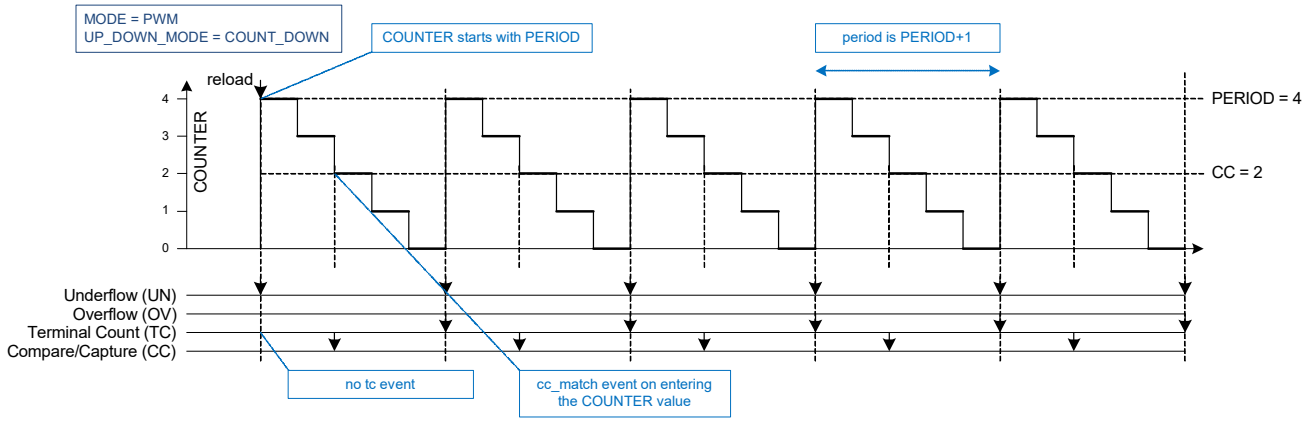


Figure 30-28 illustrates a PWM in down counting mode with different CC values. The figure also illustrates how a right-aligned PWM can be creating using the PWM in down counting mode. Note that the CC is changed (to CC_BUFF, which is not depicted) on a tc event.

Figure 30-28. Right- and Left-Aligned Down Counting PWM

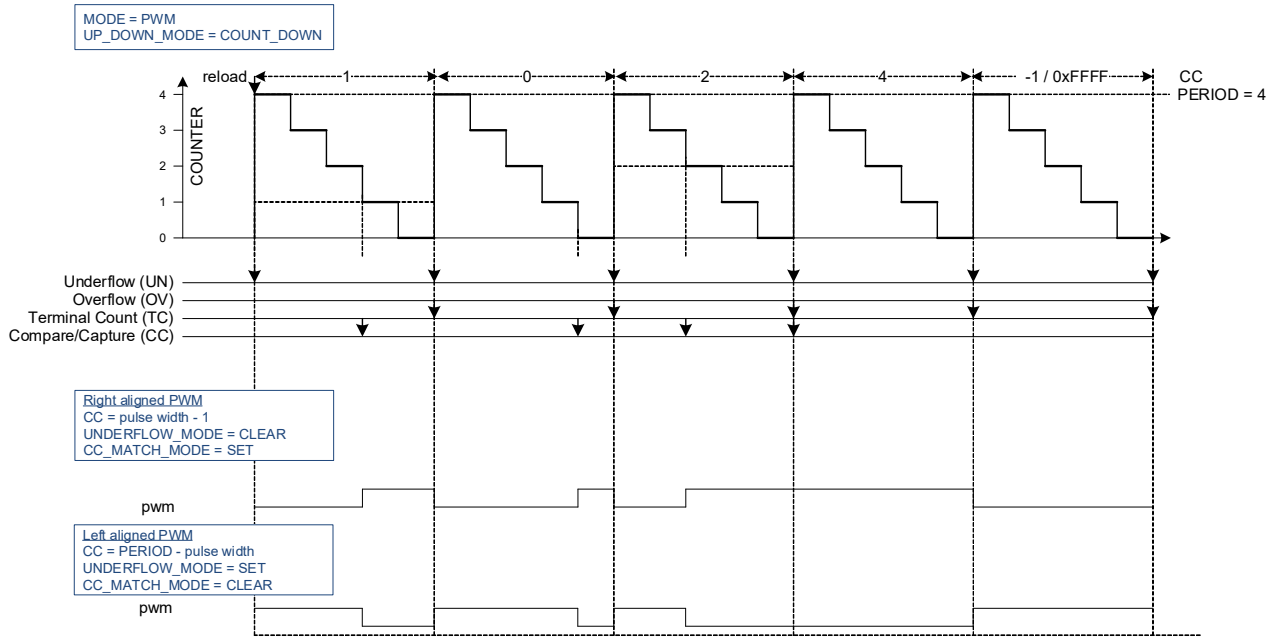


Figure 30-29 illustrates a PWM in up/down counting mode. The counter is initialized (to 1) and started with a software-based reload event.

Notes:

- When the counter changes from a state in which COUNTER is 4, an overflow is generated.
- When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated.
- When the counter changes from a state in which COUNTER is 2, a cc_match event is generated. Note that the actual counter value COUNTER from before the reload event is NOT used, instead the counter value before the reload event is considered to be 0.
- PERIOD is 4, resulting in an effective repeating counter pattern of 2*4 = 8 counter clock periods.

Figure 30-29. Up/Down Counting PWM

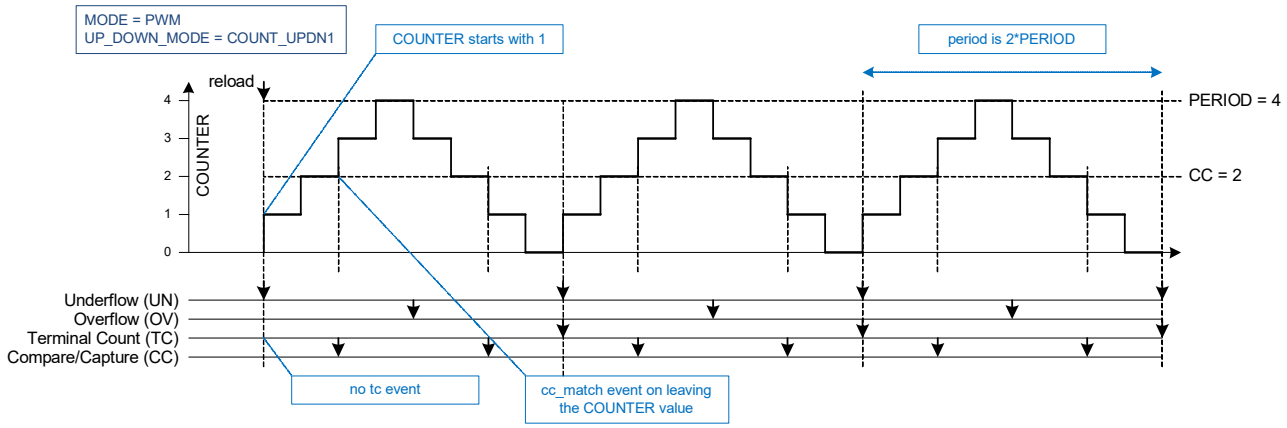
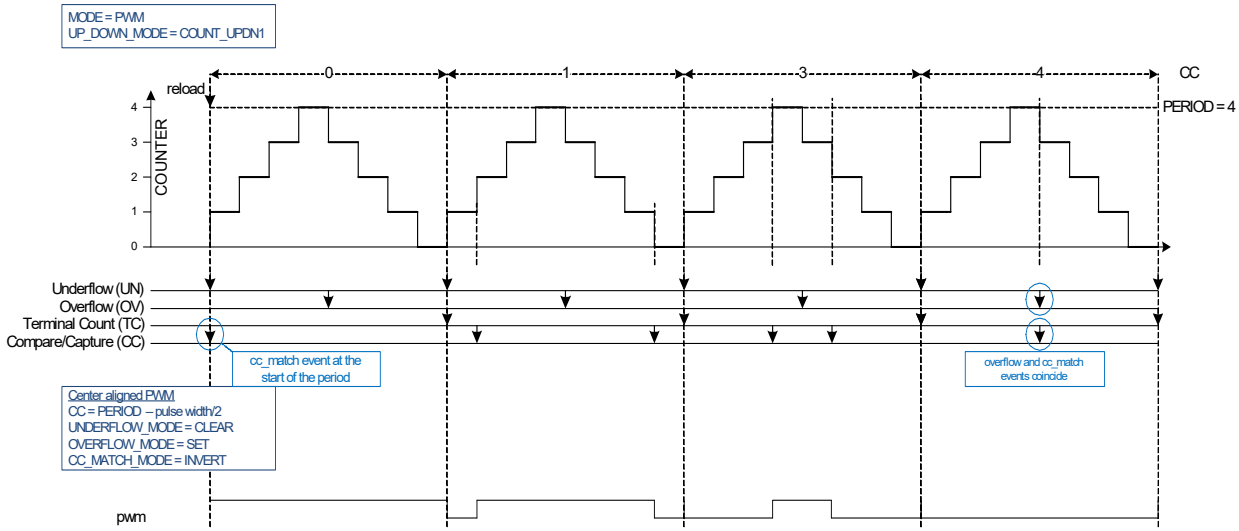


Figure 30-30 illustrates a PWM in up/down counting mode with different CC values. The figure also illustrates how a center-aligned PWM can be creating using the PWM in up/down counting mode.

Note:

- The actual counter value COUNTER from before the reload event is NOT used. Instead the counter value before the reload event is considered to be 0. As a result, when the first CC value at the reload event is 0, a cc_match event is generated.
- CC is changed (to CC_BUFF, which is not depicted) on a tc event.

Figure 30-30. Up/Down Counting Center-Aligned PWM



Different stop/kill modes exist. The mode is specified by PWM_STOP_ON_KILL and PWM_SYNC_KILL.

The following three modes are supported:

- PWM_STOP_ON_KILL is '1' (PWM_SYNC_KILL is don't care): Stop on Kill mode. This mode stops the counter on a stop/kill event. Reload or start event is required to restart the counter. Both software and external trigger input can be selected as stop kill. Edge detection mode is required.
- PWM_STOP_ON_KILL is '0' and PWM_SYNC_KILL is '0': Asynchronous Kill mode. This mode keeps the counter running, but suppresses the PWM output signals synchronously on the next count clock ("active count" pre-scaled clk_counter) and continues to do so for the duration of the stop/kill event. Only the external trigger input can be selected as asynchronous kill. Pass through detection mode is required.

- PWM_STOP_ON_KILL is ‘0’ and PWM_SYNC_KILL is ‘1’: Synchronous Kill mode. This mode keeps the counter running, but suppresses the PWM output signals synchronously on the next count clock (“active count” pre-scaled clk_counter) and continues to do so until the next tc event without a stop/kill event. Only the external trigger input can be selected as synchronous kill. Rising edge detection mode is required.

Figure 30-31, Figure 30-32. and Figure 30-33 illustrate the above three modes.

Figure 30-31. PWM Stop on Kill

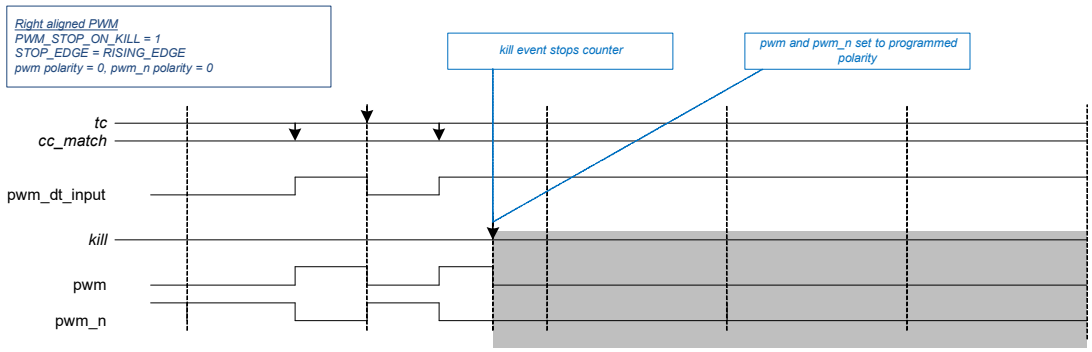


Figure 30-32. PWM Async Kill

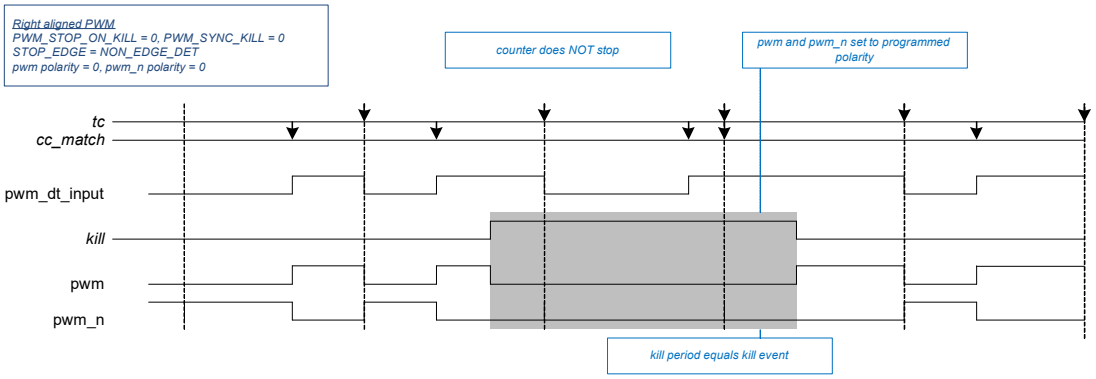


Figure 30-33. PWM Sync Kill

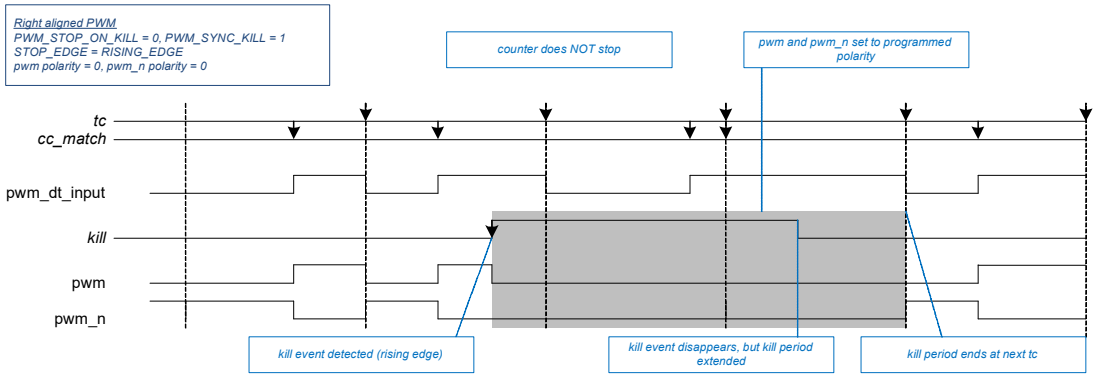
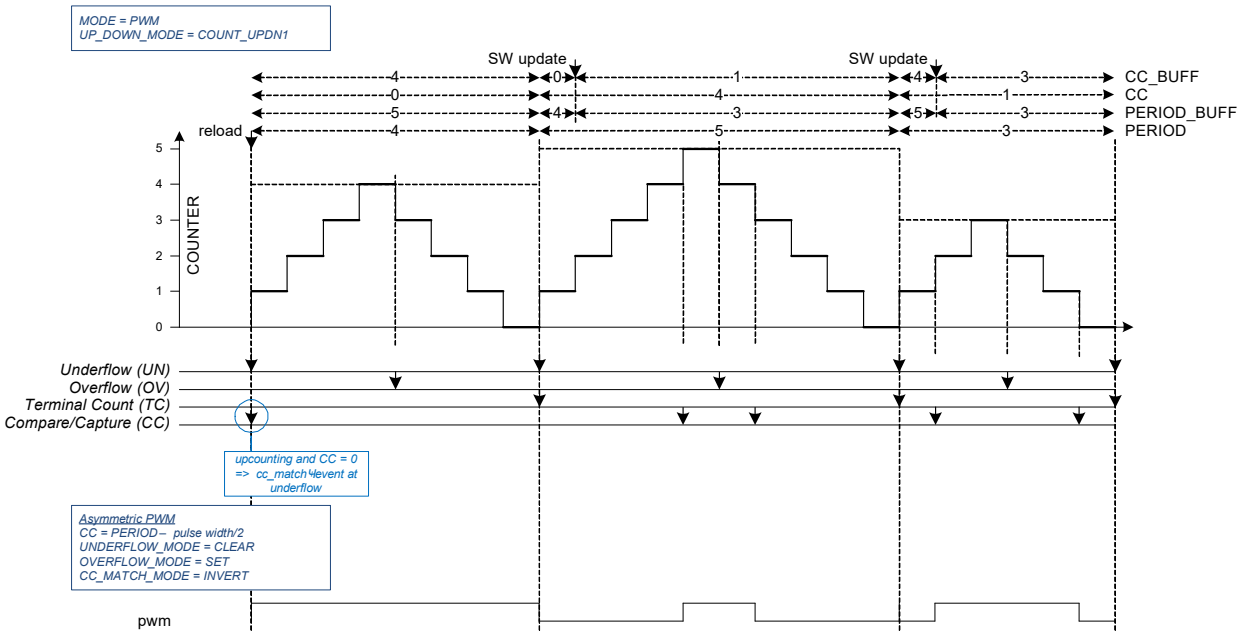


Figure 30-34 illustrates center-aligned PWM with PERIOD/PERIOD_BUFF and CC/CC_BUFF registers (up/down counting mode 1). At the TC condition, the PERIOD and CC registers are automatically exchanged with the PERIOD_BUFF and CC_BUFF registers. The swap event is generated by hardware trigger 1, which is a constant ‘1’ and therefore always active at the TC condition. After the hardware exchange, the software handler on the tc interrupt updates PERIOD_BUFF and CC_BUFF.

Figure 30-34. PWM Mode CC Swap Event



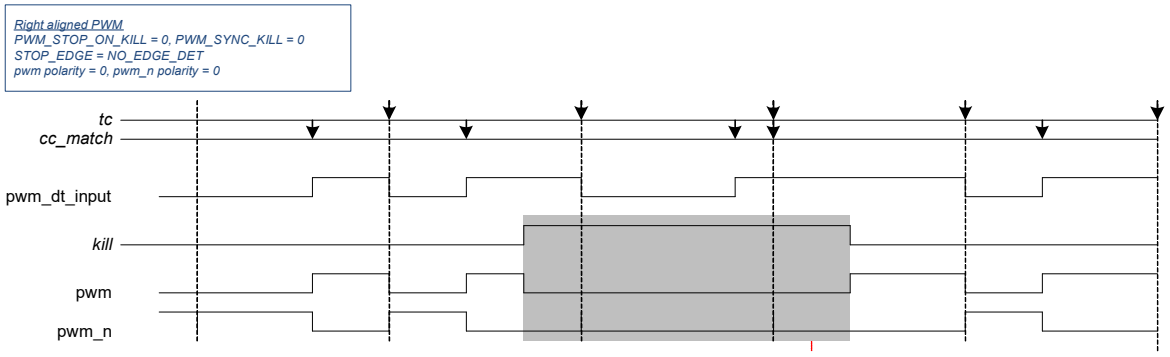
The PERIOD swaps with PERIOD_BUFF on a terminal count. The CC swaps with CC_BUFF on a terminal count. Software can then update PERIOD_BUFF and CC_BUFF so that on the next terminal count PERIOD and CC will be updated with the values written into PERIOD_BUFF and CC_BUFF.

A potential problem arises when software updates are not completed before the next tc event with an active pending swap event. For example, if software updates PERIOD_BUFF before the tc event and CC_BUFF after the tc event, swapping does not reflect the CC_BUFF register update. To prevent this from happening, the swap event should be generated by software through a register write after both the PERIOD_BUFF and CC_BUFF registers are updated. The swap event is kept pending by the hardware until the next tc event occurs.

The previous section addressed synchronized updates of the CC/CC_BUFF and PERIOD/PERIOD_BUFF registers of a single PWM using a software-generated swap event. During motor control, three PWMs work in unison and updates to all period and compare register pairs should be synchronized. All three PWMs have synchronized periods and as a result have synchronized tc events. The swap event for all three PWMs is generated by software through a single register write. The software should generate the swap events after the PERIOD_BUFF and CC_BUFF registers of all three PWMs are updated.

Note: When the counter is not running ((temporarily) stopped or killed), the PWM output signal values are determined by their respective polarity settings. When the counter is disabled the output values are low.

Figure 30-35. PWM Outputs When Killed



30.3.4.1 Asymmetric PWM

This PWM mode supports the generation of an asymmetric PWM. For an asymmetric PWM, the pwm_dt_input pulse is not necessarily centered in the middle of the period. This functionality is realized by having a different CC value when counting up and when counting down. The CC and CC_BUFF values are exchanged on an overflow event.

The COUNT_UPDN2 mode should use the same period value when counting up and counting down. When PERIOD and PERIOD_BUFF are swapped on a tc event (overflow or underflow event), care should be taken to ensure that:

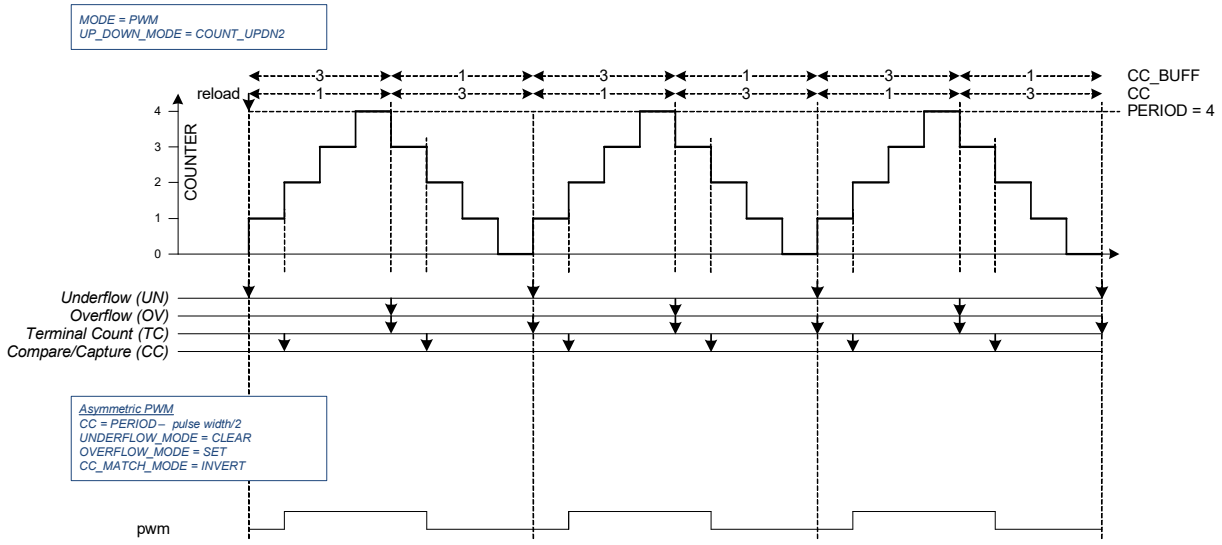
- Within a PWM period (tc event coincides with an overflow event), the period values are the same (an overflow swap of PERIOD and PERIOD_BUFF should not change the period value; that is, PERIOD_BUFF should be PERIOD)
- Between PWM periods (tc event coincides with an underflow event), the period value can change (an underflow swap of PERIOD and PERIOD_BUFF may change the period value; that is, PERIOD_BUFF may be different from PERIOD).

Figure 30-36 illustrates how the COUNT_UPDN2 mode is used to generate an asymmetric PWM.

Notes:

- When up counting, when CC value at the underflow event is 0, a cc_match event is generated.
- When down counting, when CC value at the overflow event is PERIOD, a cc_match event is generated.
- A tc event is generated for both an underflow and overflow event. The tc event is used to exchange the CC and CC_BUFF values.

Figure 30-36. Asymmetric PWM



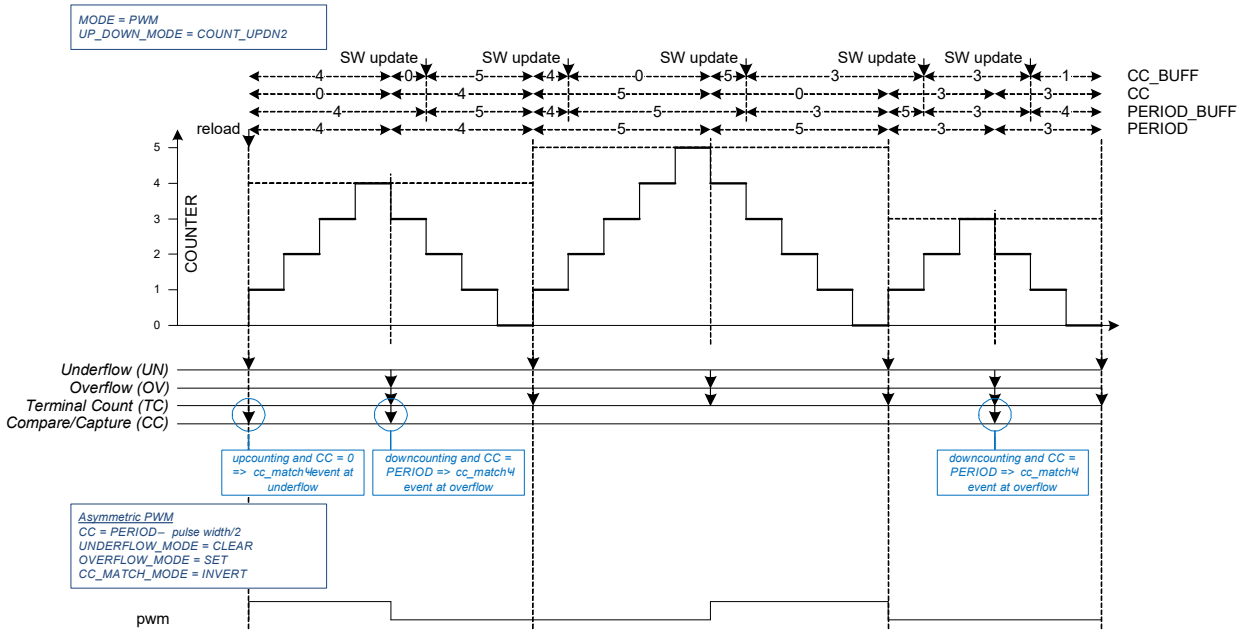
The previous waveform illustrated functionality when the CC values are neither “0” nor PERIOD. Corner case conditions in which the CC values equal “0” or PERIOD are illustrated as follows.

Figure 30-37 illustrates how the COUNT_UPDN2 mode is used to generate an asymmetric PWM.

Notes:

- When up counting, when CC value at the underflow event is 0, a cc_match event is generated.
- When down counting, when CC value at the overflow event is PERIOD, a cc_match event is generated.
- A tc event is generated for both an underflow and overflow event. The tc event is used to exchange the CC and CC_BUFF values.
- Software updates CC_BUFF and PERIOD_BUFF in an interrupt handler on the tc event (and overwrites the hardware updated values from the CC/CC_BUFF and PERIOD/PERIOD_BUFF exchanges).

Figure 30-37. Asymmetric PWM when Compare = 0 or Period



30.3.4.2 Configuring Counter for PWM Mode

The steps to configure the counter for the PWM mode of operation and the affected register bits are as follows.

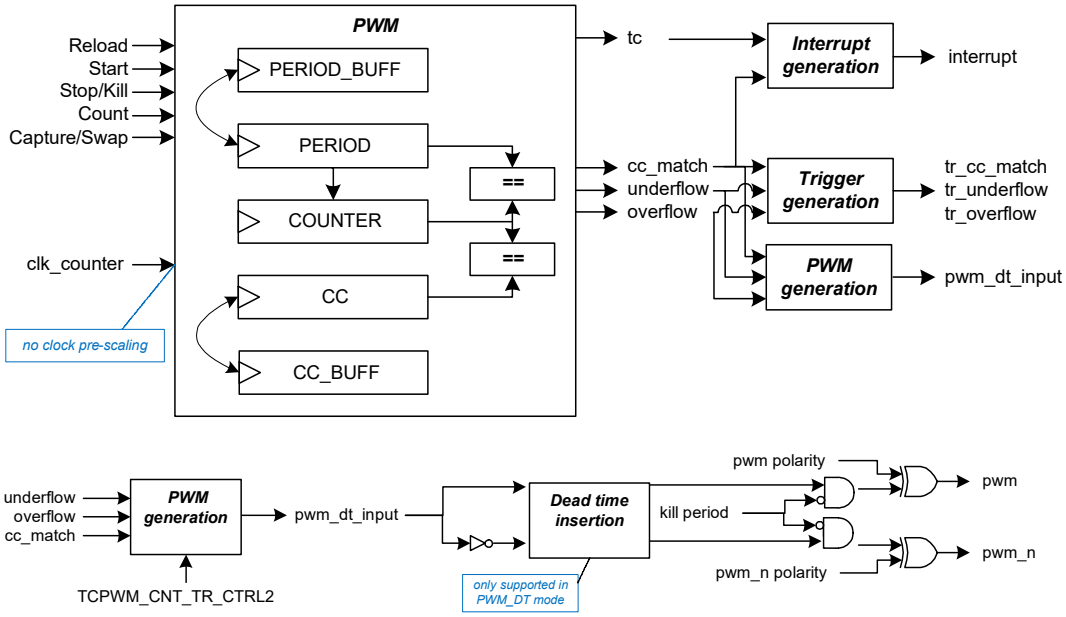
1. Disable the counter by writing '1' to the TCPWM_CTRL_CLR register.
2. Select PWM mode by writing '100b' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set clock prescaling by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register.
4. Set the required 16/32-bit period in the TCPWM_CNT_PERIOD register and the buffer period value in the TCPWM_CNT_PERIOD_BUFF register to swap values, if required.
5. Set the 16/32-bit compare value in the TCPWM_CNT_CC register and buffer compare value in the TCPWM_CNT_CC_BUFF register to swap values, if required.
6. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register to configure left-aligned, right-aligned, or center-aligned PWM.
7. Set the PWM_STOP_ON_KILL and PWM_SYNC_KILL fields of the TCPWM_CNT_CTRL register as required.
8. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, kill, swap, and count).
9. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (reload, start, kill, swap, and count).
10. pwm and pwm_n can be controlled by the TCPWM_CNT_TR_CTRL2 register to set, reset, or invert upon CC, OV, and UN conditions.
11. If required, set the interrupt upon TC or CC condition.
12. Enable the counter by writing '1' to the TCPWM_CTRL_SET register. A reload trigger must be provided through firmware (TCPWM_CMD_RELOAD register) to start the counter if the hardware reload signal is not enabled.

30.3.5 Pulse Width Modulation with Dead Time Mode

The PWM-DT functionality is the same as PWM functionality, except for the following differences:

- PWM_DT supports dead time insertion; PWM does not support dead time insertion.
- PWM_DT does not support clock pre-scaling; PWM supports clock pre-scaling.

Figure 30-38. PWM with Dead Time Functionality



Dead time insertion is a step that operates on a preliminary PWM output signal pwm_dt_input, as illustrated in Figure 30-38. Figure 30-39 illustrates dead time insertion for different dead times and different output signal polarity settings.

Figure 30-39. Dead-time Timing

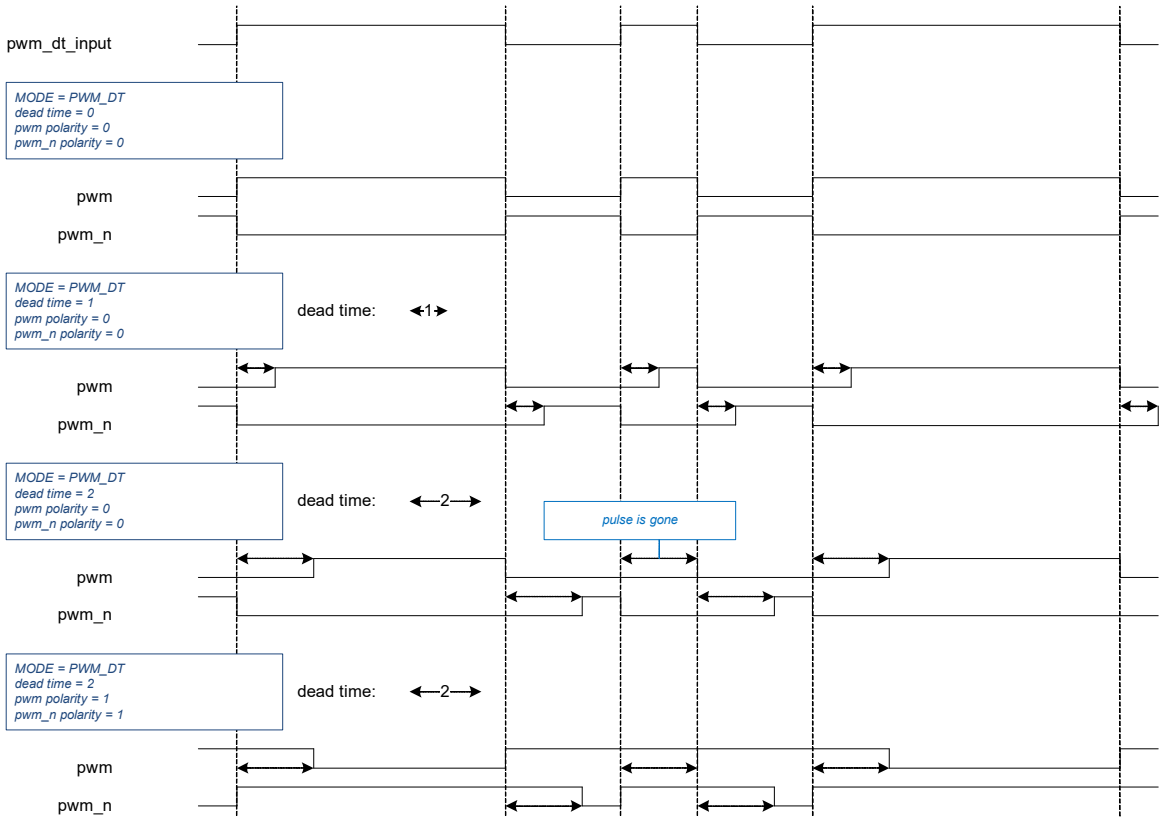
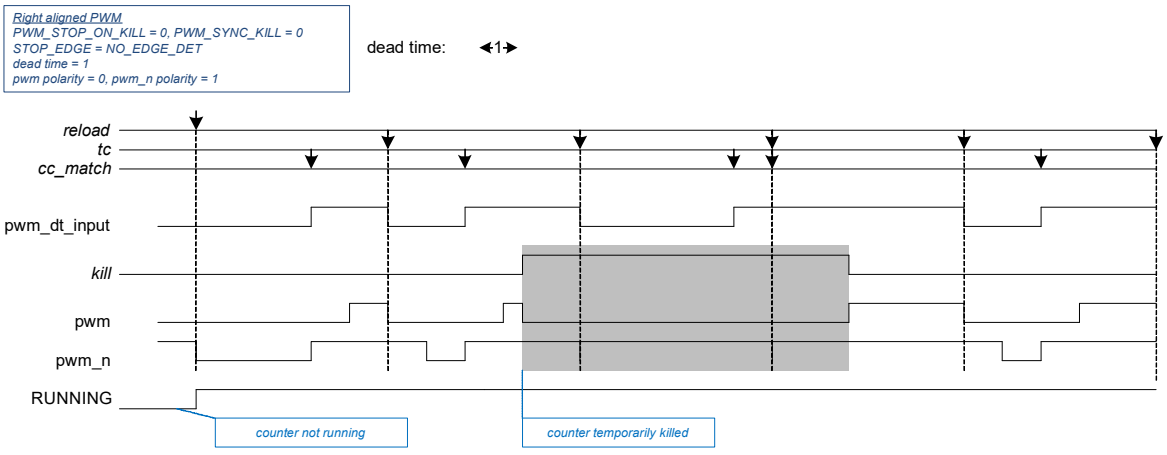


Figure 30-40 illustrates how the polarity settings and stop/kill functionality combined control the PWM output signals “pwm” and “pwm_n”.

Figure 30-40. Dead Time and Kill



30.3.5.1 Configuring Counter for PWM with Dead Time Mode

The steps to configure the counter for PWM with Dead Time mode of operation and the affected register bits are as follows:

1. Disable the counter by writing ‘1’ to the TCPWM_CTRL_CLR register.
2. Select PWM with Dead Time mode by writing ‘101’ to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required dead time by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register.
4. Set the required 16/32-bit period in the TCPWM_CNT_PERIOD register and the buffer period value in the TCPWM_CNT_PERIOD_BUFF register to swap values, if required.
5. Set the 16/32-bit compare value in the TCPWM_CNT_CC register and the buffer compare value in the TCPWM_CNT_CC_BUFF register to swap values, if required.
6. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register to configure left-aligned, right-aligned, or center-aligned PWM.
7. Set the PWM_STOP_ON_KILL and PWM_SYNC_KILL fields of the TCPWM_CNT_CTRL register as required.
8. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, kill, swap, and count).
9. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (reload, start, kill, swap, and count).
10. pwm and pwm_n can be controlled by the TCPWM_CNT_TR_CTRL2 register to set, reset, or invert upon CC, OV, and UN conditions.
11. If required, set the interrupt upon TC or CC condition.
12. Enable the counter by writing ‘1’ to the TCPWM_CTRL_SET register. A reload trigger must be provided through firmware (TCPWM_CMD_RELOAD register) to start the counter if the hardware reload signal is not enabled.

30.3.6 Pulse Width Modulation Pseudo-Random Mode (PWM_PR)

The PWM_PR functionality changes the counter value using the linear feedback shift register (LFSR). This results in a pseudo random number sequence. A signal similar to PWM signal is created by comparing the counter value COUNTER with CC. The generated signal has different frequency/noise characteristics than a regular PWM signal.

Table 30-26. PWM_PR Mode Trigger Inputs

Trigger Inputs	Usage
reload	Same behavior as start event. Can be used only when the counter is not running.
start	Starts the counter. The counter is not initialized by hardware. The current counter value is used. Behavior is dependent on UP_DOWN_MODE. Can be used only when the counter is not running.
stop/kill	Stops the counter. Different stop/kill modes exist.
count	Not used.
capture	This event acts as a swap event. When this event is active, the CC/CC_BUFF and PERIOD/PERIOD_BUFF registers are exchanged on a tc event (when specified by CTRL.AUTO_RELOAD_CC and CTRL.AUTO_RELOAD_PERIOD). A swap event requires rising, falling, or rising/falling edge event detection mode. Pass-through mode is not supported, unless the selected event is a constant '0' or '1'. When a swap event is detected and the counter is running, the event is kept pending until the next tc event. When a swap event is detected and the counter is not running, the event is cleared by hardware.

Note: Event detection is on the peripheral clock, clk_peri.

Table 30-27. PWM_PR Supported Features

Supported Features	Description
Clock pre-scaling	Pre-scales the counter clock, clk_counter.
One-shot	Counter is stopped by hardware, after a single period of the counter (counter value equals period value PERIOD).
Auto reload CC	CC and CC_BUFF are exchanged on a swap event AND tc event (when specified by CTRL.AUTO_RELOAD_CC).
Auto reload PERIOD	PERIOD and PERIOD_BUFF are exchanged on a swap event and tc event (when specified by CTRL.AUTO_RELOAD_PERIOD).
Kill modes	Specified by PWM_STOP_ON_KILL. See memory map for further details.

Note: The count event is not used. As a result, the PWM_PR functionality operates on the pre-scaled counter clock (clk_counter), rather than on an “active count” pre-scaled counter clock.

Table 30-28. PWM_PR Trigger Outputs

Trigger Outputs	Description
cc_match (CC)	Counter changes from a state in which COUNTER equals CC.
Underflow (UN)	Not used.
Overflow (OV)	Not used.

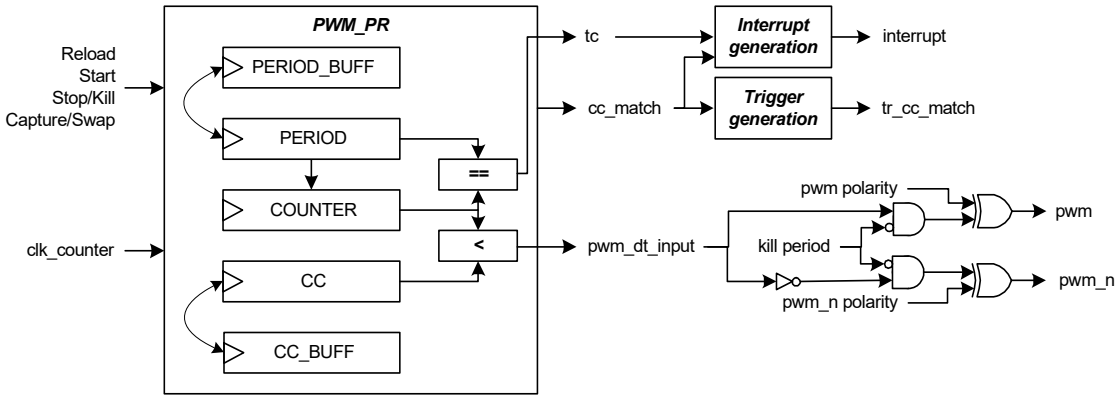
Table 30-29. PWM_PR Interrupt Outputs

Interrupt Outputs	Description
cc_match (CC)	Counter changes from a state in which COUNTER equals CC.
tc	Counter changes from a state in which COUNTER equals PERIOD.

Table 30-30. PWM_PR PWM Outputs

PWM Outputs	Description
pwm	PWM output.
pwm_n	Complementary PWM output.

Figure 30-41. PWM_PR Functionality



The PWM_PR functionality is described as follows:

- The counter value COUNTER is initialized by software (to a value different from 0).
- A reload or start event starts PWM_PR operation.
- During PWM_PR operation:
  - The counter value COUNTER is changed based on the LFSR polynomial: $x^{16} + x^{14} + x^{13} + x^{11} + 1$ (en.wikipedia.org/wiki/Linear_feedback_shift_register).

    temp = (COUNTER >> (16-16)) ^ (COUNTER >> (16-14)) ^ (COUNTER >> (16-13)) ^ (COUNTER >> (16-11)) or

    temp = (COUNTER >> 0) ^ (COUNTER >> 2) ^ (COUNTER >> 3) ^ (COUNTER >> 5);

    (COUNTER = (temp << 15)) | (COUNTER >> 1)

    This will result in a pseudo random number sequence for COUNTER. For example, when COUNTER is initialized to 0xACE1, the number sequence is: 0xACE1, 0x5670, 0xAB38, 0x559C, 0x2ACE, 0x1567, 0x8AB3… This sequence will repeat itself after $2^{16} – 1$ or 65535 counter clock cycles.
  - A 32-bit counter uses the LFSR polynomial: x^32 + x^30 + x^26 + x^25 + 1
  - When the counter value COUNTER equals CC, a cc_match event is generated.
  - When the counter value COUNTER equals PERIOD, a tc event is generated.
  - On a tc event, the CC/CC_BUFF and PERIOD/PERIOD_BUFF can be conditionally exchanged under control of the capture/swap event and the CTRL.AUTO_RELOAD_CC and CTRL.AUTO_RELOAD_PERIOD field (see PWM functionality).
  - The output pwm_dt_input reflects: COUNTER[14:0] < CC[15:0]. Note that only the lower 15 bits of COUNTER are used for comparison, while the COUNTER itself can run up to 16- or 32-bit values. As a result, for CC greater or equal to 0x8000, pwm_dt_input is always 1. The pwm polarity can be inverted (as specified by CTRL.QUADRATURE_MODE[0]).

As mentioned, different stop/kill modes exist. The mode is specified by PWM_STOP_ON_KILL (PWM_SYNC_KILL should be ‘0’ – asynchronous kill mode). The memory map describes the modes and the desired settings for the stop/kill event. The following two modes are supported:

- PWM_STOP_ON_KILL is ‘1’. This mode stops the counter on a stop/kill event.
- PWM_STOP_ON_KILL is ‘0’. This mode keeps the counter running, but suppresses the PWM output signals immediately and continues to do so for the duration of the stop/kill event.

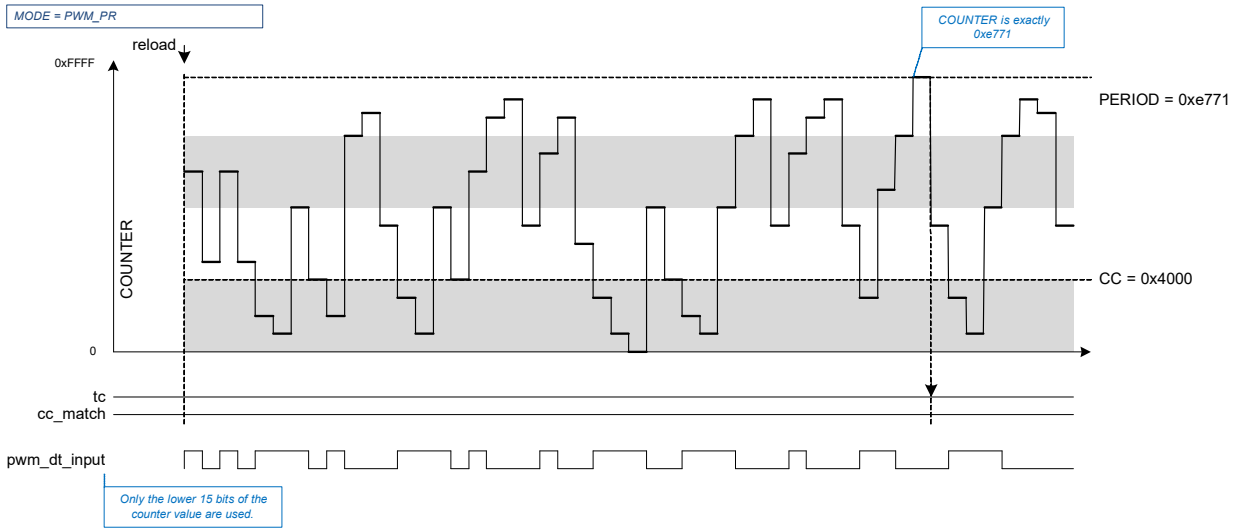
Note that the LFSR produces a deterministic number sequence (given a specific counter initialization value). Therefore, it is possible to calculate the COUNTER value after a certain number of LFSR iterations, n. This calculated COUNTER value can be used as PERIOD value, and the tc event will be generated after precisely n counter clocks.

Figure 30-42 illustrates PWM_PR functionality.

Notes:

- The grey shaded areas represent the counter region in which the pwm_dt_input value is '1', for a CC value of 0x4000. There are two areas, because only the lower 15 bits of the counter value are used.
- When CC is set to 0x4000, roughly one-half of the counter clocks will result in a pwm_dt_input value of '1'.

Figure 30-42. PWM_PR Output



30.3.6.1 Configuring Counter for Pseudo-Random PWM Mode

The steps to configure the counter for pseudo-random PWM mode of operation and the affected register bits are as follows.

1. Disable the counter by writing '1' to the TCPWM_CTRL_CLR register.
2. Select pseudo-random PWM mode by writing '110' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required period (16 bit) in the TCPWM_CNT_PERIOD register and buffer period value in the TCPWM_CNT_PERIOD_BUFF register to swap values, if required.
4. Set the 16-bit compare value in the TCPWM_CNT_CC register and the buffer compare value in the TCPWM_CNT_CC_BUFF register to swap values.
5. Set the PWM_STOP_ON_KILL and PWM_SYNC_KILL fields of the TCPWM_CNT_CTRL register as required.
6. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, kill, and swap).
7. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (reload, start, kill, and swap).
8. pwm and pwm_n can be controlled by the TCPWM_CNT_TR_CTRL2 register to set, reset, or invert upon CC, OV, and UN conditions.
9. If required, set the interrupt upon TC or CC condition.
10. Enable the counter by writing '1' to the TCPWM_CTRL_SET register. A reload trigger must be provided through firmware (TCPWM_CMD_RELOAD register) to start the counter if the hardware reload signal is not enabled.

30.4 TCPWM Registers

Table 30-31. List of TCPWM Registers

Register	Comment	Features
TCPWM_CTRL	TCPWM control register	Enables the counter block
TCPWM_CTRL_CLR	TCPWM control clear register	Used to avoid race-conditions on read-modify-write attempt to the CTRL register
TCPWM_CTRL_SET	TCPWM control set register	Used to avoid race-conditions on read-modify-write attempt to the CTRL register
TCPWM_CMD_CAPTURE	TCPWM capture command register	Generate a capture trigger input from software
TCPWM_CMD_RELOAD	TCPWM reload command register	Generate a reload trigger input from software
TCPWM_CMD_STOP	TCPWM stop command register	Generate a stop trigger input from software
TCPWM_CMD_START	TCPWM start command register	Generate a start trigger input from software
TCPWM_INTR_CAUSE	TCPWM counter interrupt cause register	Determines the source of the combined interrupt signal
TCPWM_CNT_CTRL	Counter control register	Configures counter mode, encoding modes, one-shot mode, swap mode, kill mode, dead time, clock pre-scaling, and counting direction
TCPWM_CNT_STATUS	Counter status register	Reads the direction of counting, dead time duration, and clock pre-scaling; checks whether the counter is running
TCPWM_CNT_COUNTER	Count register	Contains the 16- or 32-bit counter value
TCPWM_CNT_CC	Counter compare/capture register	Captures the counter value or compares the value with counter value
TCPWM_CNT_CC_BUFF	Counter buffered compare/capture register	Buffer register for counter CC register; swaps period value
TCPWM_CNT_PERIOD	Counter period register	Contains upper value of the counter
TCPWM_CNT_PERIOD_BUFF	Counter buffered period register	Buffer register for counter period register; swaps compare value
TCPWM_CNT_TR_CTRL0	Counter trigger control register 0	Selects trigger for specific counter events
TCPWM_CNT_TR_CTRL1	Counter trigger control register 1	Determine edge detection for specific counter input signals
TCPWM_CNT_TR_CTRL2	Counter trigger control register 2	Controls counter output lines upon CC, OV, and UN conditions
TCPWM_CNT_INTR	Interrupt request register	Sets the register bit when TC or CC condition is detected
TCPWM_CNT_INTR_SET	Interrupt set request register	Sets the corresponding bits in interrupt request register
TCPWM_CNT_INTR_MASK	Interrupt mask register	Mask for interrupt request register
TCPWM_CNT_INTR_MASKED	Interrupt masked request register	Bitwise AND of interrupt request and mask registers

31. Inter-IC Sound Bus

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The Inter-IC Sound Bus (I^2S) is a serial bus interface standard used to connect digital audio devices together. The specification is from Philips® Semiconductor (I^2S bus specification: February 1986, revised June 5, 1996). In addition to the standard I^2S format, the I^2S block also supports the Left Justified (LJ) format and the Time Division Multiplexed (TDM) format.

31.1 Features

- Supports standard Philips I^2S, LJ, and eight-channel TDM digital audio interface formats
- Supports both master and slave mode operation in all the digital audio formats
- Supports independent operation of Receive (Rx) and Transmit (Tx) directions
- Supports operating from an external master clock provided through an external IC such as audio codec
- Provides configurable clock divider registers to generate the required sample rates
- Supports data word length of 8-bit, 16-bit, 18-bit, 20-bit, 24-bit, and 32-bit per channel
- Supports channel length of 8-bit, 16-bit, 18-bit, 20-bit, 24-bit, and 32-bit per channel (channel length fixed at 32-bit in TDM format)
- Provides two hardware FIFO buffers, one each for the Tx block and Rx block, respectively
- Supports both DMA- and CPU-based data transfers

31.2 Architecture

Figure 31-1. I²S Block Diagram

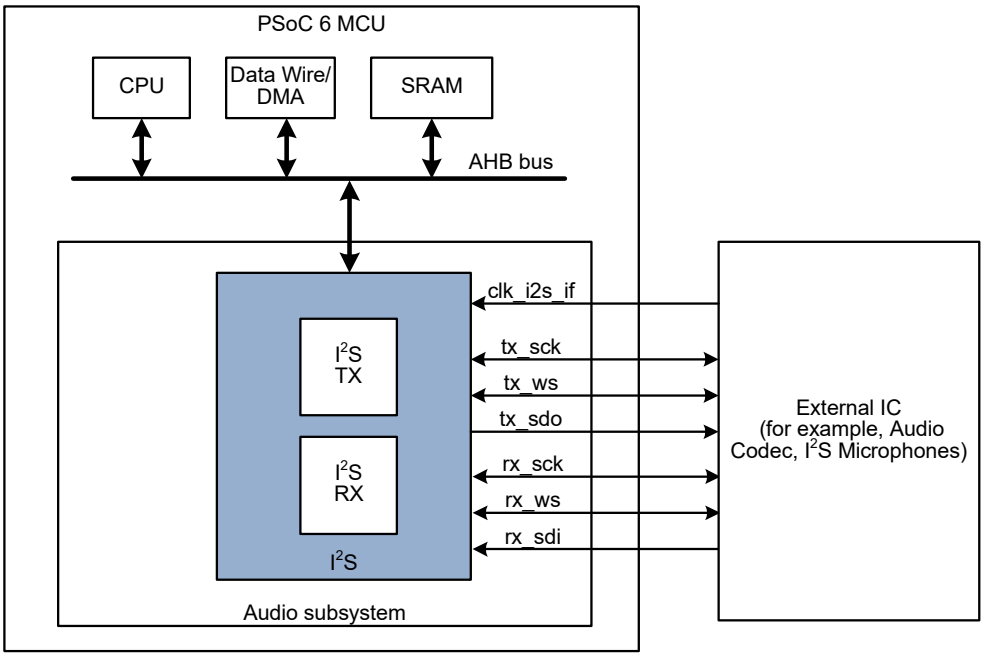


Figure 31-1 shows the high-level block diagram of the I²S block, which consists of two sub-blocks – I²S Transmitter (Tx) and I²S Receiver (Rx). The digital audio interface format and master/slave mode configuration can be done independently for the Tx and Rx blocks. In the master mode, the word select (ws) and serial data clock (sck) are generated by the I²S block in the PSoC 6 MCU. In the slave mode, the ws and sck signals are inputs signals to the PSoC 6 MCU, and are generated by the external master device. The I²S block configuration, control, and status registers, along with the FIFO data buffers are accessible through the AHB bus. AHB bus masters such as CPU and DMA can access the I²S registers through the AHB interface. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for information on port pin assignments of the I²S block signals and AC/DC electrical specifications.

31.3 Digital Audio Interface Formats

The I²S block supports the following digital audio interface formats.

- Standard I²S format
- Left Justified format
- Time Division Multiplexed (TDM) format

The Tx and Rx sub-blocks can be independently configured to support one of the above formats in either master or slave mode. The I2S_MODE bits in the I2S_TX_CTL and I2S_RX_CTL registers are used to configure the digital audio interface format for the Tx and Rx blocks respectively. The MS (Master/Slave) bit in the I2S_TX_CTL and I2S_RX_CTL registers is used to configure the blocks in master or slave mode.

31.3.1 Standard I²S Format

Figure 31-2 shows the timing diagrams for the different word length and channel length combinations in the standard I²S digital audio format. In the standard I²S format, the word select signal (ws) is low for left channel data, and high for right channel data. The ws signal transitions one bit-clock (sck) early relative to the start of the left/right channel data. All the serial data (sd), ws signal transitions on the falling edge of the sck signal, and the read operations on the ws and sd lines are usually done on the rising edge of sck. Therefore, the I²S Tx block writes to the serial data (tx_sdo) line on the falling edge of tx_sck,

and the I^2S Rx block reads the data (rx_sdi) on the rising edge of rx_sck. The serial data is transmitted most significant bit (MSb) first. Depending on whether the block is in master or slave mode, the ws/sck signals are either generated by the block (master mode) or input signals to the block (slave mode).

The I^2S block supports configurable word length and channel length selection options. The word length for the Tx and Rx blocks can be configured using the WORD_LEN bits in the I2S_TX_CTL and I2S_RX_CTL registers, respectively. The channel length for the Tx and Rx blocks can be configured using the CH_LEN bits in the I2S_TX_CTL and I2S_RX_CTL registers respectively. The channel length configuration should always be greater than or equal to the word length configuration. Ensure that when the I^2S Rx block is operated in slave mode, the master Tx device ensures that its channel length configuration aligns with the I^2S Rx block channel length setting. If there is channel length mismatch, the PSoC I^2S Rx block in slave mode will not operate correctly.

In the Tx block, when the channel length is greater than the word length, the unused bits can be transmitted either as ‘0’ or ‘1’. This selection is made using the OVHDATA bit in the I2S_TX_CTL register. In the Rx block, when the word length is less than 32 bits, the unused most significant bits written to the 32-bit Rx FIFO register can either be set to ‘0’ or sign bit extended. This selection is made using the BIT_EXTENSION bit in the I2S_RX_CTL register.

Figure 31-2. Standard I²S Format (Word Length and Channel Length Combination Timing Diagrams)

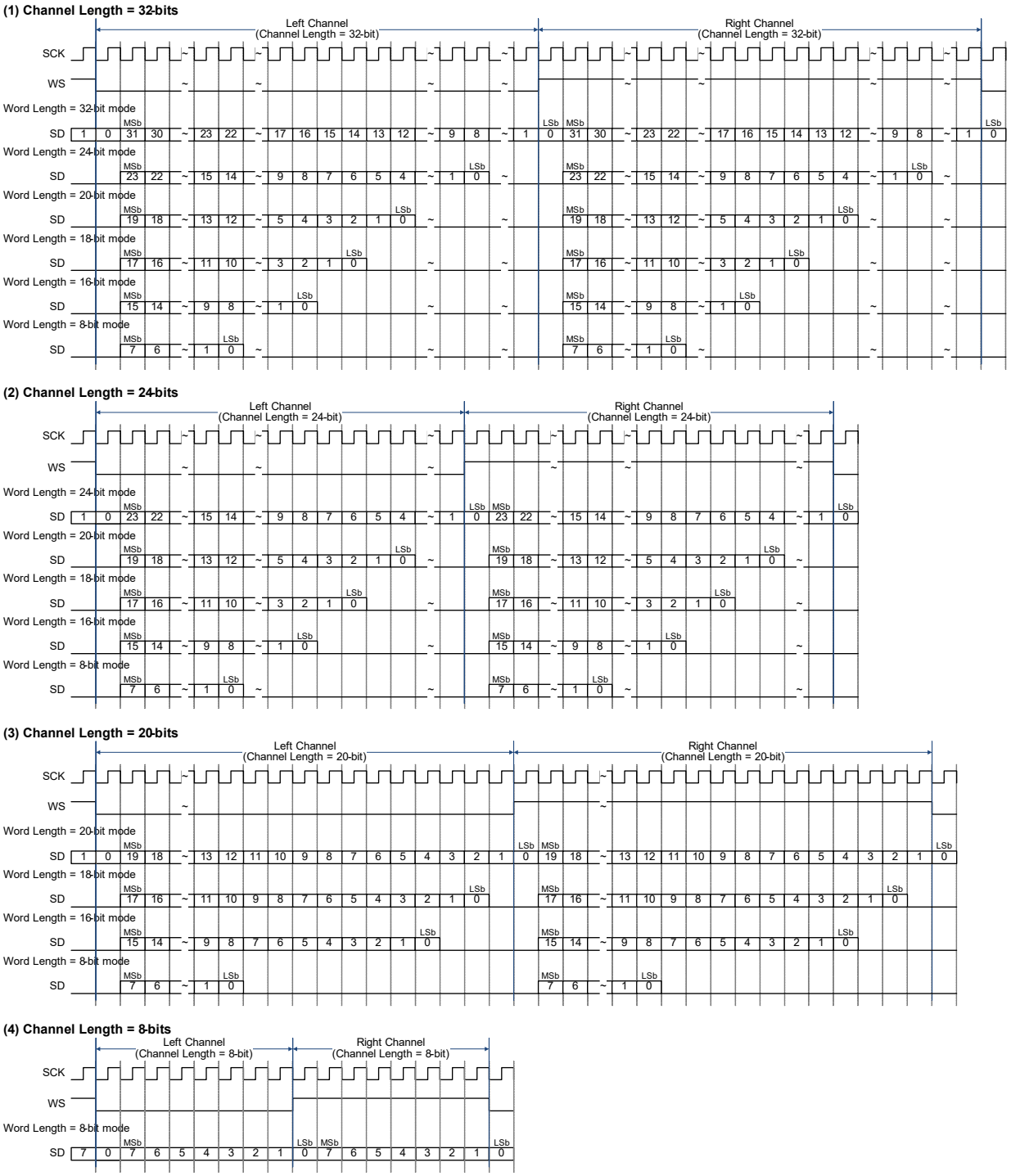


Table 31-1 lists the supported word length and channel length combinations.

Table 31-1. Word Length and Channel Length Combinations

		Word Length					
		8-bit	16-bit	18-bit	20-bit	24-bit	32-bit
Channel Length	32-bit	Valid	Valid	Valid	Valid	Valid	Valid
	24-bit	Valid	Valid	Valid	Valid	Valid	Invalid
	30-bit	Valid	Valid	Valid	Valid	Invalid	Invalid
	18-bit	Valid	Valid	Valid	Invalid	Invalid	Invalid
	16-bit	Valid	Valid	Invalid	Invalid	Invalid	Invalid
	8-bit	Valid	Invalid	Invalid	Invalid	Invalid	Invalid

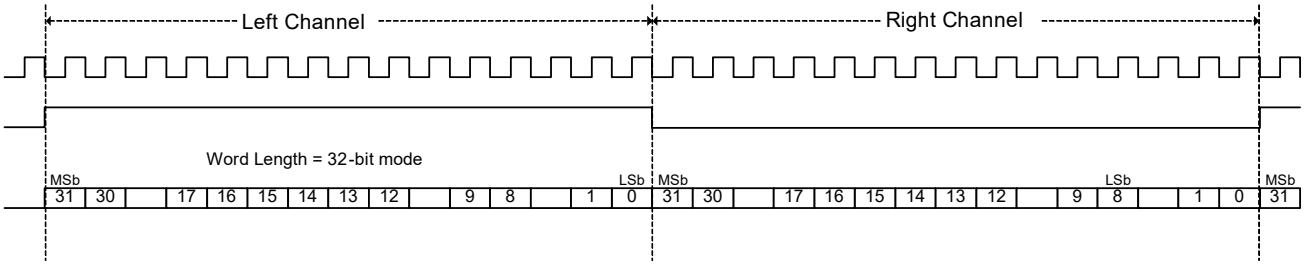
31.3.2 Left Justified (LJ) Format

Figure 31-3 shows the timing diagrams for the Left Justified interface format using the 32-bit channel length and 32-bit word length configuration as an example. The only differences between the standard I^{2}S and LJ formats are:

- In the standard I^{2}S format, WS signal is low for left channel data and high for right channel data. In the LJ format, WS signal is high for left channel data and low for right channel data.
- In the standard I^{2}S format, WS signal transitions one bit-clock (sck) early relative to the start of the channel data (coincides with LSb of the previous channel). In the LJ format, there is no early transition, and the WS signal transitions coincide with the start of the channel data.

Apart from these differences, all the features explained in the standard I^{2}S format section apply to the LJ format as well.

Figure 31-3. Left Justified Digital Audio Format



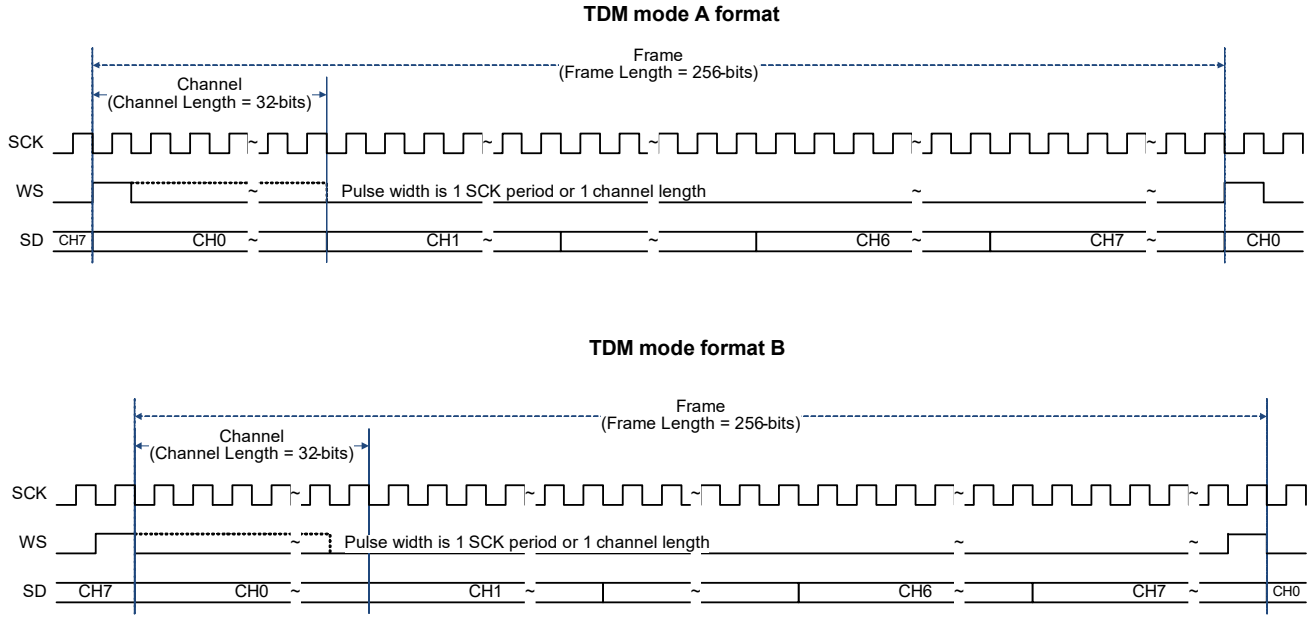
31.3.3 Time Division Multiplexed (TDM) Format

Figure 31-4 shows the timing diagrams for the two types of Time Division Multiplexed (TDM) formats supported by the I^{2}S block. The differences between the standard I^{2}S/LJ formats and the TDM format are as follows:

- Standard I^{2}S/LJ formats support only two channels (left/right) per frame, while TDM format supports up to eight channels per frame.
- In the TDM format, channel length for all eight channels is fixed at 32 bits. In the standard I^{2}S/LJ formats, the channel length is configurable. The word length per channel is configurable similar to the standard I^{2}S and the data is also transmitted most significant bit first. Similar to I^{2}S, when the word length per channel is less than the 32-bit channel length for Tx block, the OVHDATA bit in the I2S_TX_CTL register is used to fill the unused least significant channel data bits with either all zeros or all ones
- In the TDM format, all eight channels of data are always present in a frame, and thus the frame width is fixed at 256 bits. You have the option to configure the number of active channels in a frame by configuring the CH_NR bits in the I2S_TX_CTL and I2S_RX_CTL registers. In the standard I^{2}S/LJ format, the CH_NR should always be configured for two channels. The number of active channels in the TDM format can be less than or equal to eight channels. The unused (inactive) channels always follow the active channels in a frame. As an example, if CH_NR is set for four channels, CH0 to CH3 are the active channels and CH4 to CH7 are the unused channels. The OVHDATA bit in the I2S_TX_CTL register is used to fill the unused channels with either all zeros or all ones.

- The pulse width of the word select (WS) signal in the TDM format can be configured to be either one bit clock (sck) wide or one channel wide. The selection is made using the WS_PULSE bit in the I2S_TX_CTL and I2S_RX_CTL registers. The pulse width is fixed to one channel width in the I^{2}S/LJ format.
- Two types of TDM formats are supported. In TDM mode A, the WS rising edge signal to signify the start of frame coincides with the start of CH0 data. In TDM mode B, the WS rising edge signal to signify the start of frame is one bit clock (sck) early, relative to the start of CH0 data (coincides with the last bit of the previous frame). The selection between the two TDM formats is made using the I2S_MODE bits in the I2S_TX_CTL and I2S_RX_CTL registers.

Figure 31-4. TDM Digital Audio Interface Format



31.4 Clocking Polarity and Delay Options

The I^{2}S block supports configurable clock polarity and delay options to alleviate any timing issues in the system involving PCB signal propagation delays, and delays associated with internal device signal routing.

When the I^{2}S Tx block operates in the slave mode, the tx_sck and tx_ws signals are input signals to the PSoC 6 MCU, and the tx_sdo output signal is transmitted off the tx_sck falling edge. The tx_sdo signal is sampled by the external master device Rx block on the subsequent tx_sck rising edge. Timing issues arise if the tx_sdo signal reaching the master side Rx block does not meet the setup and hold time requirements for input data on the master side. The I^{2}S Tx block in the PSoC 6 MCU has an option to advance the serial data transmission by 0.5 SCK cycles when the B_CLOCK_INV bit in the I2S_TX_CTL register is set. This feature can be used if there are timing issues while operating the I^{2}S Tx block in slave mode.

Similarly, when the I^{2}S Rx block operates in the master mode, the rx_sck and rx_ws signals are output signals from the PSoC 6 MCU, and the rx_sdi signal is transmitted by the external master device on the falling edge of rx_sck. The PSoC I2S Rx block samples the rx_sdi signal on the subsequent rx_sck rising edge. Timing issues arise if the rx_sdi signal reaching the PSoC block does not meet the setup and hold time requirements for input data. The I^{2}S Rx block has an option to delay the serial data capture by 0.5 SCK cycles when the B_CLOCK_INV bit in the I2S_RX_CTL register is set. This feature can be used if there are timing issues while operating the I^{2}S Rx block in master mode.

In addition to these clock delay options, there is also an option to invert the outgoing bit clock (sck) in master mode by setting the SCKO_POL bit in the I2S_TX_CTL and I2S_RX_CTL registers. Similarly, in the slave mode, there is an option to invert the incoming bit clock (sck) by setting the SCKI_POL bit in the I2S_TX_CTL and I2S_RX_CTL registers.

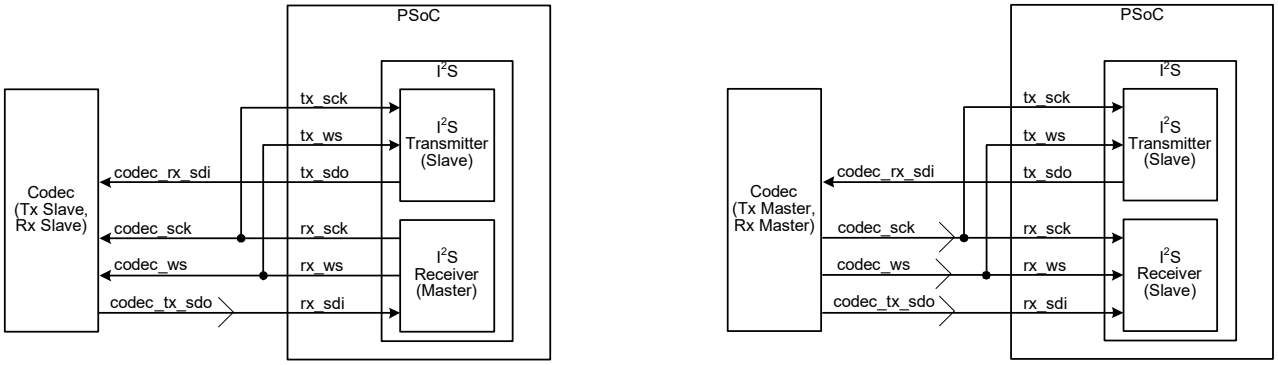
Refer to the registers TRM for detailed description of the B_CLOCK_INV, SCKI_POL, and SCKO_POL register configurations.

31.5 Interfacing with Audio Codecs

The I²S block in the PSoC 6 MCU interfaces with an audio codec device based on the choice of codec device and the end application requirements. Some scenarios and the connection diagrams are as follows:

- Codecs with separate ws and sck signals for the Rx and Tx directions: To interface with these codecs, the connections between the PSoC I²S block and the codec device will be as shown in Figure 31-1 where the PSoC I²S Tx signals (tx_sck, tx_ws, tx_sdo) connect to the codec Rx signals, and the PSoC I²S Rx signals (rx_sck, rx_ws, rx_sdi) connect to the codec Tx signals. The direction of sck (tx_sck, rx_sck) and ws (tx_ws, rx_ws) signals depends on which device is the master and which device is the slave.
- Codecs with common ws and sck signals for both Rx and Tx directions: There are two possible configurations to interface these codecs with the PSoC 6 MCU as shown in Figure 31-5. In both configurations, the sck signals (tx_sck, rx_sck, codec_sck) are shorted externally. The same goes for the ws signal connections as well (tx_ws, rx_ws, codec_ws). Ensure that only one block is driving the sck and ws lines. So when the codec acts as the slave device, the PSoC I²S Rx block should be in the master mode, and the PSoC I²S Tx block should be in the slave mode (or PSoC I²S Rx as slave and PSoC I²S Tx as master). When the codec acts as the master device, both the PSoC I²S Rx and PSoC I²S Tx blocks should be in slave mode.

Figure 31-5. Interfacing with Codecs having Common ws and sck Signals



31.6 Clocking Features

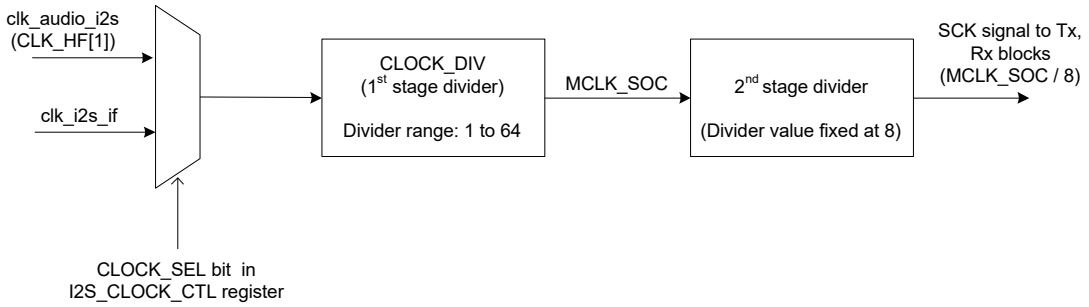
The I²S unit has three clock inputs.

Table 31-2. Clock Inputs

Signal	DESCRIPTION
clk_sys_i2s	System clock. This clock is used for the AHB slave Interface, control, status, and interrupt registers, and also clocks the DMA trigger control logic.
clk_audio_i2s	I²S internal clock. This clock is used for I²S transmitter (Tx)/receiver (Rx) blocks; it is asynchronous with the clk_sys_i2s. This clock is connected to the CLK_HF[1] high-frequency clock in the device. Refer to the Clocking System chapter on page 242 for more details on high frequency clocks.
clk_i2s_if	I²S external clock. This clock is provided from an external I²S bus host through a port pin. It is used in place of the clk_audio_i2s clock to synchronize I²S data to the clock used by the external I²S bus host.

Figure 31-6 shows the clocking divider structure in the I²S block. In the master mode, the sck and ws signals are generated either using the clk_audio_i2s internal clock or the clk_i2s_if external clock. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for the port pin assignment of clk_i2s_if clock. The CLOCK_SEL bit in the I2S_CLOCK_CTL register controls the selection between internal and external clocks.

Figure 31-6. Clocking Divider Structure



There are two stages of clock dividers in the I^{2}S block as follows.

- The first stage clock divider is used to generate the internal I^{2}S master clock (MCLK_SOC). The input clock to the first stage divider is either clk_audio_i2s or clk_i2s_if. The first stage clock divider is configured using the CLOCK_DIV bits in I2S_CLOCK_CTL register. Divider values from 1 to 64 are supported.
- The second stage clock divider is used to generate the sck signals. The input clock is the output from the first stage clock divider. This divider value is fixed at '8' (FTX_SCK = FRX_SCK = FMCLK_SOC/8). The word select (ws) signal frequency depends on the sck frequency, and the configured channel length value.

When in slave mode, the internal clock (MCLK_SOC) frequency should still be eight times the frequency of the input serial clock. You must choose the appropriate clock source and the CLOCK_DIV divider value to guarantee this condition is met in the slave mode of operation. Usually, when the PSoC I^{2}S block operates in the slave mode, the host sends a master clock which is an integral multiple of the sampling rate. This master clock can be routed to the clk_i2s_if port pin. The CLOCK_DIV divider value can then be adjusted to ensure that the MCLK_SOC is eight times the input SCK frequency.

Table 31-3 gives an example of the clock divider settings for operating the I^{2}S block at the standard sampling rates in the standard I^{2}S format. Note that the first stage divider values in the table are the register field values – the actual divider values are one more than the configured register values as explained in the clock divider section. Refer to the PSoC 61 datasheet/ PSoC 62 datasheet for details on maximum values of SCK frequency, and the output sampling rates.

Table 31-3. I^{2}S Divider Values for Standard Audio Sampling Rates in Standard I^{2}S Format

Sampling Rate (SR) (kHz)	WORD_LEN (bits)	SCK (2*WORD_LEN*SR) (MHz)	CLK_HF1 (or clk_i2s_if) (MHz)	(CLK_HF[1])/SCK (Total Divider Ratio)	CLK_CLOCK_DIV (First Divider)	Second Stage Divider (Fixed at 8)
8	32	0.512	49.152	96	11	8
16	32	1.024		48	5	
32	32	2.048		24	2	
48	32	3.072		16	1	
44.1	32	2.8224	45.1584	32	3	

31.7 FIFO Buffer and DMA Support

The I²S block has two FIFO buffers - one each for the Tx block and Rx block, respectively. The ordering format of the channel data in both Tx and Rx FIFOs depends on the configured digital audio format. This ordering format should be considered when writing to the Tx FIFO or reading from the Rx FIFO. In the standard I²S and LJ digital audio formats, the ordering of the data is (L, R, L, R, L, ...) where L refers to the left channel data and R refers to the right channel data. In the TDM format with the number of active channels set to four, the data order will be (CH0, CH1, CH2, CH3, CH0, CH1, CH2, CH3, CH0, ...). If the number of active channels is set to eight, the cycle will repeat after CH0–CH7 data.

I²S Tx FIFO: The I²S Tx block has a hardware FIFO of depth 256 elements where each element is 32-bit wide. In addition to this 256-element FIFO, the I²S block has an internal transmit buffer that can store four 32-bit data to be transmitted. This four-element buffer is used as an intermediary to hold data to be transferred on the I²S bus, and is not exposed to the AHB BUS interface.

The TX FIFO can be paused by setting the TX_PAUSE bit in I2S_CMD. When the TX_PAUSE bit is set, the data sent over I²S is “0”, instead of TX FIFO data. To resume normal operation, the TX_PAUSE bit must be cleared.

The I2S_TX_FIFO_CTL register is used for FIFO control operations. The TRIGGER_LEVEL bits in the I2S_TX_FIFO_CTL register can be used to generate a transmit trigger event when the Tx FIFO has less entries than the value configured in the TRIGGER_LEVEL bits.

The FIFO freeze operation can be enabled by setting the FREEZE bit in the I2S_TX_FIFO_CTL register. When the FREEZE bit is set and the Tx block is operational (TX_START bit in I2S_CMD is set), hardware reads from the Tx FIFO do not remove the FIFO entries. Also, the Tx FIFO read pointer will not be advanced. Any writes to the I2S_TX_FIFO register will increment the Tx FIFO write pointer; when the Tx FIFO becomes full, the internal write pointer stops incrementing. The freeze operation may be used for firmware debug purposes. This operation is not intended for normal operation. To return to normal operation after using the freeze operation, the I²S must be reset by clearing the TX_ENABLED bit in the I2S_CTL register, and then setting the bit again.

The CLEAR bit in the I2S_TX_FIFO_CTL register is used to clear the Tx FIFO by resetting the read/write pointers associated with the FIFO. Write accesses to the Tx FIFO using the I2S_TX_FIFO_WR or I2S_TX_FIFO_WR_SILENT registers are not allowed while the CLEAR bit is set.

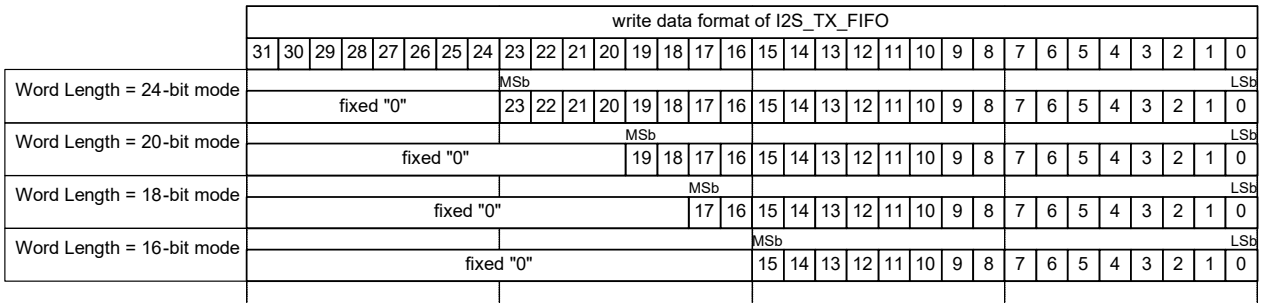
The I2S_TX_FIFO_STATUS register provides FIFO status information. This includes number of used entries in the Tx FIFO and the current values of the Tx FIFO read/write pointers. This register can be used for debug purposes. The I²S Tx FIFO read pointer is updated whenever the data is transferred from the Tx FIFO to the internal transmit buffer. Tx FIFO write pointer is updated whenever the data is written to the I2S_TX_FIFO_WR register, either through the CPU or the DMA controller.

For Tx FIFO data writes using the CPU, the hardware can be used to trigger an interrupt event for any of the FIFO conditions such as TX_TRIGGER, TX_NOT_FULL, and TX_EMPTY. As part of the interrupt handler, the CPU can write to the I2S_TX_FIFO_WR register. The recommended method is to write (256 - TRIGGER_LEVEL) words to the I2S_TX_FIFO_WR register every time the TX_TRIGGER interrupt event is triggered. In addition, interrupt events can be generated for FIFO overflow/underflow conditions.

For DMA-based Tx data transfers, the I²S Tx DMA trigger signal (tr_i2s_tx_req) can be enabled by writing ‘1’ to the TX_REQ_EN bit in I2S_TR_CTL register. The trigger signal output will become high whenever the Tx FIFO has less entries than that configured in the TRIGGER_LEVEL field. The DMA channel can be configured to transfer up to (256 - TRIGGER_LEVEL) words from the applicable source address (such as Flash and SRAM regions). The destination address of the DMA should always be the I2S_TX_FIFO_WR register address, with the destination address increment feature disabled in the DMA channel configuration. This FIFO address increment logic is handled internally to adjust the write pointer, and the DMA should not increment the destination address. For more details on DMA channel configuration, refer to the DMA Controller (DW) chapter on page 91.

The data in the I2S_TX_FIFO is always right-aligned. The I2S_TX_FIFO_WR format for different word length configurations is provided in Figure 31-7.

Figure 31-7. I2S_TX_FIFO_WR Register Format for Different Word Lengths



I²S Rx FIFO: The I²S Rx block has a hardware FIFO of depth 256 elements where each element is 32-bit wide. In addition to this 256-element FIFO, the I²S block has an internal receive buffer that can store four 32-bit data to be received. This four-element buffer is used as an intermediary to hold data received on the I²S bus, and is not exposed to the AHB BUS interface.

The I2S_RX_FIFO_CTL register is used for FIFO control operations. The TRIGGER_LEVEL bits in the I2S_RX_FIFO_CTL register is used to generate a receive trigger event when the Rx FIFO has more entries than the value configured in the TRIGGER_LEVEL bits. In the standard I²S/LJ format, the TRIGGER_LEVEL bits can be configured up to the allowed maximum value of 253. In the TDM format, the maximum value of TRIGGER_LEVEL is [254–CH_NR) where CH_NR is the number of active channels in the TDM frame.

The FIFO freeze operation can be enabled by setting the FREEZE bit in the I2S_RX_FIFO_CTL register. When the FREEZE bit is set and the Rx block is operational (RX_START bit in the I2S_CMD register is set), hardware will not write to the Rx FIFO. Also, the Rx FIFO write pointer will not be advanced. Any reads from the I2S_RX_FIFO register will increment the Rx FIFO read pointer; when the Rx FIFO becomes empty, the internal read pointer stops incrementing. The freeze operation may be used for firmware debug purposes. This operation is not intended for normal operation. To return to normal operation after using the freeze operation, the I²S must be reset by clearing the RX_ENABLED bit in the I2S_CTL register and then setting the bit again.

The CLEAR bit in I2S_RX_FIFO_CTL register is used to clear the Rx FIFO by resetting the read/write pointers associated with the FIFO. Read accesses from the Rx FIFO using the I2S_RX_FIFO_RD or I2S_RX_FIFO_RD_SILENT registers are not allowed while the CLEAR bit is set.

The I2S_RX_FIFO_STATUS register provides FIFO status information. This includes number of used entries in the Rx FIFO and the current values of the Rx FIFO read/write pointers. This register can be used for debug purposes. The I²S Rx FIFO write pointer is updated whenever the data is transferred to the Rx FIFO from the internal receive buffer. Rx FIFO read pointer is updated whenever the data is read from the I2S_RX_FIFO_RD register, either through the CPU or the DMA controller. For debug purposes, the I2S_RX_FIFO_RD_SILENT register is available, which always returns the top element of the Rx FIFO without updating the read pointer.

For Rx FIFO data reads using the CPU, the hardware can be used to trigger an interrupt event for any of the FIFO conditions such as RX_TRIGGER, RX_NOT_EMPTY, and RX_FULL. As part of the interrupt handler, the CPU can read from the I2S_RX_FIFO_RD register. The recommended method is to read (TRIGGER_LEVEL + 1) words from the I2S_RX_FIFO_RD register every time the RX_TRIGGER interrupt event is triggered. In addition, interrupt events can be generated for FIFO overflow/underflow conditions.

For DMA-based Rx data transfers, the I²S Rx DMA trigger signal (tr_i2s_rx_req) can be enabled by writing '1' to the RX_REQ_EN bit in the I2S_TR_CTL register. The trigger signal output will become high whenever the Rx FIFO has more entries than that configured in the TRIGGER_LEVEL field. The DMA channel can be configured to transfer up to (TRIGGER_LEVEL + 1) words to the applicable destination address (such as SRAM regions). The source address of the DMA should always be the I2S_RX_FIFO_RD register address, with the source address increment feature disabled in the DMA channel configuration. This FIFO address increment logic is handled internally to adjust the read pointer, and the DMA should not increment the source address. For more details on DMA channel configuration, refer to the DMA Controller (DW) chapter on page 91.

The data in the I2S_RX_FIFO is always right aligned. The I2S_RX_FIFO_RD format for different word length configurations is provided in Figure 31-8. Note that the unused most significant bits are either set as '0' or sign-bit extended depending on the BIT_EXTENSION bit in the I2S_RX_CTL register.

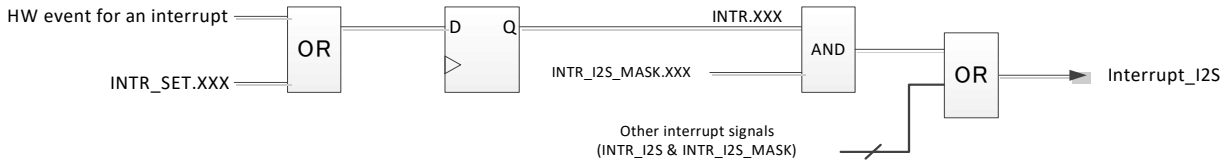
Figure 31-8. I2S_RX_FIFO_RD Register Format for Different Word Lengths

	read data format of I2S_RX_FIFO																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Word Length = 24-bit mode not Bit extension									MSb																							LSb
	fixed "0"								23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit extension	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Word Length = 20-bit mode not Bit extension													MSb																			LSb
	fixed "0"												19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit extension	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Word Length = 18-bit mode not Bit extension															MSb																	LSb
	fixed "0"														17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit extension	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Word Length = 16-bit mode not Bit extension																	MSb															LSb
	fixed "0"																15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit extension	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	"1"	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	"0"	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

31.8 Interrupt Support

The I^{2}S block has one interrupt output signal that goes to the interrupt controller in the CPU. Refer to the Interrupts chapter on page 56 for details on the vector number of the I^{2}S interrupt and the procedure to configure the interrupt priority, vector address, and enabling/disabling. The I^{2}S interrupt can be triggered under any of the following events - TX_TRIGGER, TX_NOT_FULL, TX_EMPTY, TX_OVERFLOW, TX_UNDERFLOW, TX_WD, RX_TRIGGER, RX_NOT_EMPTY, RX_FULL, RX_OVERFLOW, RX_UNDERFLOW, or RX_WD. Each of the interrupt events can be individually enabled/disabled to generate an interrupt condition. The I2S_INTR_MASK register is used to enable the required events by writing '1' to the corresponding bit. Irrespective of the INTR_MASK settings, if any of the events occur, the corresponding event status bit will be set by the hardware in the I2S_INTR register. The I2S_INTR_MASKED register is the bitwise AND of the I2S_INTR_MASK and I2S_INTR registers. The final I^{2}S interrupt signal is the logical OR of all the bits in the I2S_INTR_MASKED register. So only those events that are enabled in the I2S_INTR_MASK register are propagated as interrupt events to the interrupt controller. Interrupts can also be triggered in software by writing to the corresponding bits in I2S_INTR_SET register. Figure 31-9 illustrates the interrupt signal generation.

Figure 31-9. Interrupt Signal Generation



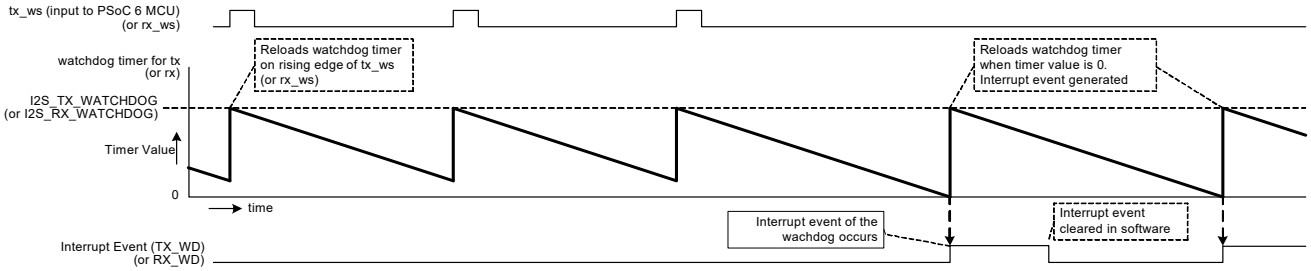
In the interrupt service routine (ISR), the I2S_INTR_MASKED register should be read to know the events that triggered the interrupt event. Multiple events can trigger the interrupt because the final interrupt signal is the logical OR output of the events. The ISR should do the tasks corresponding to each interrupt event that was triggered. At the end of the ISR, the value read in the I2S_INTR_MASKED register earlier should be written to the I2S_INTR register to clear the bits whose interrupt events were processed in the ISR. Unless the bits are not cleared by writing '1' to the I2S_INTR register, the interrupt signal will always be high. A dummy read of the I2S_INTR register should be done for the earlier register write to take effect.

31.9 Watchdog Timer

The Tx and Rx blocks have independent watchdog timers, which can be used to generate an interrupt event if the Word Select (WS) input is idle for more than the configured time period. This feature is available only in the slave mode of operation where the external master drives the WS input lines (tx_ws, rx_ws). This feature can be used to detect any signal transmission issues, master device issues, or if the master has halted communication. If the master drives the same word select signal to both the tx_ws and rx_ws lines, then only one of the watchdog timers can be enabled to cause the interrupt event. Although the following explanation covers Tx watchdog, the same explanation applies to Rx watchdog as well.

To enable the Tx watchdog timer feature, WD_EN bit in the I2S_TX_CTL register should be set. The watchdog timer reload value (32-bit timer) is configured by writing to the I2S_TX_WATCHDOG register. A value of zero written to the I2S_TX_WATCHDOG register will also disable the watchdog timer. Figure 31-10 illustrates the watchdog behavior when the timer is enabled. The timer runs off the CLK_PERI system clock. Refer to the Clocking System chapter on page 242 for details on generation of CLK_PERI. The timer starts running when WD_EN and TX_START bits are set. The timer reload happens either on a rising edge event on tx_ws input signal, or when the timer values reaches zero. When the timer value reaches zero, the TX_WD interrupt event is generated. The TX_WD bit in the I2S_INTR_MASK register should be set to enable interrupt generation by the watchdog timer interrupt event. The interrupt event can be cleared by writing '1' to the TX_WD bit in the I2S_INTR register.

Figure 31-10. Watchdog Timer Working



32. PDM-PCM Converter

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PDM-PCM unit accepts a stereo or mono serial data stream (pulse modulated 1-bit stream) coming from external digital PDM microphones. The PDM-PCM converter consists of a fifth order cascaded integrator comb (CIC) filter followed by a decimator, and a final stage high-pass filter. This block simplifies the conversion process by exposing the different configuration settings as registers, which you can program to meet the application needs. The entire PDM-PCM conversion process is handled in hardware; the PCM output data streaming can be done using the DMA controller thus freeing up the CPU bandwidth from performing periodic audio streaming activities.

32.1 Features

- Supports Mono/Stereo mode pulse density modulation (PDM) to pulse code modulation (PCM) conversion
- Accepts 1-bit PDM input and can generate 16-, 18-, 20-, or 24-bit PCM digital data output
- Configurable PDM microphone clock frequency
- Ability to generate standard audio sampling rates by adjusting the decimation rate and clock divider values
- Digital volume control: Programmable gain amplifier (PGA) control from –12 dB to +10.5 dB in 1.5-dB steps
- Smooth PGA and soft-mute control
- Hardware receive buffer: 24-bit wide, 255-element FIFO with support for DMA controller-based data transfer
- Optional high-pass filter to remove DC and low-frequency noise

32.2 Architecture

Figure 32-1. Block Diagram

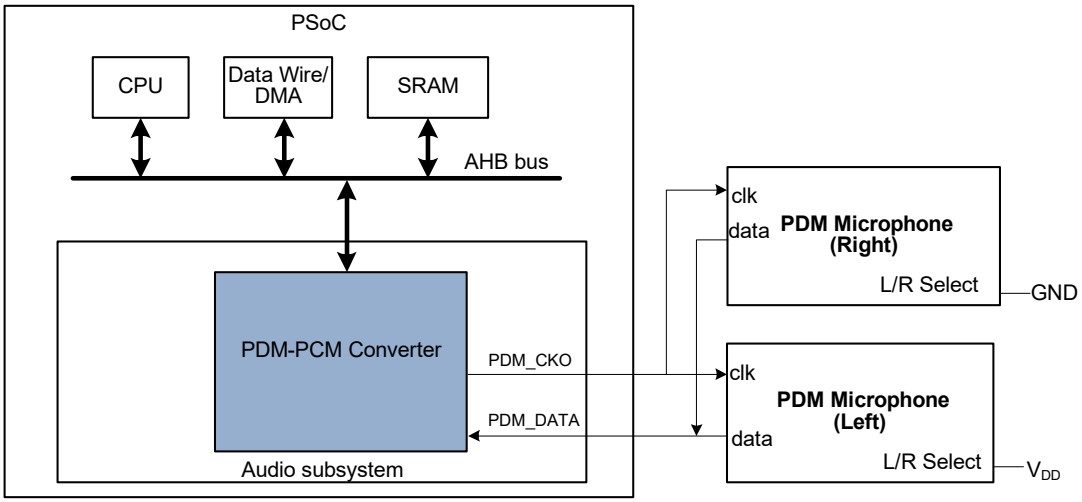


Figure 32-1 shows the block diagram of the PDM-PCM converter. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for information on port pin assignments of the PDM block signals, electrical specifications, and the interrupt vector number.

32.2.1 Enable/Disable Converter

The PDM-PCM converter can be powered ON or OFF by using the ENABLED bit in the PDM_CTL register. The block can be turned OFF when not used to save power. When the block is powered off by writing '0' to the ENABLED bit, the non-retention registers lose their current values. Refer to the registers TRM to know which registers are retention and non-retention type. When the block is enabled again, the non-retention registers are reset to their default values. User firmware should ensure that all non-retention registers are configured as required after the ENABLED bit is set, and before the PDM-PCM conversion is started by setting the STREAM_EN bit in the PDM_CMD register. See Operating Procedure on page 448 for the recommended procedure to configure this PDM-PCM converter before starting the PDM-PCM conversion.

32.2.2 Clocking Features

The block uses the CLK_HF[1] root high-frequency clock for performing the PDM-PCM conversion process. Refer to the Clocking System chapter on page 242 for details on selecting the clock source for CLK_HF[1], and configuring the divider registers to generate CLK_HF[1].

Figure 32-2. PDM-PCM Clocking Dividers

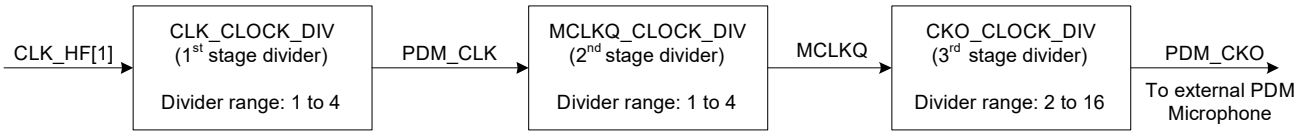


Figure 32-2 shows the clock divider structure in the block. The block has three stages of clock dividers to generate the clock (PDM_CKO), which goes to the external PDM microphone clock input.

1. The first stage clock divider (CLK_CLOCK_DIV field in the PDM_CLOCK_CTL register) is used to generate the actual clock signal (PDM_CLK) that goes to the PDM-PCM converter. The input is CLK_HF[1]; the CLK_CLOCK_DIV can be a value between 0 and 3 (divider value between 1 and 4).

   $f_{PDM\_CLK} = f_{CLK\text{-}HF[1]} / (CLK\_CLOCK\_DIV + 1)$

2. The second stage clock divider (MCLKQ_CLOCK_DIV field in the PDM_CLOCK_CTL register) is used to generate the internal master clock and can take a value between 0 and 3 (divider value between 1 and 4). The input clock is pdm_clk and the output of the divider is MCLKQ.

$f_{MCLKQ} = f_{PDM\_CLK} / (MCLKQ\_CLOCK\_DIV + 1)$

3. The third stage clock divider (CKO_CLOCK_DIV field in the PDM_CLOCK_CTL register) is used to generate the clock that goes to the PDM microphone clock (PDM_CKO) through the output pin of the PSoC 6 MCU. The input clock is MCLKQ. The divider register can be between 1 and 15 (divider value between 2 and 16).

   $f_{PDM\_CKO} = f_{MCLKQ} / (CKO\_CLOCK\_DIV + 1)$, if $CKO\_CLOCK\_DIV \geq 1$

   $f_{PDM\_CKO} = f_{MCLKQ} / 2$, if $CKO\_CLOCK\_DIV = 0$

32.2.3 Over-Sampling Ratio

Over-sampling ratio (OSR) is the ratio between the frequency of the PDM microphone clock (f_{PDM_CKO}) and the output PCM sampling rate frequency (f_S). The OSR is determined by the SINC_RATE bits in the PDM_CLOCK_CTL register. The relation is as follows.

$OSR = f_{PDM\_CKO} / f_S = 2 \times SINC\_RATE$

Table 32-1 gives an example of the PDM clock divider and SINC_RATE register configurations to generate the PCM output at standard sampling rates. Note that the PDM divider values in the table are the register field values – the actual divider values are one more than the configured register values as explained in the clock divider section. Refer to the PSoC 61 datasheet/ PSoC 62 datasheet for details on maximum values of PDM_CKO frequency, PDM_CLK frequency, and the output sampling rates.

Table 32-1. PDM Clock Divider Values for Standard Audio Sampling Rates

Sampling Rate (SR) (kHz)	SINC_RATE (= OSR/2)	PDM_CKO (=2*SINC_RATE*SR) (MHz)	CLK_HF1 (MHz)	(CLK_HF[1])/ (PDM_CKO) (Total Divider Ratio)	CLK_CLOCK_DIV (First Divider)	MCLKQ_CLOCK_DIV (Second Divider)	CKO_CLOCK_DIV (Third Divider)
8	32	0.512	49.152	96	1	3	11
16	32	1.024		48	1	3	5
32	32	2.048		24	0	3	5
48	32	3.072		16	0	1	7
44.1	32	2.8224	45.1584	32	0	1	7

There are various methods to generate the CLK_HF[1] frequencies listed in the table depending on the clocking options available on the device. Refer to the Clocking System chapter on page 242 for details on the clocking options available in the device, including the clock sources and PLL/FLL circuitry. For example, an external crystal oscillator (ECO) can be used in conjunction with a phase-locked loop (PLL) to generate the CLK_HF[1] at the desired frequency of 49.152 MHz or 45.1584 MHz.

One possible combination of PLL divider values to generate the 49.152 MHz frequency from a 17.2032 MHz ECO are: REFERENCE_DIV = 7, FEEDBACK_DIV = 100, OUTPUT_DIV = 5. One possible combination of PLL divider values to generate the 45.1584 MHz frequency from a 17.2032 MHz ECO are: REFERENCE_DIV = 8, FEEDBACK_DIV = 105, OUTPUT_DIV = 5.

32.2.4 Mono/Stereo Microphone Support

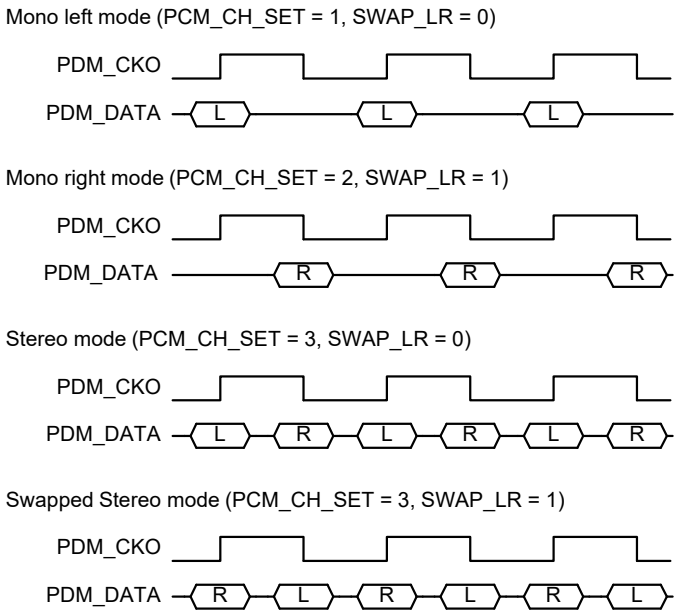
The PDM-PCM converter supports mono-left, mono-right, stereo, and swapped stereo modes of operation. The operation mode is controlled by the PDM_CH_SET and SWAP_LR bits in the PDM_MODE_CTL register. The register settings for the different operation modes are given in Table 32-2. The table also lists the invalid register settings, which you must not use in the firmware.

Table 32-2. Operation Mode Register Settings

Register Setting	Operation Mode
PCM_CH_SET = 0, SWAP_LR = 0 or 1	Recording OFF
PCM_CH_SET = 1, SWAP_LR = 0	Mono-Left recording mode. Only the left microphone channel is sampled on the rising edge of PDM_CKO. FIFO buffer contains only left channel data.
PCM_CH_SET = 2, SWAP_LR = 1	Mono-Right recording mode. Only the right microphone channel is sampled on the falling edge of PDM_CKO. FIFO buffer contains only right channel data.
PCM_CH_SET = 3, SWAP_LR = 0	Stereo recording mode. The right microphone channel is sampled on the falling edge of PDM_CKO and left channel on rising edge. FIFO buffer contains data in L/R format (left channel followed by right channel)
PCM_CH_SET = 3, SWAP_LR = 1	Swapped Stereo recording mode. The right microphone channel is sampled on the rising edge of PDM_CKO and left channel on falling edge. FIFO buffer contains data in L/R format (left channel followed by right channel)
PCM_CH_SET = 1, SWAP_LR = 1 or PCM_CH_SET = 2, SWAP_LR = 0	Invalid setting (not supported). Do not operate the PDM-PCM converter in these configuration settings.

Figure 32-3 shows the timing diagrams for the different operating modes.

Figure 32-3. PDM Mono/Stereo Timing



To alleviate uncertain board delay impact on PDM_IN setup and hold timing constraints, the PDM-PCM provides CKO_DELAY bits in the PDM_MODE_CTL register to add extra delay for the PDM_CKO path to internal sampler. CKO_DELAY can be a value between 0 and 7. A value of '0' implies that internal sampling of PDM data is advanced by three PDM_CLK clock cycles for the PDM_CKO transition. A value of '7' implies that internal sampling of PDM data is delayed by four PDM_CLK clock cycles for the PDM_CKO transition. Refer to the registers TRM for details on the meaning of different CKO_DELAY values.

Note: Variations have been observed in the recommendation for left/right sampling logic among the different PDM microphone manufacturers. The SWAP_LR bit in the PDM-PCM converter ensures that you can adjust the sampling logic according to the microphone datasheet recommendations. Refer to the PDM microphone manufacturer datasheet for the exact timing details. Also, in stereo mode, use the same manufacturer for both the left/right PDM microphones to ensure the timing behavior is uniform for both channels.

32.2.5 Hardware FIFO Buffers and DMA Controller Support

The PDM-PCM converter has a hardware FIFO depth of 255 elements where each element is 24-bit wide.

The PDM_RX_FIFO_CTL register is used for FIFO control operations. Refer to the register description in the registers TRM for more details. The TRIGGER_LEVEL field in the PDM_RX_FIFO_CTL register is used to generate a receive trigger event (interrupt event, DMA trigger signal) when the Rx FIFO has more entries than the value configured in the TRIGGER_LEVEL field. The TRIGGER_LEVEL field can be configured up to 254 in the mono microphone recording mode and up to 253 in the stereo microphone recording mode.

The FIFO freeze operation can be enabled by setting the FREEZE bit in the PDM_RX_FIFO_CTL register. When the FREEZE bit is set and the Rx block is operational (STREAM_EN bit in the PDM_CMD register is set), hardware will not write to the Rx FIFO. Also, the Rx FIFO write pointer will not be advanced. Any reads from the PDM_RX_FIFO_RD register will increment the Rx FIFO read pointer; when the Rx FIFO becomes empty, the internal read pointer stops incrementing. The freeze operation may be used for firmware debug purposes. This operation is not intended for normal operation. To return to normal operation after using the freeze operation, the PDM-PCM must be reset by clearing the ENABLED bit in PDM_CTL register, and then setting the bit again.

The CLEAR bit in the PDM_RX_FIFO_CTL register is used to clear the Rx FIFO by resetting the read/write pointers associated with the FIFO. Read accesses from the Rx FIFO using PDM_RX_FIFO_RD or PDM_RX_FIFO_RD_SILENT registers are not allowed while the CLEAR bit is set.

The PDM_RX_FIFO_STATUS register provides FIFO status information. This includes number of used entries in the Rx FIFO and the current values of the Rx FIFO read/write pointers. This register can be used for debug purposes. The Rx FIFO write pointer is updated whenever the data is transferred to the Rx FIFO from the internal receive buffer. Rx FIFO read pointer is updated whenever the data is read from the PDM_RX_FIFO_RD register, either through the CPU or the DMA controller. For debug purposes, the PDM_RX_FIFO_RD_SILENT register is available, which always returns the top element of the Rx FIFO without updating the read pointer.

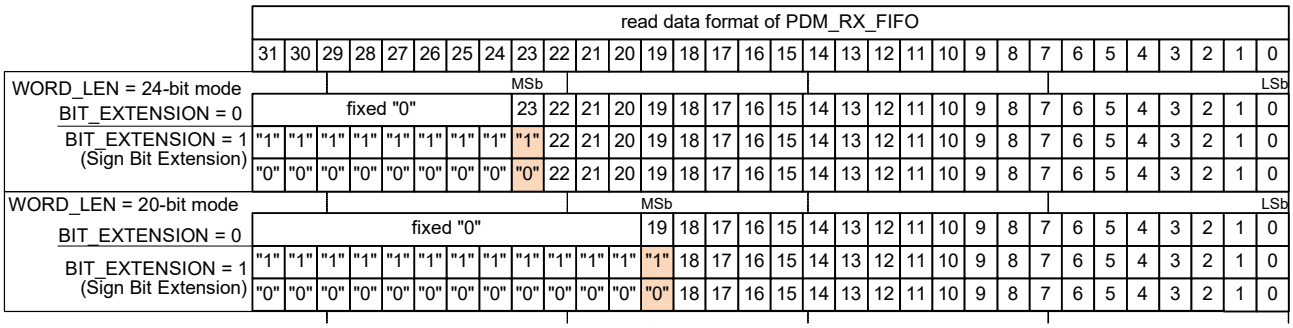
For Rx FIFO data reads using the CPU, the hardware can be used to trigger an interrupt event for any of the FIFO conditions such as RX_TRIGGER and RX_NOT_EMPTY. As part of the interrupt handler, the CPU can read from the PDM_RX_FIFO_RD register. The recommended method is to read (TRIGGER_LEVEL + 1) words from the PDM_RX_FIFO_RD register every time the RX_TRIGGER interrupt event is triggered. In addition, interrupt events can be generated for FIFO overflow and underflow conditions.

For DMA-based data transfers, the DMA trigger signal (tr_pdm_rx_req) can be enabled by writing '1' to the RX_REQ_EN bit in the PDM_TR_CTL register. The trigger signal output will become high whenever the Rx FIFO has more entries than that configured in the TRIGGER_LEVEL field. Refer to the Trigger Multiplexer Block chapter on page 294 for details on how to connect the DMA trigger signal to a particular DMA channel. The DMA channel can be configured to transfer up to (TRIGGER_LEVEL + 1) words to the applicable destination address (such as SRAM regions). The source address of the DMA should always be the PDM_RX_FIFO_RD register address, with the source address increment feature disabled in the DMA channel configuration. This FIFO address increment logic is handled internally to adjust the read pointer, and the DMA should not increment the source address. For more details on DMA channel configuration, refer to the DMA Controller (DW) chapter on page 91.

The successive data read from the PDM_RX_FIFO_RD follows the Left 1/Right 1/Left 2/Right 2/… format in stereo and swapped stereo modes of operation. For mono left and mono right recording modes, the data read from FIFO contains either the left channel data (mono left mode) or the right channel data (mono right mode).

The data in the PDM_RX_FIFO_RD is always right-aligned. The PDM_TX_FIFO_RD format for different word length configurations is provided in Figure 32-4. Note that the unused most significant bits are either set as '0' or sign-bit extended depending on the BIT_EXTENSION field setting in the PDM_DATA_CTL register.

Figure 32-4. FIFO Register Structure



32.2.6 Interrupt Support

The block has one output signal (interrupt_pdm) that goes to the interrupt controller in the CPU. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for details on the vector number of the PDM-PCM interrupt. Refer to the Interrupts chapter on page 56 for the procedure to configure the interrupt priority, vector address, and enabling/disabling.

The PDM interrupt can be triggered under any of the following events – RX_TRIGGER, RX_NOT_EMPTY, RX_OVERFLOW, or RX_UNDERFLOW. Table 32-3 lists the trigger conditions and details of these events.

Table 32-3. Interrupt Event Trigger Condition

Interrupt Event	Trigger Condition
RX_TRIGGER	The PDM Rx FIFO has more entries than the value specified in the TRIGGER_LEVEL field in the PDM_RX_FIFO_CTL register. At least (TRIGGER_LEVEL + 1) words can be read from the PDM_RX_FIFO_RD register in the interrupt service routine.
RX_NOT_EMPTY	The PDM Rx FIFO has at least one word that can be read from the PDM_RX_FIFO_RD register.
RX_OVERFLOW	The PDM Rx FIFO content is overwritten by the PDM-PCM converter due to the number of unread words exceeding the 256 word buffer capacity. The output PCM data after the overflow condition is discarded and not transferred to the FIFO. This event can be used to detect bandwidth constraints in the end application due to DMA or the interrupt used for data transfer not getting the required priority for executing the data transfers.
RX_UNDERFLOW	Attempt to read from an empty PDM Rx FIFO. This can happen due to incorrect configuration of the DMA or the interrupt service routine code used to do the data transfer.

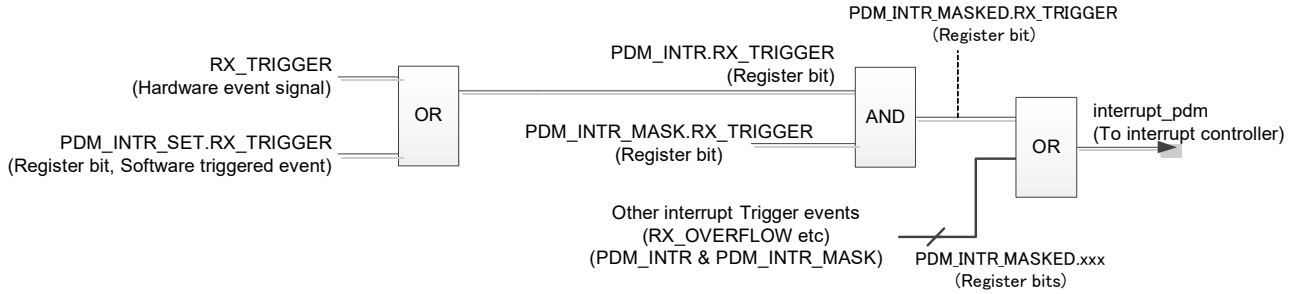
Each of the interrupt events can be individually enabled or disabled to generate the interrupt condition. The PDM_INTR_MASK register is used to enable the required events by writing ‘1’ to the corresponding bit.

Irrespective of the INTR_MASK settings, if any event occurs, the corresponding event status bit will be set by the hardware in the PDM_INTR register. The PDM_INTR_MASKED register is the bitwise AND of the PDM_INTR_MASK and PDM_INTR registers. The final PDM interrupt signal is the logical OR of all the bits in the PDM_INTR_MASKED register. So only those events that are enabled in the PDM_INTR_MASK register are propagated as interrupt events to the interrupt controller.

Interrupt events can also be triggered in software by writing to the corresponding bits in PDM_INTR_SET register.

Figure 32-5 illustrates the interrupt signal generation from the PDM-PCM block as explained above. Only the RX_TRIGGER interrupt generation is highlighted in the figure; the remaining interrupt events also follow the same generation logic.

Figure 32-5. PDM Interrupt Signal Generation



In the interrupt service routine (ISR) corresponding to the interrupt vector number of interrupt_pdm, the PDM_INTR_MASKED register should be read to know the events that triggered the interrupt event. Multiple events can trigger the interrupt because the final interrupt signal is the logical OR output of the events. The ISR should do the tasks corresponding to each interrupt event that was triggered. At the end of the ISR, the value read in the PDM_INTR_MASKED register earlier should be written to the PDM_INTR register to clear the bits whose interrupt events were processed in the ISR. A dummy read of the PDM_INTR register should be done for the earlier register write to PDM_INTR to take effect.

All of the interrupt event bits in PDM_INTR register will continue to indicate the event condition regardless of the true state until that bit is cleared (for example, when set, the RX_OVERFLOW bit will continue to indicate an OVERFLOW state until the RX_OVERFLOW bit is cleared regardless of the true state of the FIFO. A mere FIFO read of the FIFO will not clear the RX_OVERFLOW bit.). Unless the PDM_INTR bits that are used to generate the interrupt are not cleared by writing '1' to the PDM_INTR bits, the interrupt signal will always be high.

32.2.7 Digital Volume Gain

The PDM-PCM converter supports independent digital volume control on the left/right channels with a range from –12.5 dB to +10.5 dB in steps of 1.5 dB. It is programmed by configuring the PGA_R and PGA_L bits in the PDM_CTL register. PGA gain may be changed on the fly during normal operation, or as a one-time setting before starting the PDM-PCM conversion process.

32.2.8 Smooth Gain Transition

To reduce zipper or clip noise during on-the-fly gain transition or during soft mute operation, a built-in volume smoother is implemented with fine gain steps and fine time steps that enable soft ramp up or ramp down of the volume levels. Two fine gain options of 0.13 dB and 0.26 dB step sizes are available. The fine gain is set by the STEP_SEL bit in the PDM_CTL register. In addition to the fine gain steps, a time step is available for the fine gain change in terms of the number of sample cycles. The time step can be configured to a value between 64 sample periods and 512 sample periods using the S_CYCLES bits in the PDM_MODE_CTL register. So the STEP_SEL and S_CYCLES bit settings together determine the rate at which PGA gain or the soft mute transitions take effect.

32.2.9 Soft Mute

The PDM-PCM contains a built-in software-controlled mute function that digitally attenuates signals to imperceptible levels or zero. When mute function is enabled by setting the SOFT_MUTE bit in the PDM_CTL register, the corresponding PCM output is decreased from current level to mute state through predefined granular gain step per time constant transition. The STEP_SEL bit setting determines the gain step and the S_CYCLES bits determine the time constant. During soft-mute, the block is still ON and the PCM data streaming is operational; the DMA or CPU-based data transfer also happens as usual. Only the PCM output level is muted. When mute function is disabled by setting SOFT_MUTE = 0, the mute function is OFF and the PDM-PCM returns to normal operation where output signal level goes up to normal with current PGA gain.

32.2.10 Word Length and Sign Bit Extension

The PCM output word length can be configured for either 16-bits, 18-bits, 20-bits, or 24-bits using WORD_LEN bits in the PDM_DATA_CTL register. Irrespective of the word length setting, the PCM output is always read from the FIFO data buffer register (PDM_RX_FIFO_RD) as a 32-bit value. The unused most significant bits in the 32-bit value can either be sign extended or extended by '0' by using the BIT_EXTENSION bit in the PDM_DATA_CTL register.

32.2.11 High-Pass Filter

The PDM-PCM converter has a final stage high-pass filter (HPF) that blocks DC offset and low-frequency noise in signal band. The HPF is enabled when the HPF_EN_N bit in the PDM_MODE_CTL register is zero, and disabled when the HPF_EN_N bit is 1.

The filter response for HPF is characterized as:

$$H_{(z)} = \frac{1 - z^{-1}}{1 - (1 - 2^{-HPF\_GAIN})z^{-1}}$$

The HPF operates at the final PCM output sampling rate. HPF_GAIN is a 4-bit gain configuration setting in the PDM_MODE_CTL register. In default mode, HPF_GAIN = 0xB, so the HPF can be formulated by polynomial:

$$H_{(z)} = \frac{1 - z^{-1}}{1 - 0.99951z^{-1}}$$

The HPF_GAIN setting can be tuned to adjust HPF cutoff corner frequency for better system configuration.

32.2.12 Enable/Disable Streaming

The PDM-PCM conversion process can be dynamically enabled/disabled by using the STREAM_EN bit in the PDM_CMD register.

32.2.13 Power Modes

The PDM-PCM can operate in Active and Sleep CPU modes while in LP or ULP system power modes. It is not operational in system Deep Sleep or Hibernate power modes. When the device transitions from Deep Sleep/ Hibernate power modes to the LP/ULP power modes, the non-retention registers lose their previous configuration values. So the non-retention registers must be appropriately configured before enabling the PDM-PCM again for LP/ULP mode operation. One option is to store the non-retention register values in SRAM before entering Deep Sleep/ Hibernate modes. When returning to the LP/ULP modes, the SRAM values can be copied to the registers after enabling the PDM-PCM by setting the ENABLED bit in the PDM_CTL register. Refer to the registers TRM to identify the non-retention registers for the PDM.

32.3 Operating Procedure

32.3.1 Initial Configuration

The sequence of steps for initial configuration of the PDM-PCM converter before starting the conversion process is as follows:

1. Configure the clock dividers and decimation rate in the PDM_CLOCK_CTL register. This register configuration should be done before enabling the PDM-PCM converter. If the ENABLED bit in the PDM_CTL register is set, it should be cleared before changing the clock configuration.
2. Enable the block; set the PGA gain and fine gain step setting as required by writing to the PDM_CTL register.
3. Configure the PDM_MODE_CTL and PDM_DATA_CTL registers as required.
4. Configure the Rx FIFO trigger level setting by writing to the TRIGGER_LEVEL bits in the PDM_RX_FIFO_CTL register. The CLEAR and FREEZE bits in PDM_RX_FIFO_CTL are not set for normal operation.
5. Configure the events that must generate the interrupt by setting the corresponding bits in the PDM_INTR_MASK register, and clearing the remaining bits.
6. Configure the interrupt PDM interrupt vector and enable the interrupt vector. See the Interrupts chapter on page 56 for details.
7. If a DMA-based data transfer is required, connect the PDM DMA trigger signal (tr_pdm_rx_req) to the trigger input of the required DMA channel. See the Trigger Multiplexer Block chapter on page 294 for details on how to connect to the DMA channel trigger input. Configure the DMA channel as required - the source address of the DMA descriptor is PDM_RX_FIFO_RD register with the source address increment feature disabled and the source data length is word type (32-bits). The DMA channel can be used to transfer (TRIGGER_LEVEL + 1) words from the PDM_RX_FIFO_RD register whenever the trigger signal becomes high. The destination address configuration depends on the application requirements. See the DMA Controller (DW) chapter on page 91 for details on DMA channel configuration.
8. Enable the DMA trigger signal generation by setting the RX_REQ_EN bit in the PDM_TR_CTL register.

32.3.2 Interrupt Service Routine (ISR) Configuration

The code for the PDM interrupt service routine should have the following flow:

1. The events that triggered the interrupt can be found by reading the PDM_INTR_MASKED register in the ISR. All the bits that are set causes the interrupt event. The register value should also be in a variable “var”.
2. For each of the event bits that are set in PDM_INTR_MASKED, appropriate application level tasks can be executed. For example, the RX_TRIGGER event can be used for CPU-based data transfers if a DMA-based data transfer is not used. DMA transfers should use the tr_pdm_rx_req trigger signal (by setting the RX_REQ_EN bit in the PDM_TR_CTL register). The DMA trigger should not use the RX_TRIGGER interrupt event to reduce CPU usage for data transfer. The RX_OVERFLOW event can be used to take appropriate counter measures such as giving higher priority to PDM-PCM DMA channel. The RX_UNDERFLOW event typically indicates wrong data transfer logic in the application – either in the CPU-based data transfer code or in the DMA channel configuration used to transfer data.
3. After the event conditions have been processed, the “var” value read from PDM_INTR_MASKED should be written to the PDM_INTR register to clear the events that are set in the register. Due to the buffered write logic, the

PDM_INTR register should also be read after the write process to ensure the write process is completed in the slower peripheral clock domain.

32.3.3 Enabling / Disabling Streaming

The PDM-PCM conversion process starts after the STREAM_EN bit is set in the PDM_CMD register. Depending on the application needs, the streaming can be dynamically started and stopped using the STREAM_EN bit. Clear the Rx FIFO before starting the streaming process to reset the read/write pointers and FIFO state. The procedure to clear the FIFO is to write a ‘1’ to the CLEAR bit in PDM_RX_FIFO_CTL followed by writing a ‘0’ to the CLEAR bit. When the CLEAR bit is set, all the data entries in the Rx FIFO are cleared by resetting the internal read/write pointers. Read accesses to the PDM_RX_FIFO_RD and PDM_RX_FIFO_RD_SILENT registers are prohibited when the CLEAR bit is 1. Therefore, the CLEAR bit should be cleared before starting the streaming operation.

33. Universal Serial Bus (USB) Device Mode

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU USB block can act as a USB device that communicates with a USB host. The USB block is available as a fixed-function digital block in the PSoC 6 MCU. It supports full-speed communication (12 Mbps) and is designed to be compliant with the USB Specification Revision 2.0. USB devices can be designed for plug-and-play applications with the host and also support hot swapping. This chapter details the PSoC 6 MCU USB block and transfer modes. For details about the USB specification, see the USB Implementers Forum website.

33.1 Features

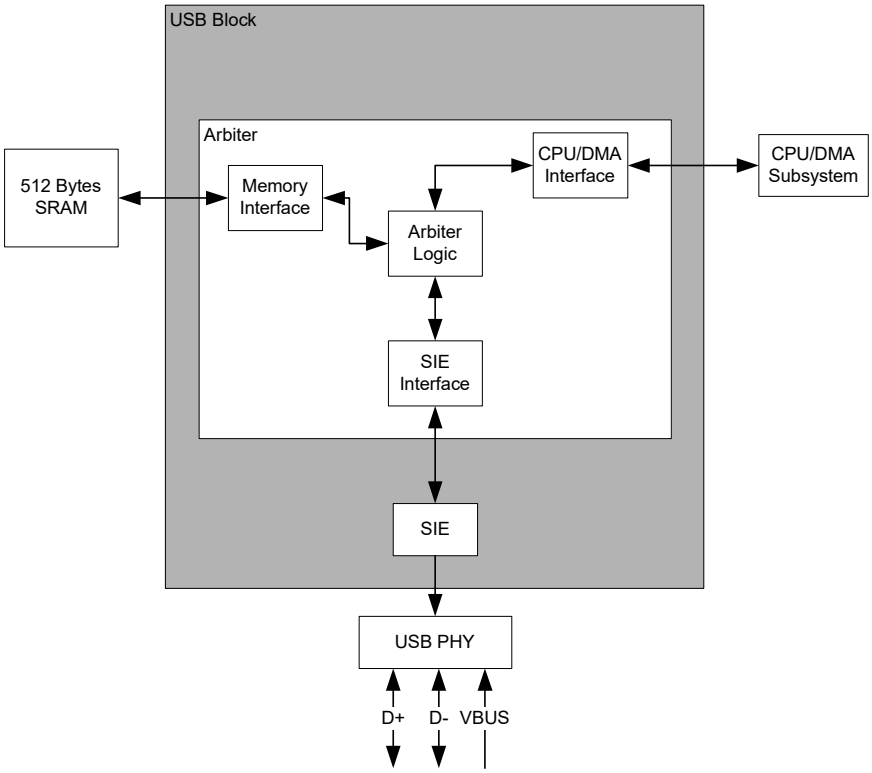
The USB in the PSoC 6 MCU has the following features:

- Complies with USB Specification 2.0
- Supports full-speed peripheral device operation with a signaling bit rate of 12 Mbps
- Supports eight data endpoints and one control endpoint
- Provides shared 512-byte buffer for data endpoints
- Provides dedicated 8-byte memory for control endpoint (EP0)
- Supports four types of transfers – bulk, interrupt, isochronous, and control
- Supports bus- and self-powered configurations
- Enables USB suspend mode for low power
- Supports three types of logical transfer modes:
  - No DMA mode (Mode 1)
  - Manual DMA mode (Mode 2)
  - Automatic DMA mode (Mode 3)
- Supports maximum packet size of 512 bytes using Mode 1 and Mode 2, and maximum packet size of 1023 bytes for isochronous transfer using Mode 3
- Provides integrated 22-Ω USB termination resistors on D+ and D– lines, and 1.5-kΩ pull-up resistor on the D+ line
- Supports USB 2.0 Link Power Management (LPM)
- Can be configured using the USB Device Configurator available with the ModusToolbox software

33.2 Architecture

Figure 33-1 illustrates the device architecture of the USB block in PSoC 6 MCUs. It consists of the USB Physical Layer (USB PHY), Serial Interface Engine (SIE), and the local 512-byte memory buffer.

Figure 33-1. USB Device Block Diagram



33.2.1 USB Physical Layer (USB PHY)

The USB PHY allows physical layer communication with the USB host through the D+, D–, and VBUS pins. It handles the differential mode communication with the host, VBUS detection, and monitoring events such as SE0 on the USB bus.

33.2.2 Serial Interface Engine (SIE)

The SIE handles the decoding and creating of data and control packets during transmit and receive. It decodes the USB bit streams into USB packets during receive, and creates USB bit streams during transmit. The following are the features of the SIE:

- Conforms to USB Specification 2.0
- Supports one device address
- Supports eight data endpoints and one control endpoint
- Supports interrupt trigger events for each endpoint
- Integrates an 8-byte buffer in the control endpoint

The registers for the SIE are mainly used to configure the data endpoint operations and the control endpoint data buffers. This block also controls the interrupt events available for each endpoint.

33.2.3 Arbiter

The Arbiter handles access of the SRAM memory by the endpoints. The SRAM memory can be accessed by the CPU, DMA, or SIE. The arbiter handles the arbitration between the CPU, DMA, and SIE. The arbiter consists of the following blocks:

- SIE Interface Module
- CPU/DMA Interface
- Memory Interface
- Arbiter Logic

The arbiter registers are used to handle the endpoint configurations, read address, and write address for the endpoints. It also configures the logical transfer type required for each endpoint.

33.2.3.1 SIE Interface Module

This module handles all the transactions with the SIE. The SIE reads data from the SRAM memory and transmits to the host. Similarly, it writes the data received from the host to the SRAM memory. These requests are registered in the SIE Interface module and are handled by this module.

33.2.3.2 CPU/DMA Interface Block

This module handles all transactions with the CPU and DMA. The CPU requests for reads and writes to the SRAM memory for each endpoint. These requests are registered in this interface and are handled by the block. When the DMA is configured, this interface is responsible for all transactions between the DMA and USB. The block supports the DMA request line for each data endpoint. The behavior of the DMA depends on the type of logical transfer mode configured in the configuration register.

33.2.3.3 Memory Interface

The memory interface is used to control the interface between the USB and SRAM memory unit. The maximum memory size supported is 512 bytes organized as 256 × 16-bit memory unit. This is a dedicated memory for the USB. The memory access can be requested by the SIE or by the CPU/DMA. The SIE Interface block and the CPU/DMA Interface block handle these requests.

33.2.3.4 Arbiter Logic

This is the main block of the arbiter. It is responsible for arbitrations for all the transactions that happen in the arbiter. It arbitrates the CPU, DMA, and SIE access to the memory unit and the registers. This block also handles memory management, which is either 'Manual' or 'Automatic'. In Manual memory management, the read and write address manipulations are done by the firmware. In Automatic management, all the memory handling is done by the block itself. This block takes care of the buffer size allocation. It also handles common memory area. This block also handles the interrupt requests for each endpoint.

33.3 Operation

33.3.1 USB Clocking Scheme

The USB device block should be clocked at 48 MHz with an accuracy of ±0.25%. In the PSoC 6 MCU, CLK_HF3 is the clock source. The USB device block also requires a 100-kHz peripheral clock for USB bus reset timing. The required USB clock can be generated using one of the following clocking schemes:

- IMO (trimmed with USB) -> PLL -> CLK_HF3
- ECO (with the required accuracy) -> FLL -> CLK_HF3
- ECO (with the required accuracy) -> PLL -> CLK_HF3
- Use external clock (EXTCLK) with the required accuracy

33.3.2 USB PHY

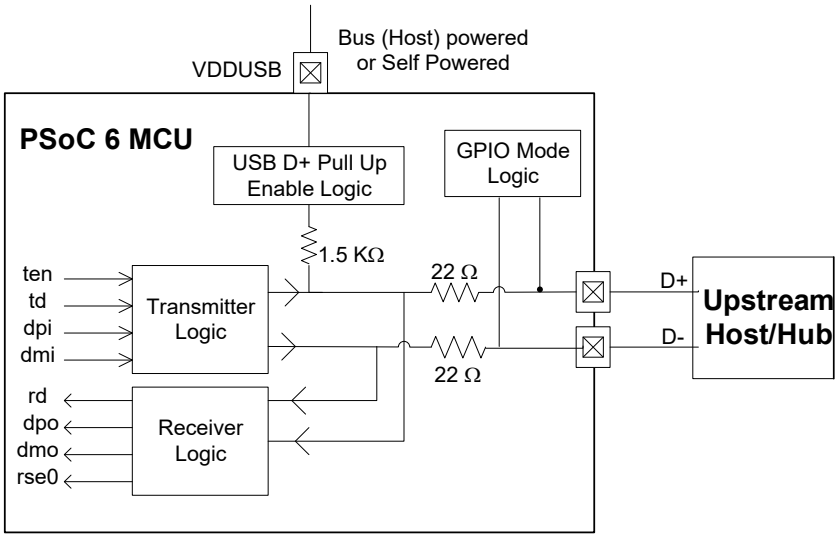
The USB includes the transmitter and receiver (transceiver), which corresponds to the USB PHY. Figure 33-2 shows the PHY architecture. The USB PHY also includes the pull-up resistor on the D+ line to identify the device as full-speed type to the host. The PHY integrates the 22-Ω series termination resistors on the USB line. The signal between the USB device and the host is a differential signal. The receiver receives the differential signal from the host and converts it to a single-ended signal for processing by the SIE. The transmitter converts the single-ended signal from the SIE to the differential signal, and transmits it to the host. The differential signal is given to the upstream devices at a nominal voltage range of 0 V to 3.3 V.

33.3.2.1 Power Scheme

The USB PHY is powered by the VBUS power pad of the PSoC 6 MCU. The VBUS pad can be driven either by the host VBUS (bus-powered) or an external power supply (self-powered).

The USB PHY needs a nominal voltage of 3.3 V for its communication with the host.

Figure 33-2. USB PHY Architecture



33.3.2.2 VBUS Detection

USB devices can either be bus-powered (power sourced from the host) or self-powered (power sourced from an external power supply). The VDDUSB power pad pin powers the USB PHY and USB I/Os (D+ and D– pins). The presence of VBUS can be detected using the following steps:

1. Enable the interrupt on VDDUSB power pad. For this write '1' to the VDDIO_ACTIVE[5] bit of VDD_INTR_MASK register.
2. VDDIO_ACTIVE[5] bit of the supply detection interrupt register (VDD_INTR) is set to '1' whenever a change to supply is detected. Clear the interrupt cause by writing '1' to the bitfield.
3. Check the status of the VDD_ACTIVE[5] bit of the external power supply detection register (VDD_ACTIVE). The bit is set to '1' when there is supply and '0' when there is no supply.

33.3.2.3 USB D+ Pin Pull-up Enable Logic

When a USB device is self-powered, the USB specification warrants that the device enable the pull-up resistor on its D+ pin to identify itself as a full-speed device to the host. When the host VBUS is removed, the device should disable the pull-up resistor on the D+ line to not back power the host. The USB PHY includes an internal 1.5-kΩ pull-up resistor on the D+ line to indicate to the host that the PSoC 6 MCU is a full-speed device. The pull-up resistor can be enabled or disabled by configuring the DP_UP_EN bit in the USBLPM_POWER_CTRL register.

33.3.2.4 Transmitter and Receiver Logic

The transceiver block transmits and receives USB differential signals with an upstream device, and includes the USB D+ pull-up resistor used to maintain an idle state on the bus. Output data is differentially transmitted to upstream devices at a nominal voltage of 3.3 V. The differential inputs received from upstream devices are converted into single-ended data and sent to the core logic at a nominal voltage of 1.8 V. The D+ and D– pins are terminated with 22-Ω resistors to meet the USB impedance specification.

33.3.2.5 GPIO Mode Logic

The D+ and D– pins can be used either as GPIO pins or USB I/O pins. This is controlled by the IOMODE bit of the USBDEV_USBIO_CR1 register. This bit should be set HIGH for GPIO functionality and LOW for USB operation.

33.3.2.6 Link Power Management (LPM)

The USB PHY supports link power management (LPM), which is similar to the suspend mode, but has transitional latencies in tens of microseconds between power states, compared to the greater than 20 ms latency associated with suspend/resume modes. For more details on LPM, refer to the USB 2.0 specification. The following features are supported for LPM.

- The LPM_CTL register should be configured to enable/disable LPM, type of response when LPM is enabled, and the response when a sub PID other than the LPM token is received from the host.
- The LPM_STAT register stores the values of the Best Effort Service Latency (BESL) and the remote wakeup feature as sent by the host. The firmware should read this register on the LPM interrupt event and enter the appropriate low-power mode (Deep Sleep or Sleep) based on the BESL value from the host.

33.3.3 Endpoints

The SIE and arbiter support eight data endpoints (EP1 to EP8) and one control endpoint (EP0). The data endpoints share the SRAM memory area of 512 bytes. The endpoint memory management can be either manual or automatic. The endpoints are configured for direction and other configuration using the SIE and arbiter registers. The endpoint read address and write address registers are accessed through the arbiter.

The endpoints can be individually made active. In the Auto Management mode, the register EP_ACTIVE is written to control the active state of the endpoint. The endpoint activation cannot be dynamically changed during runtime. In Manual Memory Management mode, the firmware decides the memory allocation, so it is not required to specify the active endpoints. The EP_ACTIVE register is ignored during the manual memory management mode. The EP_TYPE register is used to control the transfer direction (IN, OUT) for the endpoints. The control endpoint has separate eight bytes for its data (EP0_DR registers).

33.3.4 Transfer Types

The PSoC 6 MCU USB supports full-speed transfers and is compliant with the USB 2.0 specification. It supports four types of transfers:

- Interrupt Transfer
- Bulk Transfer
- Isochronous Transfer
- Control Transfer

For further details about these transfers, refer to the USB 2.0 specification.

33.3.5 Interrupt Sources

The USB device block generates 14 interrupts to the CPU. These interrupts are mapped to three general-purpose interrupt lines – INTR_LO, INTR_MED, and INTR_HI. Each of these three interrupt lines has an associated status register, which identifies the cause of the interrupt event. These are the USBLPM_INTR_CAUSE_LO, USBLPM_INTR_CAUSE_MED, and USBLPM_INTR_CAUSE_HI registers. The routing of these interrupts is controlled by the USBLPM_INTR_LVL_SEL register fields.

The following events generate an interrupt on one of the three interrupt lines:

- USB start of frame (SOF) event

- USB bus reset event
- Eight data endpoint (EP1 – EP8) interrupt events
- Control endpoint (EP0) interrupt event
- Link power management (LPM) event
- Resume event
- Arbiter Interrupt event

33.3.5.1 USB Start of Frame (SOF) Event

The SOF interrupt is generated upon receiving an SOF packet from the USB host. The SOF interrupt is enabled using the SOF_INTR_MASK bitfield in the USBLPM_INTR_SIE_MASK register.

- The SOF interrupt status is reflected in the SOF_INTR status bit in the USBLPM_INTR_SIE status register.
- The SOF interrupt status is also available in the SOF_INTR_MASKED bit of the USBLPM_INTR_SIE_MASKED register – this bit is the logical AND of the corresponding SOF bits in the USBLPM_INTR_SIE_MASK register and the USBLPM_INTR_SIE register.
- If there is no SOF interrupt for 3 ms, the USB device goes into SUSPEND state.

33.3.5.2 USB Bus Reset Event

The USB bus reset interrupt is generated when a USB bus reset condition occurs. The bus reset interrupt is enabled by setting the BUS_RESET_INTR_MASK bit in the USBLPM_INTR_SIE_MASK register.

- The bus reset interrupt status is reflected in the BUS_RESET_INTR bit in the USBLPM_INTR_SIE status register.
- The bus reset interrupt status is also available in the BUS_RESET_INTR_MASKED bit of the USBLPM_INTR_SIE_MASKED register – this bit is the logical AND of the corresponding bus reset bits in the USBLPM_INTR_SIE_MASK register and the USBLPM_INTR_SIE register.
- The SIE logic triggers the counter to start running on the divided version of CLK_PERI when an SE0 condition is detected on the USB bus. When the counter reaches the count value configured in the USBDEV_BUS_RST_CNT register, the bus reset interrupt is triggered. Typically, divided CLK_PERI is set to 100 kHz and USBDEV_BUS_RST_CNT is set to ‘10’.

33.3.5.3 Data Endpoint Interrupt Events

These are eight interrupt events corresponding to each data endpoint (EP1-EP8). Each of the endpoint interrupt events can be enabled/disabled by using the corresponding bit in the USBDEV_SIE_EP_INT_EN register. The interrupt status of each endpoint can be known by reading the USBDEV_SIE_EP_INT_SR status register. An endpoint whose interrupt is enabled can trigger the interrupt on the following events:

- Successful completion of an IN/OUT transfer
- NAK-ed IN/OUT transaction if the corresponding NAK_INT_EN bit in the SIE_EPx_CR0 register is set
- When there is an error in the transaction, the ERR_IN_TXN bit in the SIE_EPx_CR0 register is set and interrupt is generated.
- If the STALL bit in SIE_EPx_CR0 is set, then stall events can generate interrupts. This stall event can occur if an OUT packet is received for an endpoint whose mode bits in SIE_EPx_CR0 are set to ACK_OUT or if an IN packet is received with mode bits set to ACK_IN.

33.3.5.4 Control Endpoint Interrupt Event

The interrupt event corresponding to the control endpoint (EP0) is generated under the following events:

- Successful completion of an IN/OUT transfer
- When a SETUP packet is received on the control endpoint

The EP0 interrupt is setup using the EP0_INTR_SET bit of the USBLPM_INTR_SIE_SET register.

33.3.5.5 Link Power Management (LPM) Event

Generated whenever the LPM token packet is received. The LPM interrupt is enabled by setting the LPM_INTR_MASK bit in the USBLPM_INTR_SIE_MASK register. The LPM interrupt status is reflected in the LPM_INTR status bit in the USBLPM_INTR_SIE status register.

The LPM interrupt status is also available in the LPM_INTR_MASKED bit of the USBLPM_INTR_SIE_MASKED register; this bit is the logical AND of the corresponding LPM bits in the USBLPM_INTR_SIE_MASK and USBLPM_INTR_SIE registers.

The firmware needs to read the USBLPM_LPM_STAT register to read the BESL remote wakeup values and appropriately enter the desired low-power mode. Enter the low-power mode in the main code. The exit from LPM is identical to the resume event wakeup in the case of suspend mode.

33.3.5.6 RESUME Interrupt

The RESUME interrupt is asserted by the USB block when it detects ‘0’ on the DP pad. The RESUME interrupt is enabled by setting the RESUME_INTR_MASK bitfield of the USBLPM_INTR_SIE_MASK register.

- The RESUME interrupt status is reflected in the RESUME_INTR status bit in the USBLPM_INTR_SIE status register.
- The RESUME interrupt status is also available in the RESUME_INTR_MASKED bit of the USBLPM_INTR_SIE_MASKED register – this bit is the logical AND of the corresponding RESUME bits in the USBLPM_INTR_SIE_MASK register and the USBLPM_INTR_SIE register.

The RESUME interrupt is an Active mode interrupt and not available in Deep Sleep or Hibernate mode.

33.3.5.7 Arbiter Interrupt Event

The arbiter interrupt can arise from five possible sources. Each interrupt source is logically ANDed with its corresponding ENABLE bit and the results are logically ORed to result in a single arbiter interrupt event.

The arbiter interrupt event can arise under any of the following five scenarios:

- DMA Grant
- IN Buffer Full
- Buffer Overflow
- Buffer Underflow
- DMA Termin

DMA Grant

This event is applicable in Mode 2 or Mode 3. (See Logical Transfer Modes on page 456 for details on DMA modes). This event is triggered when the DMA controller pulses the Burstend signal corresponding to that endpoint, for which a DMA request had been raised to the DMA controller earlier. The request may have been either a manual DMA request or an automatic arbiter DMA request. A common grant status exists for both modes of requests. This grant status indicates completion of the DMA transaction. This status indication can be used by firmware to determine when the next manual DMA request can be raised. Multiple requests raised for the same endpoint before the DMA grant status is set will be dropped by the block. Only the first of multiple requests will be transmitted to DMA controller.

IN Buffer Full

This event status can occur in any of the DMA modes (Mode 1, 2, or 3) and is applicable only for IN endpoints.

- Store and Forward Mode (Modes 1 and 2): This status is set when the entire packet data is transferred to the local memory. The check is that data written for the particular endpoint is equal to the programmed byte count for that endpoint in the USBDEV_SIE_EPx_CNT0 and USBDEV_SIE_EPx_CNT1 registers.
- Cut Through Mode (Mode 3): In this mode, the IN buffer full status is set when the IN endpoint's dedicated buffer is filled with the packet data. The size of this buffer is determined by the value programmed in bits [3:0] of the USBDEV_BUF_SIZE register. This status indication can be used to determine when the mode value in the USBDEV_SIE_EPx_CR0 register can be programmed to acknowledge an IN token for that endpoint.

Buffer Overflow

This event status is active only in the Cut Through Mode (Mode 3). The following conditions can cause this bit to be set:

- Data overflow on the endpoint dedicated buffer space
  - In an IN endpoint, the dedicated buffer can overflow if the DMA transfer writes a larger number of bytes than the space available in the dedicated buffer. Until an IN token is received for that endpoint, it cannot use the common buffer area, hence resulting in an overflow of data. The possible causes of this buffer overflow can be incorrect programming of either the DMA transfer descriptor transfer size or the USBDEV_BUF_SIZE register.
  - In an OUT endpoint, the dedicated buffer can overflow if two OUT transactions occur consecutively. The data from the previous transaction is still present in the common area and the current ongoing transaction fills up the OUT endpoint's dedicated buffer space and overflows. The possible causes of this overflow can be the overall DMA bandwidth constraint due to other DMA transactions or reduced size of the dedicated OUT buffer size.
- Common area data overflow
  - In an IN endpoint, the common area overflow occurs when the DMA transfer writes a larger number of bytes than the space available in the common area. This situation may arise due to incorrect programming of either the DMA transfer descriptor transfer size or the USBDEV_DMA_THRESH and USBDEV_DMA_THRESH_MSB registers.
  - In an OUT endpoint, the common area overflow occurs when the data written to the common area has not yet been read and new data overwrites the existing data.

Buffer Underflow

This event is applicable only in the Cut Through mode (Mode 3). This underflow condition can occur only for an IN endpoint. The underflow condition can occur either in the dedicated buffer space or common buffer space. The underflow condition on the dedicated buffer space can either be due to the reduced dedicated IN buffer size or DMA bandwidth constraint. The underflow condition can occur on the common buffer space due to DMA bandwidth constraint and/or lower DMA channel priority.

DMA Termin

This status is set when USBDEV generates a dma_termin signal to indicate the total programmed/ received bytes that are written/read by the DMA controller. This status indication can be used by the firmware to reprogram the IN/OUT endpoint for the next transfer. For an OUT endpoint, this indicates that the OUT packet data is available in the system memory for further processing by the application.

33.3.6 DMA Support

Each of the eight data endpoints has one DMA channel available to transfer data between the endpoint buffer and the SRAM memory. The USB generates the DMA request signals (usb.dma_req[7:0]) to the respective DMA channels to initiate the data transfer for the endpoint. This goes to the Trigger Group 13 multiplexer as input triggers that can be routed to one of the trigger inputs of the DMA block. The Burstend signals from the DMA channel to the correspond-

ing endpoint is routed using the Trigger Group 9 multiplexer. For more details, see the DMA Controller (DW) chapter on page 91 and Trigger Multiplexer Block chapter on page 294.

33.4 Logical Transfer Modes

The USB block in PSoC 6 MCUs supports two types of logical transfers. The logical transfers can be configured using the register setting for each endpoint. Any of the logical transfer methods can be adapted to support the three types of data transfers (Interrupt, Bulk, and Isochronous) mentioned in the USB 2.0 specification. The control transfer is mandatory in any USB device.

The logical transfer mode is a combination of memory management and DMA configurations. The logical transfer modes are related to the data transfer within the USB (to and from the SRAM memory unit for each endpoint). It does not represent the transfer methods between the device and the host (the transfer types specified in the USB 2.0 specification).

The USB supports two basic types of transfer modes:

- Store and Forward mode
  - Manual Memory Management with No DMA Access (Mode 1)
  - Manual Memory Management with Manual DMA Access (Mode 2)
- Cut Through mode
  - Automatic Memory Management with Automatic DMA Access (Mode 3)

Table 33-1 gives a comparison of the two transfer modes.

Table 33-1. USB Transfer Modes

Feature	Store and Forward Mode	Cut Through Mode
SRAM Memory Usage	Requires more memory	Requires less memory
SRAM Memory Management	Manual	Auto
SRAM Memory Sharing	512 bytes of SRAM shared between endpoints. Sharing is done by firmware.	Each endpoint is allocated a lesser share of memory automatically by the block. The remaining memory is available as “common area.” This common area is used during the transfer.
IN Command	Entire packet present in SRAM memory before the IN command is received.	Memory filled with data only when SRAM IN command is received. Data is given to host when enough data is available (based on DMA configuration). Does not wait for the entire data to be filled.
OUT Command	Entire packet is written to SRAM memory on OUT command. After entire data is available, it is copied from SRAM memory to the USB device.	Waits only for enough bytes (depends on DMA configuration) to be written in SRAM memory. When enough bytes are present, it is immediately copied from SRAM memory to the USB device.
Transfer of Data	Data is transferred when all bytes are written to the memory.	Data is transferred when enough bytes are available. It does not wait for the entire data to be filled.
Types Based on DMA	No DMA mode Manual DMA mode	Only Auto DMA mode
Supported Transfer Types	Ideal for interrupt and bulk transfers	Ideal for Isochronous transfer

Every endpoint has a set of registers that need to be handled during the modes of operation, as detailed in Table 33-2.

Table 33-2. Endpoint Registers

Register	Comment	Content	Usage
USBDEV_ARB_RWx_WA	Endpoint Write Address Register	Address of the SRAM	This register indicates the SRAM location to which the data in the data register is to be written.
USBDEV_ARB_RWx_RA	Endpoint Read Address Register	Address of the SRAM	This register indicates the SRAM location from which the data must be read and stored to the data register.
USBDEV_ARB_RWx_DR	Endpoint Data Register	8-Bit Data	Data register is read/written to perform any transaction. IN command: Data written to the data register is copied to the SRAM location specified by the WA register. After write, the WA value is automatically incremented to point to the next memory location. OUT command: Data available in the SRAM location pointed by the USBDEV_ARB_RWx_RA register is read and stored to the DR. When the DR is read, the value of USBDEV_ARB_RWx_RA is automatically incremented to point to the next SRAM memory location that must be read.
USBDEV_SIE_EPx_CNT0 and USBDEV_SIE_EPx_CNT1	Endpoint Byte Count Register	Number of Bytes	Holds the number of bytes that can be transferred. IN command: Holds the number of bytes to be transferred to host. OUT command: Holds the maximum number of bytes that can be received. The firmware programs the maximum number of bytes that can be received for that endpoint. The SIE updates the register with the number of bytes received for the endpoint.
“Mode” bits in USBDEV_SIE_EPx_CR0	Mode Values	Response to the Host	Controls how the USB device responds to the USB traffic and the USB host. Some examples of modes are ACK, NAK, and STALL.

In Manual memory management, the endpoint read and endpoint write address registers are updated by the firmware. So the memory allocation can be done by the user. The memory allocation decides which endpoints are active; that is, you can decide to share the 512 bytes for all the eight endpoints or a lesser number of endpoints.

In Automatic memory management, the endpoint read and endpoint write address registers are updated by the USB block. The block assigns memory to the endpoints that are activated using the USBDEV_EP_ACTIVE register. The size of memory allocated depends on the value in the USBDEV_BUF_SIZE register. The remaining memory, after allocation, is called the common area memory and is used for data transfer.

In all of these modes, either the 8-bit endpoint data register or the 16-bit endpoint data register can be used to read/write to the endpoint buffer. While transferring data to the 16-bit data registers, ensure that the corresponding SRAM memory address locations are also 16-bit aligned.

In the following text, the algorithm for the IN and OUT transaction for each mode is discussed. An IN transaction is when the data is read by the USB host (for example, PC). An OUT transaction is when the data is written by the USB host to the USB device. The choice of using the DMA and memory management can be configured using the USBDEV_ARB_EPx_CFG register.

33.4.1 Manual Memory Management with No DMA Access

All operations in this mode are controlled by the CPU and works in a store-and-forward operation mode. An entire packet is transferred to the memory and a mode bit (such as ACK IN or ACK OUT) is set by the CPU. The SIE responds appropriately to an IN/OUT token received from the host. All memory space management is handled by the CPU.

Figure 33-3. No DMA Access IN Transaction

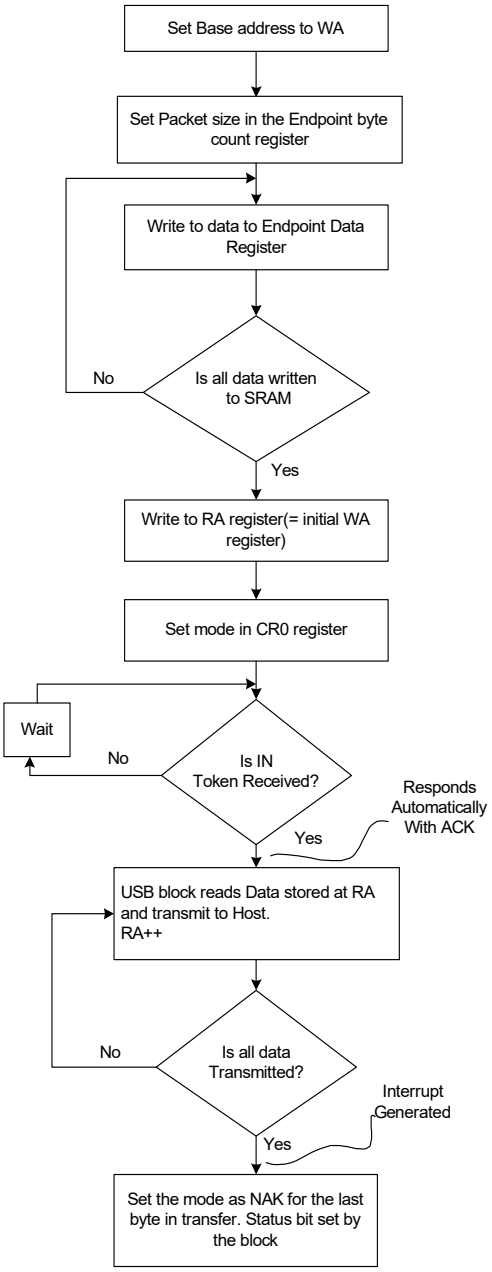
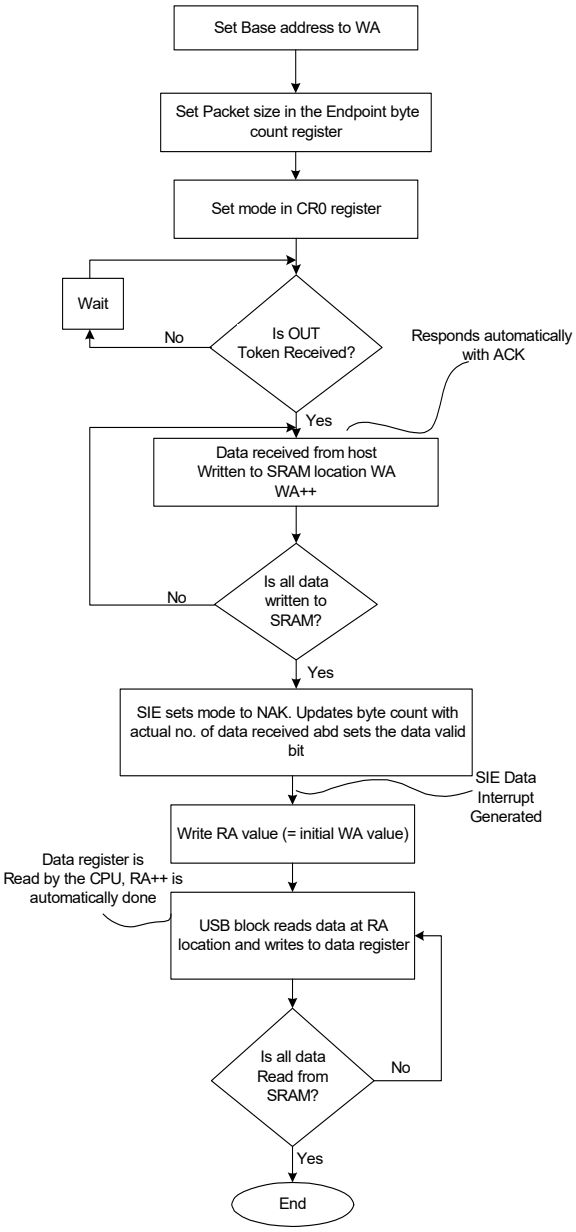


Figure 33-4. No DMA Access OUT Transaction



33.4.2 Manual Memory Management with DMA Access

This mode is similar to the No DMA Access except that write/read of packets is performed by the DMA. A DMA request for an endpoint is generated by setting the DMA_CFG bit in the USBDEV_ARB_EPx_CFG register. When the DMA service is granted and is done (DMA_GNT), an arbiter interrupt can be programmed to occur. The transfer is done using a single DMA cycle or multiple DMA cycles. After completion of every DMA cycle, the arbiter interrupt (DMA_GNT) is generated. Similarly, when all the data bytes (programmed in byte count) are written to the memory, the arbiter interrupt occurs and the IN_BUF_-FULL bit is set.

Figure 33-5 and Figure 33-6 show the flow charts for manual DMA IN and OUT transactions respectively.

Figure 33-5. Manual DMA IN Transaction

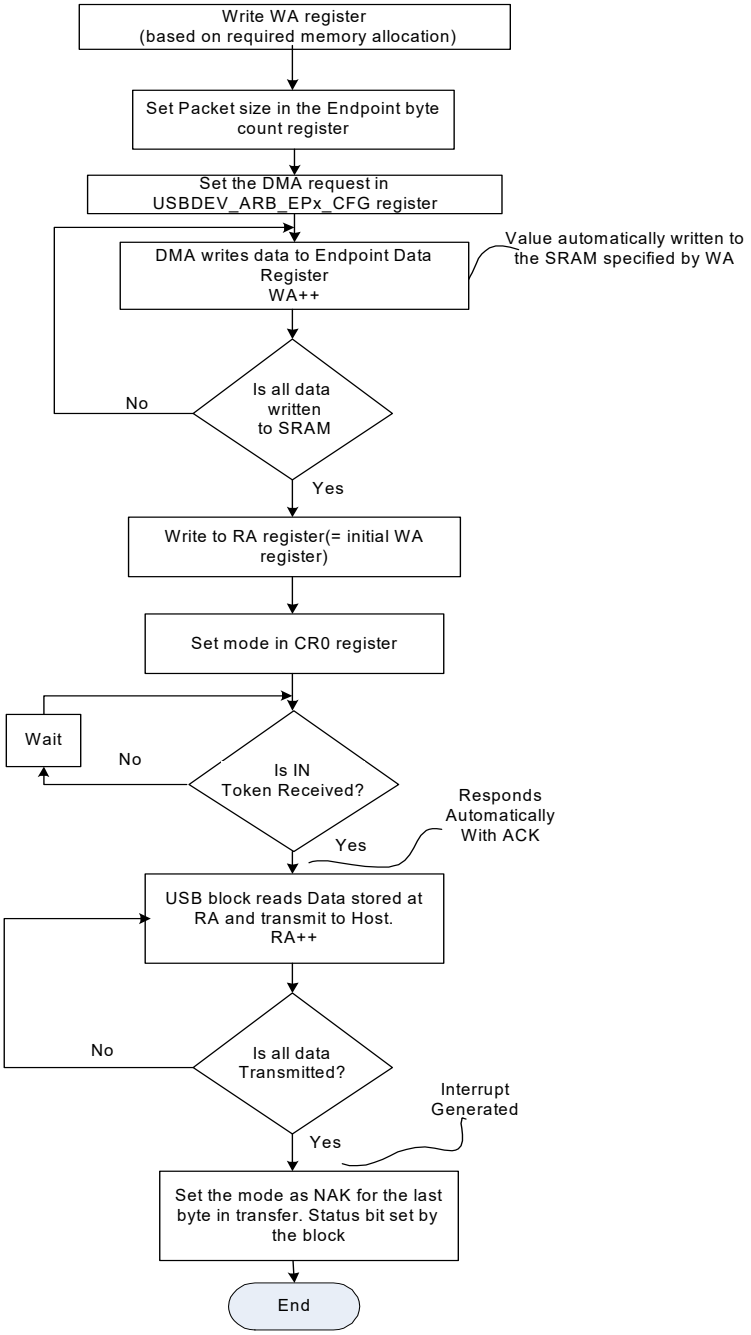
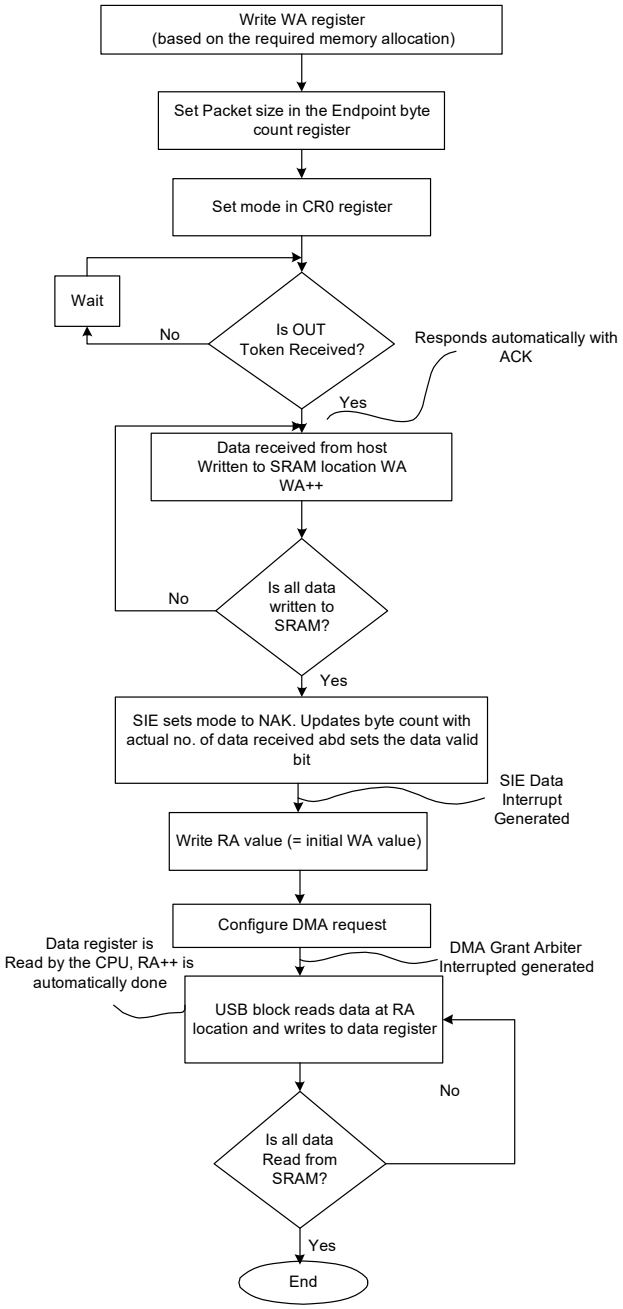


Figure 33-6. Manual DMA OUT Transaction



33.4.3 Automatic DMA Mode

This is the Automatic memory management mode with auto DMA access. The CPU programs the initial buffer size requirement for IN/OUT packets and informs the arbiter block of the endpoint configuration details for the particular application. The block then controls memory partitioning and handling of all memory pointers. During memory allocation, each active IN endpoint (set by the USBDEV_EP_ACTIVE and USBDEV_EP_TYPE registers) is allocated a small amount of memory configured using the USBDEV_BUF_-SIZE register (32 bytes for each of the eight endpoints). The remaining memory (256 bytes) is left as common area and is common for all endpoints.

In this mode, the memory requirement is less and it is suitable for full-speed isochronous transfers up to 1023 bytes.

When an IN command is sent by the host, the device responds with the data present in the dedicated memory area for that endpoint. It simultaneously issues a DMA request for more data for that EP. This data fills up the common area. The device does not wait for the entire packet of data to be available. It waits only for the (USBDEV_DMA_-THRES_MSB, USBDEV_DMA_THRES) number of data available in the SRAM memory and begins the transfer from the common area.

Similarly, when an OUT command is received, the data for the OUT endpoint is written to the common area. When some data (greater than USBDEV_DMA_THRES_MSB, USBDEV_DMA_THRES) is available in the common area, the arbiter block initiates a DMA request and the data is immediately written to the device. The device does not wait for the common area to be filled.

This mode requires configuration of the USBDEV_DMA_THRES and USBDEV_DMA_THRES_MSB registers to hold the number of bytes that can be transferred in one DMA transfer (32 bytes). Similarly, the burst count of the DMA should always be equal to the value set in the USBDEV_DMA_THRES registers. Apart from the DMA configuration, this mode also needs the configuration of the USBDEV_BUF_SIZE for the IN and the OUT buffers and the USBDEV_EP_ACTIVE and the USBDEV_EP_TYPE registers.

Each DMA channel has two descriptors and both of them are used in this mode. Each descriptor is considered as a data chunk of 32 bytes and it executes according to the trigger mechanism. The descriptors are chained and hence 64 bytes can be transferred without firmware interaction. When both descriptors complete the endpoint DMA done interrupt and the DMA error interrupt triggers (due to the lack of data to transfer). The descriptors are updated to advance the source SRAM (IN endpoint) or destination SRAM (OUT endpoint) pointer locations and then enabled again. This sequence continues till all data is transferred.

The steps for IN and OUT transactions using automatic DMA mode are shown in Figure 33-7 and Figure 33-8.

Figure 33-7. IN Transaction using Automatic DMA Mode

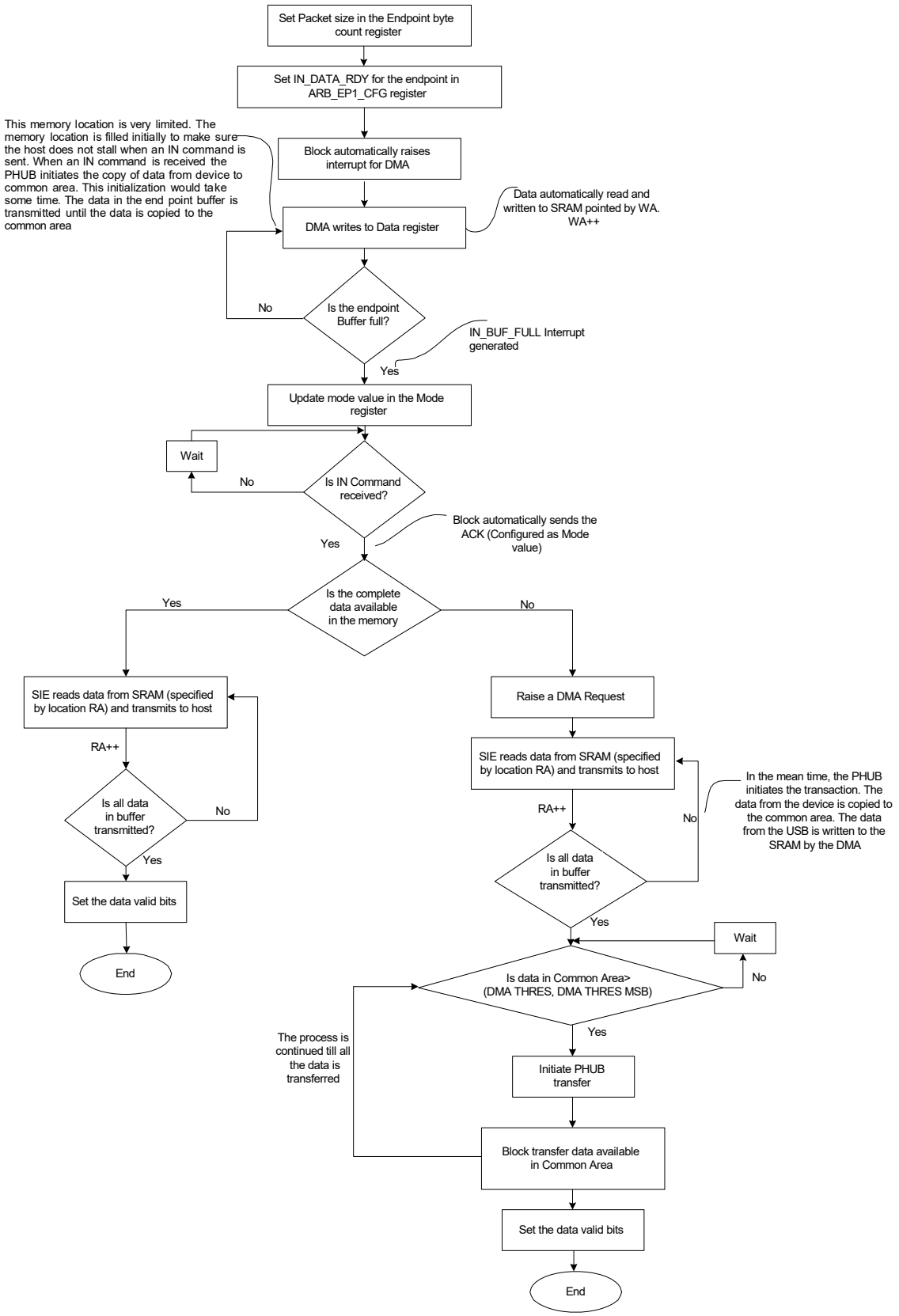
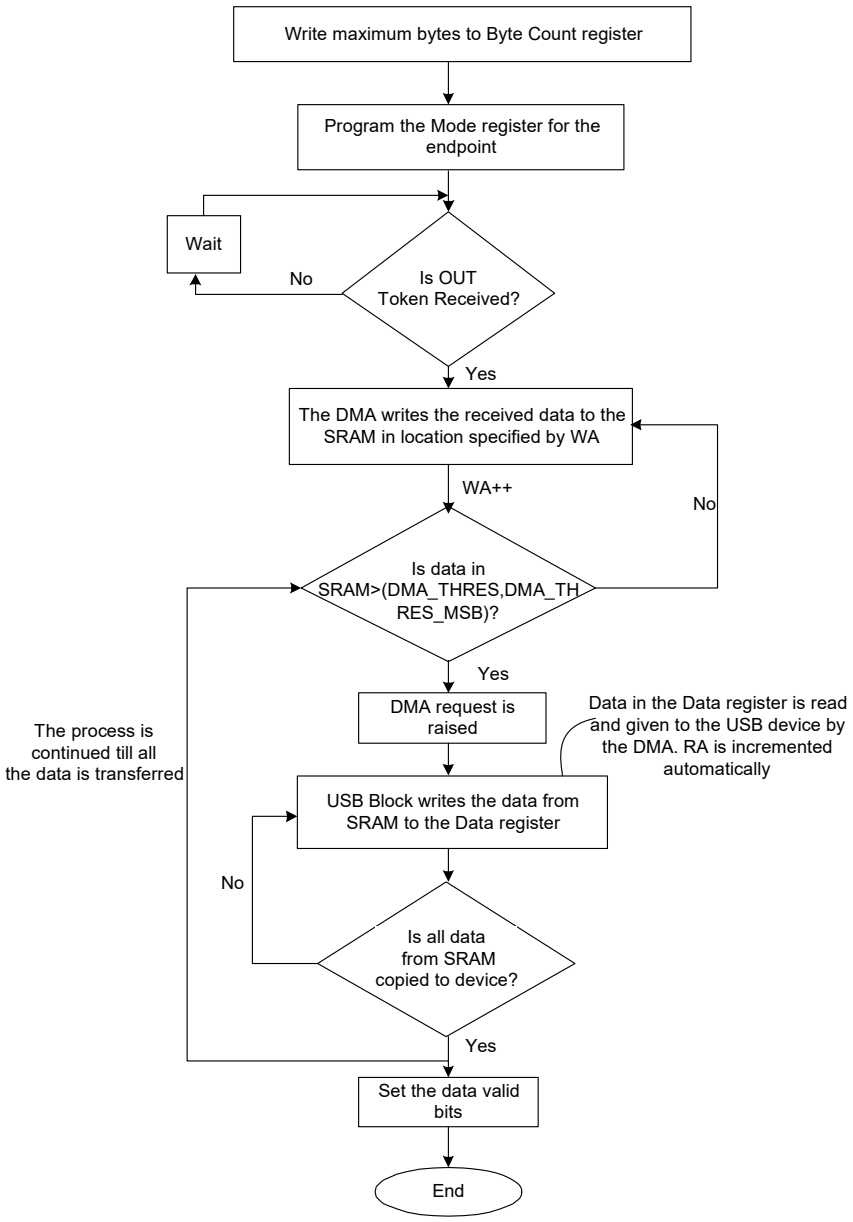


Figure 33-8. OUT Transaction using Automatic DMA Mode



33.4.4 Control Endpoint Logical Transfer

The control endpoint has a special logical transfer mode. It does not share the 512 bytes of memory. Instead, it has a dedicated 8-byte register buffer (USBDEV_EP0_DRx registers). The IN and OUT transaction for the control endpoint is detailed in the following figures.

Figure 33-9. Control Endpoint IN Transaction

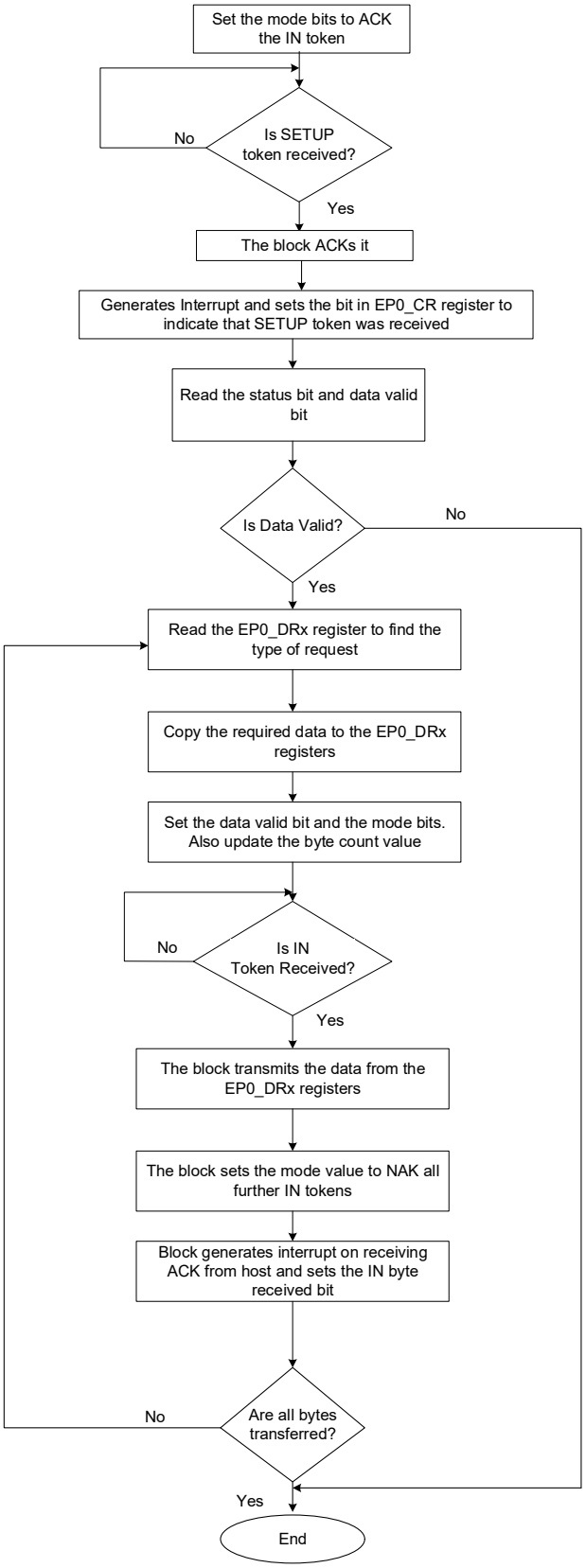
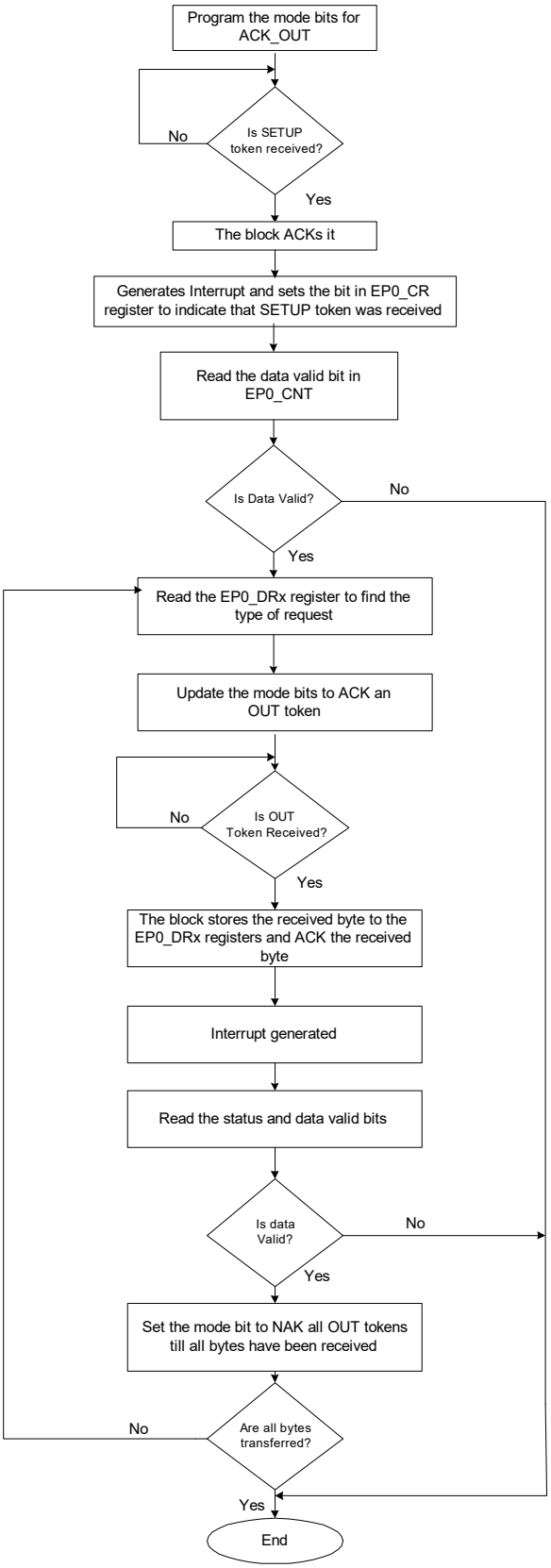


Figure 33-10. Control Endpoint OUT Transaction



33.5 USB Power Modes

The USB supports two modes of operation:

- Active mode: In this mode, the USB is powered up and clocks are turned on.
- Low-power (Deep Sleep) mode: In this mode, all clocks except the low-frequency clock are turned off.

Before entering low-power mode, the firmware should enable the GPIO interrupt (falling edge interrupt) on the D+ pin. The USB suspend mode can be determined by monitoring the SOF interrupt; this means, if there is no SOF interrupt from the USB for more than 3 ms, the USB goes into suspend mode and the block can be put into low-power mode by the firmware.

If there is any activity on the USB bus, D+ will be pulled low, which will cause a CPU interrupt. This interrupt can be used to wake up the USB device.

33.6 USB Device Registers

Name	Description
USBDEV_EP0_DR	Control endpoint EP0 data register
USBDEV_CR0	USB control 0 register
USBDEV_CR1	USB control 1 register
USBDEV_SIE_EP_INT_EN	USB SIE data endpoint interrupt enable register
USBDEV_SIE_EP_INT_SR	USB SIE data endpoint interrupt status
USBDEV_SIE_EPx_CNT0	Non-control endpoint count register
USBDEV_SIE_EPx_CNT1	Non-control endpoint count register
USBDEV_SIE_EPx_CR0	Non-control endpoint's control register
USBDEV_USBIO_CR0	USBIO control 0 register
USBDEV_USBIO_CR2	USBIO control 2 register
USBDEV_USBIO_CR1	USBIO control 1 register
USBDEV_DYN_RECONFIG	USB dynamic reconfiguration register
USBDEV_SOF0	Start of frame register
USBDEV_SOF1	Start of frame register
USBDEV_OSCLK_DR0	Oscillator lock data register 0
USBDEV_OSCLK_DR1	Oscillator lock data register 1
USBDEV_EP0_CR	Endpoint0 control register
USBDEV_EP0_CNT	Endpoint0 count register
USBDEV_ARB_RWx_WA	Endpoint write address value register. Pointer value increments by 1 when accessed by CPU/debugger
USBDEV_ARB_RWx_WA_MSB	Endpoint write address value register
USBDEV_ARB_RWx_RA	Endpoint read address value register. Pointer value increments by 1 when accessed by CPU/debugger
USBDEV_ARB_RWx_RA_MSB	Endpoint read address value register
USBDEV_ARB_RWx_DR	Endpoint data register
USBDEV_BUF_SIZE	Dedicated endpoint buffer size register
USBDEV_EP_ACTIVE	Endpoint active indication register
USBDEV_EP_TYPE	Endpoint type (IN/OUT) indication register
USBDEV_ARB_EPx_CFG	Endpoint configuration register
USBDEV_ARB_EPx_INT_EN	Endpoint interrupt enable register
USBDEV_ARB_EPx_SR	Endpoint interrupt enable register
USBDEV_ARB_CFG	Arbiter configuration register
USBDEV_USB_CLK_EN	USB block clock enable register

Name	Description
USBDEV_ARB_INT_EN	Arbiter interrupt enable register
USBDEV_ARB_INT_SR	Arbiter interrupt status register
USBDEV_CWA	Common area write address register
USBDEV_CWA_MSB	Endpoint read address value register
USBDEV_DMA_THRES	DMA burst / threshold configuration register
USBDEV_DMA_THRES_MSB	DMA burst / threshold configuration register
USBDEV_BUS_RST_CNT	Bus reset count register
USBDEV_MEM_DATA	Data register
USBDEV_SOF16	Start of frame register
USBDEV_OSCLK_DR16	Oscillator lock data register
USBDEV_ARB_RWx_WA16	Endpoint write address value register. Pointer value increments by 2 when accessed by CPU/debugger
USBDEV_ARB_RWx_RA16	Endpoint read address value register. Pointer value increments by 2 when accessed by CPU/debugger
USBDEV_ARB_RWx_DR16	Endpoint data register
USBDEV_DMA_THRES16	DMA burst / threshold configuration register
USBLPM_POWER_CTL	Power control register
USBLPM_USBIO_CTL	USB IO control register
USBLPM_FLOW_CTL	Flow control register
USBLPM_LPM_CTL	LPM control register
USBLPM_LPM_STAT	LPM status register
USBLPM_INTR_SIE	USB SOF, BUS RESET, and EP0 interrupt status register
USBLPM_INTR_SIE_SET	USB SOF, BUS RESET, and EP0 interrupt set register
USBLPM_INTR_SIE_MASK	USB SOF, BUS RESET,, and EP0 interrupt mask register
USBLPM_INTR_SIE_MASKED	USB SOF, BUS RESET, and EP0 interrupt masked register
USBLPM_INTR_LVL_SEL	Select interrupt level for each interrupt source register
USBLPM_INTR_CAUSE_HI	High-priority interrupt cause register
USBLPM_INTR_CAUSE_MED	Medium-priority interrupt cause register
USBLPM_INTR_CAUSE_LO	Low-priority interrupt cause register

34. Universal Serial Bus (USB) Host

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The USB subsystem in PSoC 6 MCUs can be configured to function as a USB host. The USB host in the PSoC 6 MCU supports both full-speed (12 Mbps) and low-speed (1.5 Mbps) devices and is designed to be complaint with the USB Specification Revision 2.0. This chapter details the PSoC 6 MCU USB and its operations. For details about the USB specification, refer to the USB Implementers Forum website.

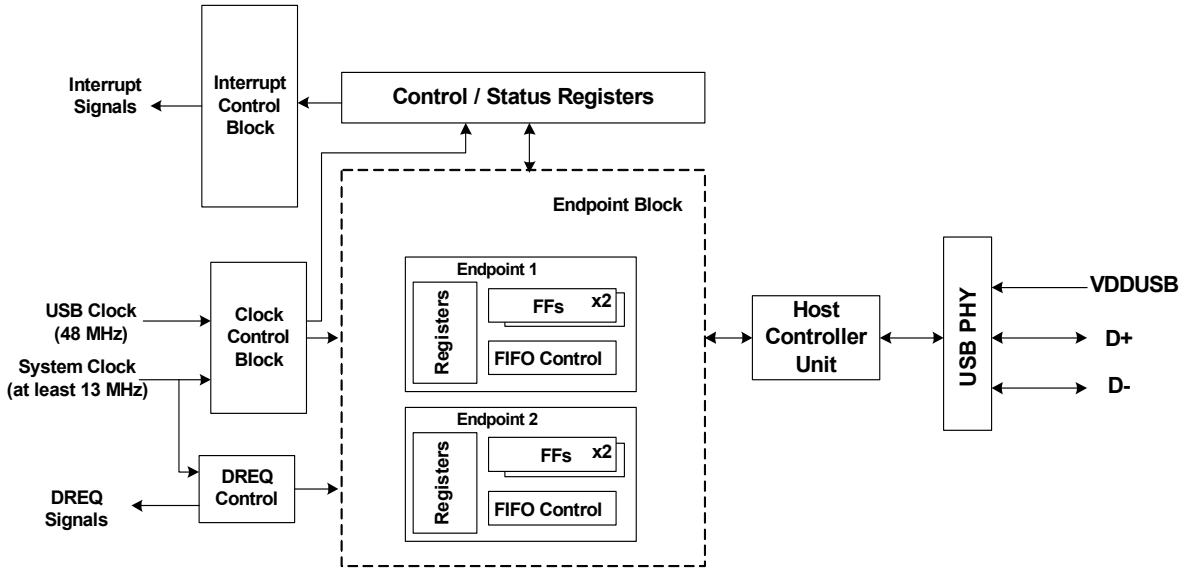
34.1 Features

The USB host in the PSoC 6 MCU has the following features:

- Automatic detection of device connection or disconnection
- Automatic detection of full-speed or low-speed transfer
- Supports USB bus reset function
- Supports IN, OUT, SETUP. and SOF tokens
- Supports Bulk, Control, Interrupt, and Isochronous transfer
- Automatic detection of handshake packet for OUT token and automatic sending of handshake packet for IN token (excluding STALL)
- Supports a maximum packet length of up to 256 bytes
- Supports action against errors (CRC error, toggle error, and timeout)

34.2 Architecture

Figure 34-1. USB Host Block Diagram



34.2.1 USB Physical Layer (USB PHY)

The USB includes the transmitter and receiver (transceiver), which corresponds to the USB PHY. This module allows physical layer communication with the USB device through the D+, D–, and VDDUSB pins. It handles differential mode communication with the device. The USB PHY also includes pull-down resistors on the D+ and D– lines. Differential signaling is used between the USB host and the device. The host controller unit receives the differential signal from the device and converts it to a single-ended signal. While transmitting, the host controller unit converts the single-ended signal to the differential signal, and transmits it to the device. The differential signal is at a nominal voltage range of 0 V to 3.3 V.

34.2.2 Clock Control Block

This block controls the gated clocks – system clock and USB clock. The USB host requires a clock frequency of 48 MHz. USB host compliance requires a clock source with an accuracy of ±0.25 percent. One way of doing this is to use Clk_HF[3] sourced from an highly accurate ECO. The PLL can be used to generate the required 48-MHz clock from the ECO. Refer to the Clocking System chapter on page 242 for details on generation of clock required for USB operation.

34.2.3 Interrupt Control Block

This block controls the interrupts associated with USB host operations. The USB host block has three general-purpose interrupt signals: USBHOST_INTR_USBHOST_LO, USBHOST_INTR_USBHOST_MED, and USBHOST_INTR_USBHOST_HI. There are 11 interrupt trigger events associated with USB host operation that can be mapped to any of the three interrupt lines. Each of these three interrupt lines has a status register to identify the interrupt source. These are the USBHOST_INTR_USBHOST_CAUSE_LO, USBHOST_INTR_USBHOST_CAUSE_MED, and USBHOST_INTR_USBHOST_CAUSE_HI, USBHOST_INTR_USBHOST_EP_CAUSE_LO, USBHOST_INTR_USBHOST_EP_CAUSE_MED and USBHOST_INTR_USBHOST_EP_CAUSE_HI registers.

34.2.4 Endpoint n (n=1, 2)

The USB host has two endpoints. The maximum buffer size of endpoint 1 is 256 bytes and that of endpoint 2 is 64 bytes. The endpoint buffers are used to send and receive data. If endpoint 1 is used as the transmitter, endpoint 2 must be used as the receiver and vice-versa. The DIR bit of the USBHOST_HOST_EPn_CTL (n=1, 2) register is used to configure the endpoint as an IN buffer (DIR=0) or OUT buffer (DIR=1).

34.2.5 DMA Request (DREQ) Control

The USB host supports data transfer using DMA. DMA operation is enabled by configuring the USBHOST_HOST_DMA_ENBL register. There are two DMA transfer modes: Automatic data transfer mode and packet transfer mode. The DMAE bit of the USBHOST_HOST_EPn_CTL (n=1, 2) register is used to set the mode of DMA transfer. The Host Endpoint Block register (USBHOST_HOST_EPn_BLK; n=1, 2) sets the total number of bytes for DMA transfer.

34.3 USB Host Operations

To operate the USB as a host, the following settings are required:

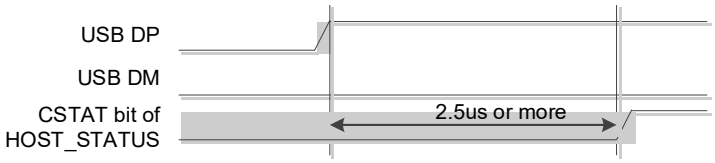
- Enable the pull-down resistors on both D+ and D– pins. To enable the pull-down resistors, set the DP_DOWN_EN bit and DM_DOWN_EN bit of the power control register (USBLPM_POWER_CTL) to ‘1’.
- Set both the HOST and ENABLE bits of the Host Control 0 register (USBHOST_HOST_CTL0) to ‘1’.
- Set the USTP bit of the Host Control 1 register (USBHOST_HOST_CTL1) to ‘0’.
- Supply the operating clock for USB (48 MHz ± 0.25%). In the PSoC 6 MCU, CLK_HF3 is the clock source. The required USB clock can be generated using one of the following clocking schemes:
  - IMO (trimmed with USB) -> PLL -> CLK_HF3
  - ECO (with the required accuracy) -> PLL -> CLK_HF3
  - Use external clock (EXTCLK) with the required accuracy.

34.3.1 Detecting Device Connection

When an external USB device is not connected, both the D+ and D– pins are set to logic LOW by the pull-down resistors. In the unconnected state, CSTAT bit of the USBHOST_HOST_STATUS register is '0' and the TMODE bit is undefined. If an external USB device is connected, then the CSTAT bit is set to '1'.

When a device is detected, the CNNIRQ bit of the Interrupt USB host (USBHOST_INTR_USBHOST) register is set to '1'. A device can be detected either by the method of interrupt or by polling. If '1' is set to the CNNIRQM bit of the Interrupt USB host (USBHOST_INTR_USBHOST_MASK) register, a device connection interrupt occurs. To clear this interrupt, write '1' to the CNNIRQ bit of USBHOST_INTR_USBHOST register. When detecting a device connection by polling, set the CNNIRQM bit of the USBHOST_INTR_USBHOST_MASK register to '0' and check when the CNNIRQ bit of USBHOST_INTR_USBHOST changes to '1'.

Figure 34-2. Device Connection



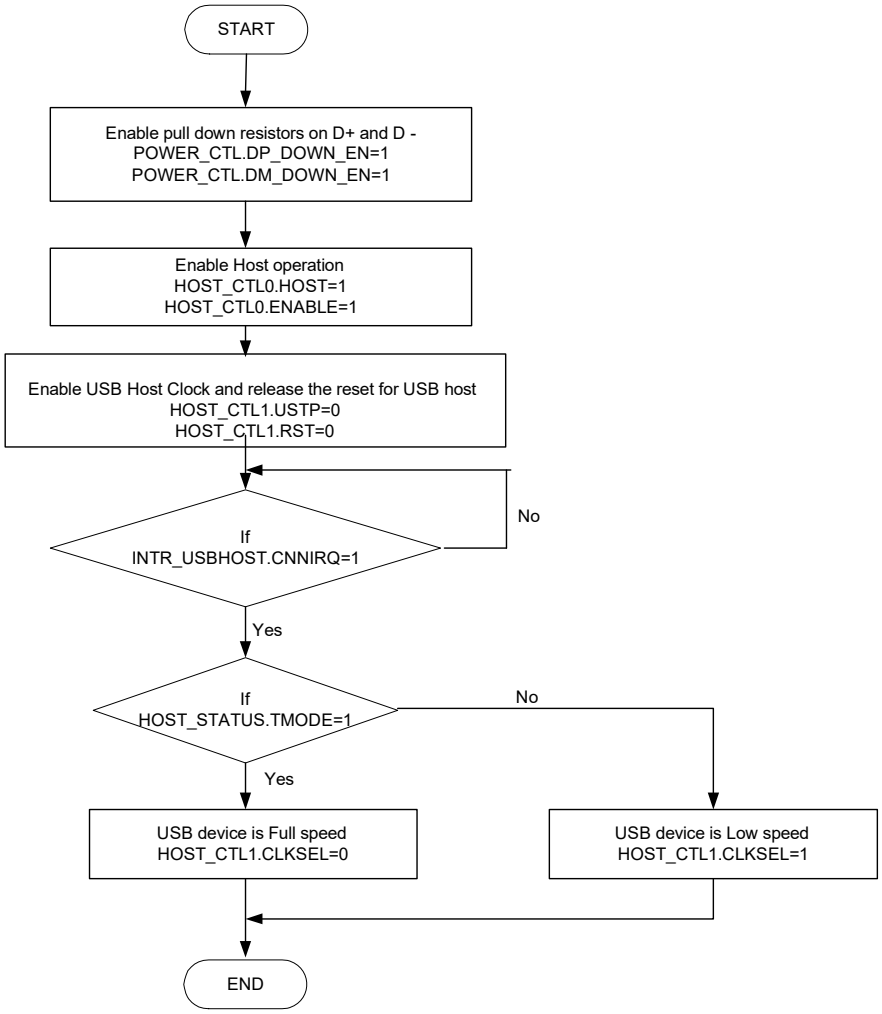
34.3.2 Obtaining Transfer Speed of the USB Device

To obtain the possible transfer speed of the remote USB device after detecting a connection, check the value of the TMODE bit of the USBHOST_HOST_STATUS register. The following shows the relationships between the transfer speed and the value of the TMODE bit of USBHOST_HOST_STATUS.

- If TMODE=1, then the device is a full-speed device
- If TMODE=0, then the device is a low-speed device

Figure 34-3 shows the flow chart for device connection detection and obtaining the transfer speed.

Figure 34-3. Device Connection and Transfer Speed Detection Flow Chart

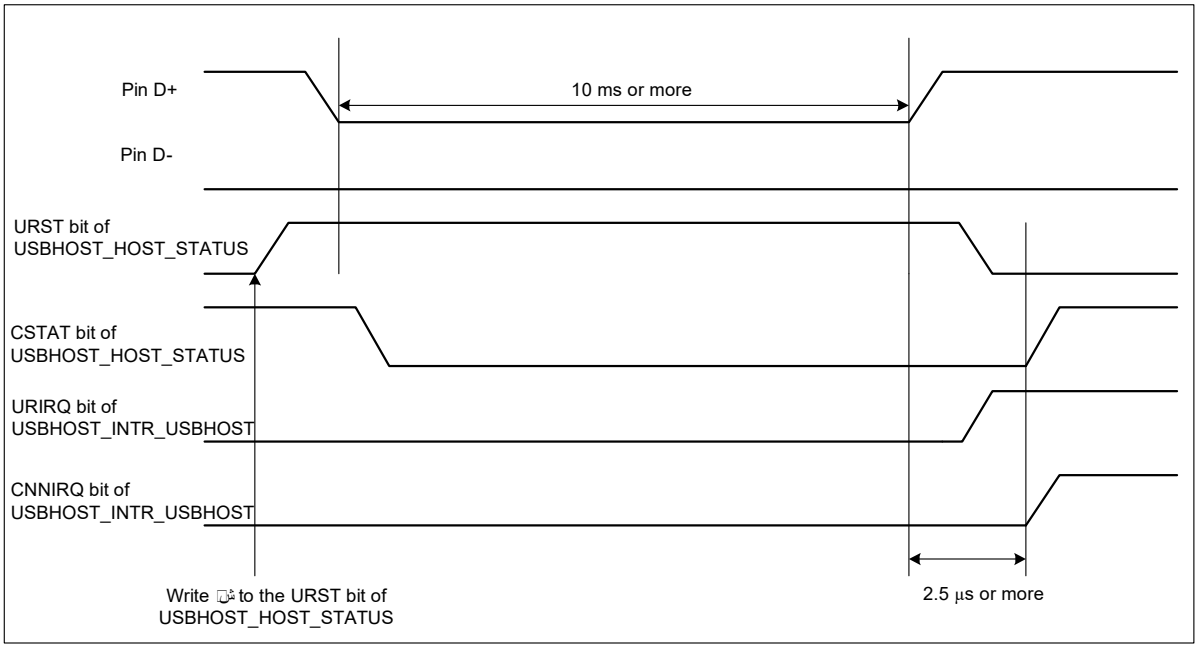


34.3.3 USB Bus Reset

The USB bus is reset by sending SE0 for 10 ms or more if the URST bit of the USBHOST_HOST_STATUS register is set to '1'. After the USB bus is reset, the URST bit of HOST_STATUS is set to '0', and the URIRQ bit of USBHOST_INTR_USBHOST is set to '1'. If the URIRQM bit of the USBHOST_INTR_USBHOST_MASK register is then set to '1', an interrupt occurs. To clear this interrupt, write '1' to the URIRQ bit of USBHOST_INTR_USBHOST.

Figure 34-4 shows the timing diagram for USB bus reset.

Figure 34-4. Device Reset Timing Diagram



34.3.4 USB Packets

Exchange of information between the host and the device takes place in the form of packets, which are always initiated by the host. There are three types of USB packets:

- Token
- Data
- Handshake

The following sections explain about the settings required to initiate the packets by the PSoC 6 MCU USB host.

34.3.4.1 Token Packet

Endpoint 1 and Endpoint 2 buffers are used to send and receive data. If the DIR bit of the USBHOST_HOST_EP1_CTL register is '1', the Endpoint 1 buffer is an OUT buffer. If the DIR bit of the USBHOST_HOST_EP2_CTL register is '0', then the Endpoint 2 buffer is an IN buffer. Both the endpoints should never be configured as an OUT buffer or IN buffer simultaneously, which implies that if Endpoint 1 is an OUT buffer then Endpoint 2 must be an IN buffer and vice-versa.

Use the following settings to process a token packet:

IN Token

1. Set the BFINI bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers to '1'.
2. Set the DIR bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers. Note that if Endpoint 1 is an OUT buffer then Endpoint 2 must be an IN buffer and vice-versa.
3. Set the BFINI bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers to ‘0’.
4. Specify the target address in the USBHOST_HOST_ADDR register.
5. Specify the maximum number of bytes for the packet in the PKSn (n = 1, 2) bit field.
6. Specify the target endpoint, token (IN token), and toggle data in the USBHOST_HOST_TOKEN register.
7. Check if the token transfer is complete; the CMPIRQ bit will be set to ‘1’. To clear this interrupt, write ‘1’ to the CMPIRQ bit of the USBHOST_INTR_USBHOST register. Note that the interrupt is triggered only if the CMPIRQM bit of the USBHOST_INTR_USBHOST_MASK register is ‘1’.
8. Check for token packet transfer errors using the USBHOST_HOST_ERR register. Handle the errors, if any, appropriately.
9. Read the EPnDRQ (n = 1 or 2) bit of the USBHOST_INTR_HOST_EP register. A value of ‘1’ indicates that the packet transfer ended normally. Clear the EPnDRQ bit by writing ‘1’.

10. Read data received from USBHOST_HOST_EPn_RWn_DR register (n =1, 2).

OUT and SETUP Tokens

1. Set the BFINI bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers to ‘1’.
2. Set the DIR bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers. Note that if Endpoint 1 is an OUT buffer then Endpoint 2 must be an IN buffer and vice-versa.
3. Set the BFINI bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers to ‘0’.
4. Specify the target address in the USBHOST_HOST_ADDR register.
5. Specify the maximum number of bytes for the packet in the PKSn (n = 1, 2) bit field.
6. Write the data to be sent to the Endpoint n (n = 1 or 2) buffer. Use USBHOST_HOST_EPn_WR1_DR for 1-byte data and USBHOST_HOST_EPn_WR2_DR for 2-byte data. Also, set the EPnDRQ (n = 1 or 2) bit of the USBHOST_INTR_HOST_EP register as ‘1’.
7. Specify the target endpoint, token (OUT token), and toggle data in the USBHOST_HOST_TOKEN register.
8. Check if the token transfer is complete; the CMPIRQ bit will be set to ‘1’. To clear this interrupt, write ‘1’ to the CMPIRQ bit of the USBHOST_INTR_USBHOST register. Note that the interrupt is triggered only if the CMPIRQM bit of the USBHOST_INTR_USBHOST_MASK register is ‘1’.
9. Check for token packet transfer errors using the USBHOST_HOST_ERR register. Handle the errors, if any, appropriately.

The USB PHY sends a token packet in the order of sync, token, address, endpoint, CRC5, and EOP based on the specified token; however, sync, CRC5, and EOP are sent automatically. After one packet is sent, the CMPIRQ bit of the USBHOST_INTR_USBHOST register is set to ‘1’. The TKNEN bit of the USBHOST_HOST_TOKEN register is set to ‘000’. At this time, if the CMPIRQM bit of USBHOST_INTR_USBHOST_MASK register is '1', an interrupt occurs. To clear this interrupt, write '1' to the CMPIRQ bit of the USBHOST_INTR_USBHOST register.

Figure 34-5. IN Token Flow Chart

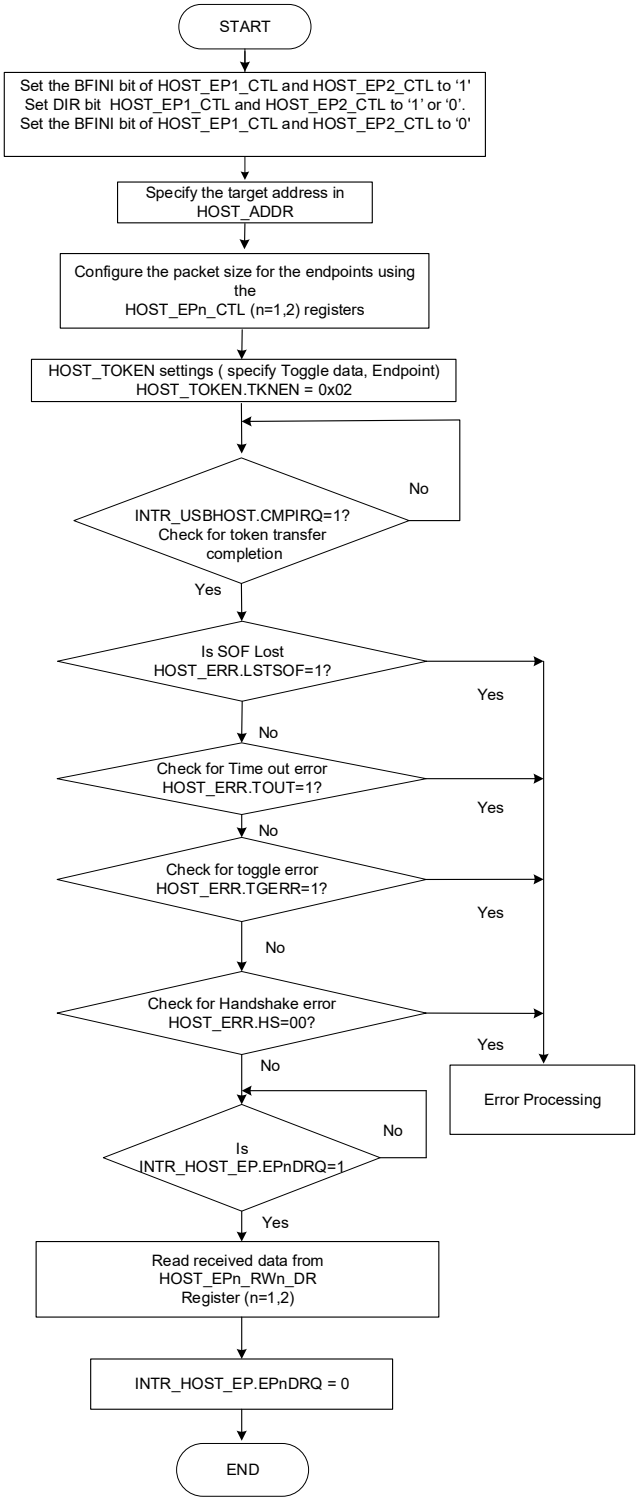
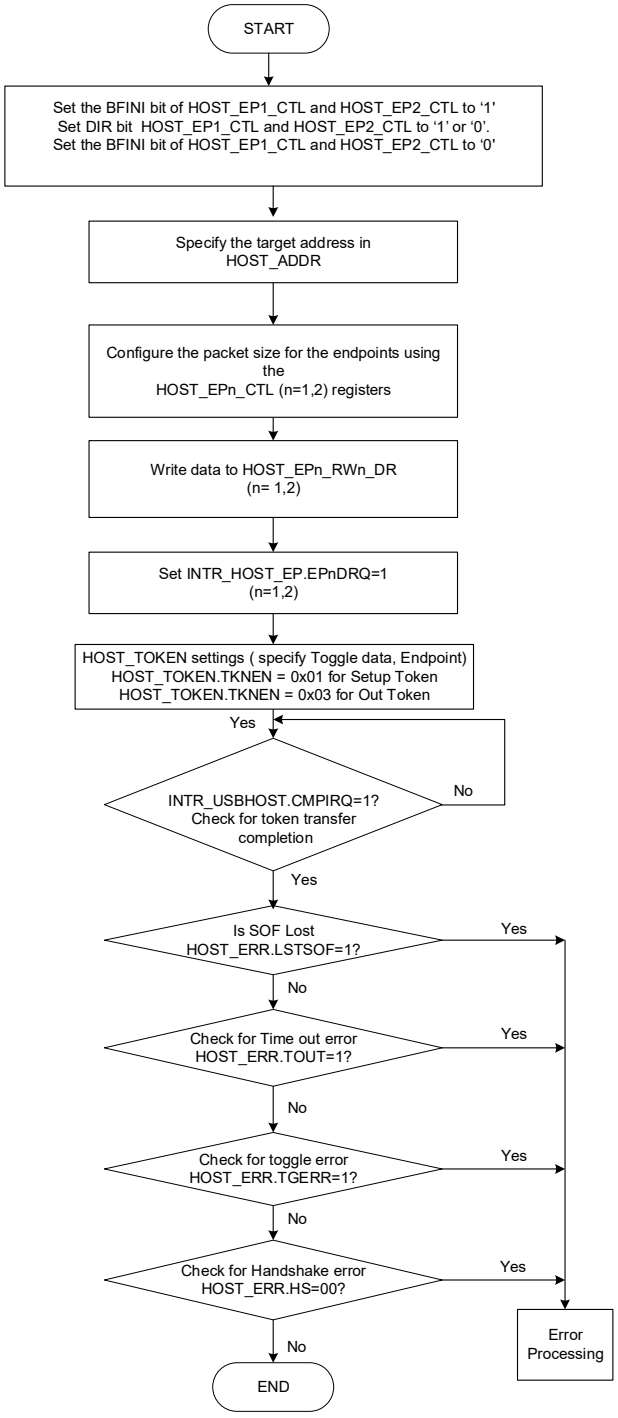


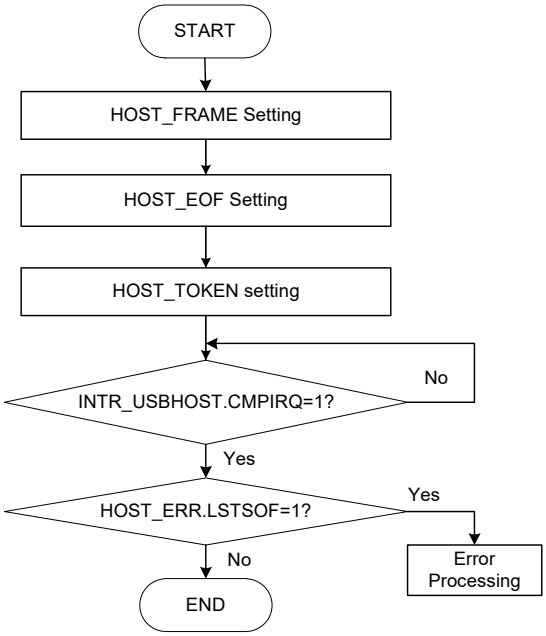
Figure 34-6. SETUP/OUT Token Flow Chart



When issuing an SOF token, specify the EOF time in the Host EOF Setup (USBHOST_HOST_EOF) register and the frame number in the Host Frame Setup (USBHOST_HOST_FRAME) register respectively. Then, specify an SOF token code in the TKNEN bit of the USBHOST_HOST_TOKEN register. After sync, SOF token, frame number, CRC5, and EOP are sent, the SOFBUSY bit of the USBHOST_HOST_STATUS register is set to '1', and the USBHOST_HOST_FRAME register is incremented by one. The CMPIRQ bit of the USBHOST_INTR_USBHOST register is also set to '1', causing the TKNEN bit of the USBHOST_HOST_TOKEN register to be cleared to '000'. To clear a token completion interrupt, write '1' to the CMPIRQ bit of the Host Interrupt (USBHOST_HIRQ) register.

An SOF is automatically sent every 1 ms while the SOFBUSY bit of the USBHOST_HOST_STATUS register is '1'. Figure 34-7 depicts steps to send an SOF token

Figure 34-7. SOF Token Flow Chart

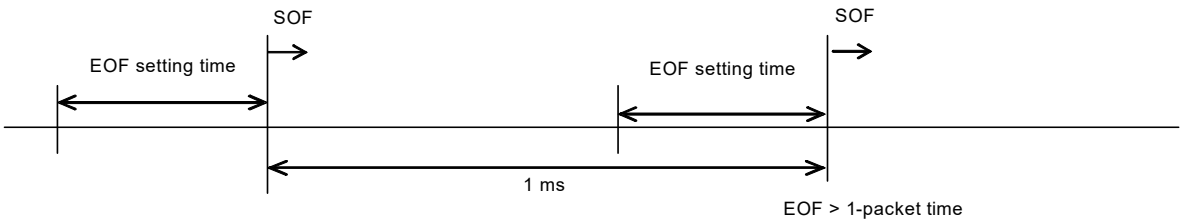


The conditions (SOF stop conditions) to set the SOFBUSY bit of the USBHOST_HOST_STATUS register to '0' are as follows:

- Writing 0 to the SOFBUSY bit of USBHOST_HOST_STATUS
- Resetting the USB bus
- Writing 1 to the SUSP bit of USBHOST_HOST_STATUS
- Disconnecting the USB device (when the CSTAT bit of USBHOST_HOST_STATUS is '0')

The USBHOST_HOST_EOF register is used to prevent the SOF from being sent simultaneously with other tokens. If the TKNEN bit of the USBHOST_HOST_TOKEN register is written within the time period specified by the USBHOST_HOST_EOF register, the specified token is placed into the wait state. After the SOF is sent, the token in the wait state is issued. The HOST_EOF Setup register specifies a 1-bit time as the time unit. When the EOF setting is shorter than the 1-packet time, SOF may be sent doubly during execution of other tokens. In this case, the LSTSOF bit of the Host Error Status (USBHOST_HOST_ERR) register is set to '1' and SOF is not sent. If '1' is set to the LSTSOF bit of the USBHOST_HOST_ERR register, the value of the Host EOF Setup register must be increased.

Figure 34-8. SOF Timing



34.3.4.2 Data Packet

Follow these steps to send or receive a data packet after sending a token packet.

- Transmitting data (host to device)
  - Sync pattern is automatically sent.
  - If the TGGL bit of USBHOST_HOST_TOKEN is 0, DATA0 is sent. If the TGGL bit is 1, DATA1 is sent.
  - If the DIR bit of the USBHOST_HOST_EP1_CTL register is 1 (that is, if EP1 is an OUT endpoint), select the Endpoint 1 buffer; otherwise, select the Endpoint 2 buffer. Then, send all the target data.
  - 16-bit CRC is automatically sent.
  - 2-bit EOP is automatically sent.
  - 1-bit J State is automatically sent.
- Receiving data (device to host)
  - Receive sync.
  - Receive toggle data, and compare it with the value of the TGGL bit of HOST_TOKEN.
  - If the toggle data matches the value of the TGGL bit, check the DIR bit of HOST_EP1_CTL. If the DIR bit is 1, select the Endpoint 2 buffer; otherwise, select the Endpoint 1 buffer. Then, distribute the received data to the respective buffers.
  - Verify the 16-bit CRC when EOF is received.

34.3.4.3 Handshake Packet

A handshake packet is used to notify the remote device of the status of the local device. A handshake packet sends either one of ACK, NAK, and STALL from the receiving side when it is judged that the receiving side is ready to receive data normally. If the USB circuit receives a handshake packet, the type of the received handshake packet is set to the HS bit of the USBHOST_HOST_ERR register.

34.3.5 Retry Function

When a NAK or CRC error occurs at the end of a packet, if '1' is set to the RETRY bit of the Host Control 2 (USBHOST_HOST_CTL2) register, processing is retried repeatedly for the period specified in the Host Retry Timer Setup (USBHOST_HOST_RTIMER) register.

When an error other than STALL or device disconnection occurs, the target token is retried if the RETRY bit of HOST_CTL2 is 1. The conditions to end retry processing are as follows.

- The RETRY bit of HOST_CTL2 is set to 0.
- '0' is detected in the retry timer.
- The interrupt flag is generated by SOF (SOFIRQ of USBHOST_INTR_USBHOST = 1).
- ACK is detected.
- A device disconnection is detected.

The retry timer is activated when a token is sent while the RETRY bit of HOST_CTL2 is '1'. The retry time is then decremented by one when a 1-bit transfer clock (12 MHz in full-speed mode) is output. When the retry timer reaches 0, the target token is sent, and processing ends. If a token retry occurs in the EOF area, the retry timer is stopped until SOF is sent. After SOF is sent, the retry timer restarts with the value that is set when the timer stopped.

34.3.6 Error Status

The USB host supports detection of the following types of errors:

- Stuffing Error

  If 1 is writter to six successive bits, 0 is inserted into one bit. If 1 is successively detected in seven bits, it is regarded as a Stuffing error, and the STUFF bit of the USBHOST_HOST_ERR register is set to 1. To clear this status, write '1' to the STUFF bit.
- Toggle Error

  When sending an IN token, the Toggle Data field of a data packet is compared with the value of the TGGL bit of the USBHOST_HOST_TOKEN register. If they do not match, the TGERR bit of the USBHOST_HOST_ERR register is set to 1. To clear the TGERR bit, write '1' to the TGERR bit of the USBHOST_HOST_ERR register.
- CRC Error

  When receiving an IN token, the data and CRC of the received data packet are obtained with the CRC polynomial G(X) = X16 + X15 + X2 + 1. If the remainder is not 0x800D, it means that a CRC error has occurred, and the CRC bit of the USBHOST_HOST_ERR register is set to 1. To clear the CRC bit, write '1' to the CRC bit of USBHOST_HOST_ERR. For more details, refer to the documentation: CRC in USB.
- Time Out Error

  The TOUT bit of the USBHOST_HOST_ERR register is set and time out error is considered under any of the following conditions:
  - A data packet or handshake packet is not input in the specified time
  - SE0 is detected when receiving data
  - Stuffing error is detected

  To clear the TOUT bit, write 0 to the TOUT bit of the USBHOST_HOST_ERR register.
- Receive Error

  The PKSn (n = 1, 2) bit of Endpoint Control register (USBHOST_HOST_EPn_CTL; n=1, 2) indicates the receiver packet size. When the received data exceeds the specified receive packet size, the RERR bit of the USBHOST_HOST_ERR register is set to 1. To clear the RERR bit, write '1' to the RERR bit of USBHOST_HOST_ERR.
- Lost SOF Error

  When the LSTSOF bit of the USBHOST_HOST_ERR register is set, it means that the SOF token cannot be

sent because another token is in process. When this bit is ‘0’, it means that no lost SOF error is detected.

34.3.7 End of Packet (EOP)

If a packet ends in the USB host, the CMPIRQ bit of the USBHOST_INTR_USBHOST register is set to 1. At this time, if the CMPIRQM bit of the USBHOST_INTR_USBHOST_MASK register is 1, an interrupt occurs.

When a packet ends, the interrupt is generated when the TKNEN bit of the USBHOST_HOST_TOKEN register is set as ‘001’ (SETUP token), ‘010’ (IN token), ‘011’ (OUT token), or ‘100’ (SOF token).

Figure 34-9. EOP Interrupt Timing Diagram (for IN/OUT/SETUP token)

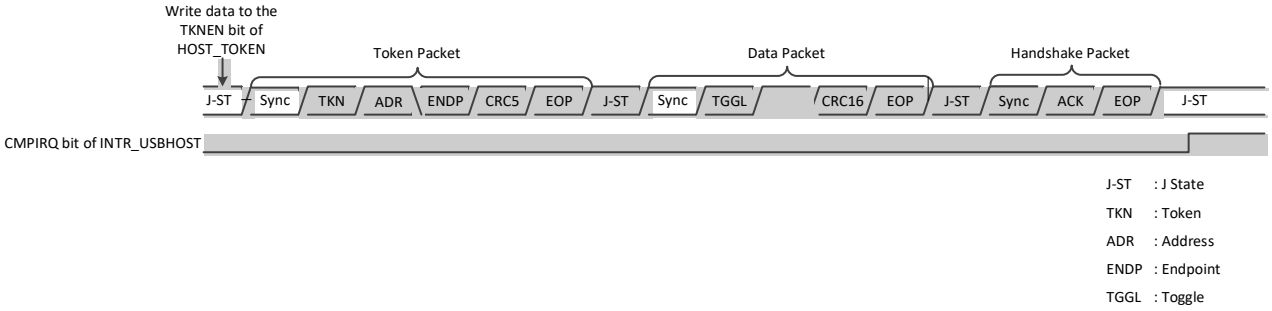
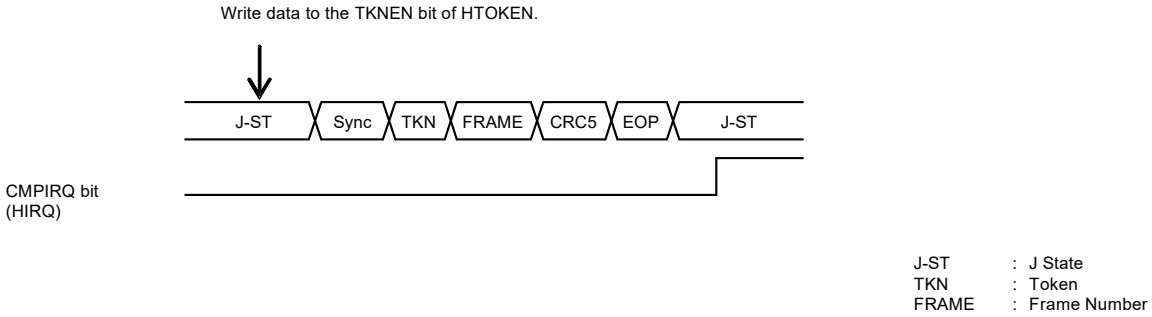


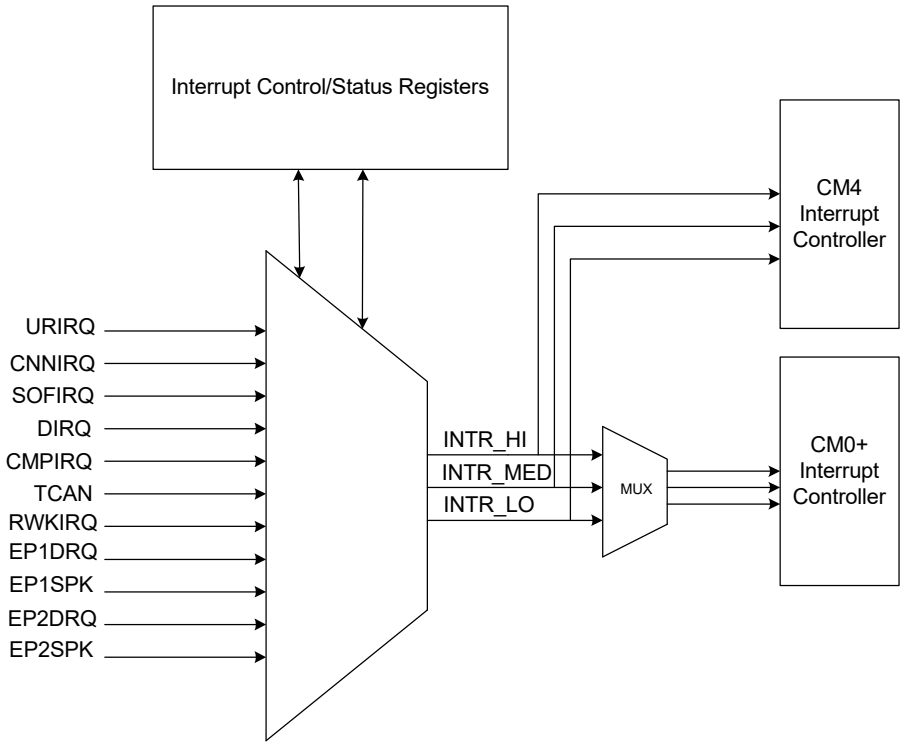
Figure 34-10. EOP Interrupt Timing Diagram (for SOF token)



34.3.8 Interrupt Sources

The USB host generates 11 interrupts to the CPU. These interrupts are mapped to three general-purpose interrupt lines: USBHOST_INTR_LO, USBHOST_INTR_MED, and USBHOST_INTR_HI. Each of these three interrupt lines has an associated status register, which identifies the cause of the interrupt event. These are USBHOST_INTR_USBHOST_CAUSE_LO, USBHOST_INTR_USBHOST_CAUSE_MED, USBHOST_INTR_USBHOST_CAUSE_HI, USBHOST_INTR_HOST_EP_CAUSE_LO, USBHOST_INTR_HOST_EP_CAUSE_MED, and USBHOST_INTR_HOST_EP_CAUSE_HI.

Figure 34-11. USB Host Interrupt Sources



The following events generate an interrupt on one of the three interrupt lines:

- Start-of-frame (SOF) event

  SOF interrupt is generated when an SOF token is sent. The SOF interrupt is enabled by setting the SOFIRQM bit of the USBHOST_INTR_USBHOST_MASK register. The status of the SOF interrupt is reflected in the SOFIRQ bit of the USBHOST_INTR_USBHOST register.

- Device connection and disconnection events

  The CNNIRQ bit of the USBHOST_INTR_USBHOST register is set to ‘1’ when a device connection is detected. The interrupt can be enabled by setting the CNNIRQM bit of the USBHOST_INTR_USBHOST_MASK register.

  When a device is disconnected, an interrupt is generated if the DIRQM bitfield of USBHOST_INTR_USBHOST_MASK is set to ‘1’. The status of the device disconnection is reflected in the DIRQ bit of the USBHOST_INTR_USBHOST register.

- USB bus reset event

  The URIRQ bit of the USBHOST_INTR_USBHOST register is set when the USB bus reset ends. An interrupt is generated if the URIRQM bit of the USBHOST_INTR_USBHOST_MASK register is set to ‘1’.

- Token completion event

  When token (IN, OUT, or SETUP token) processing is complete, the CMPIRQ bit of the USBHOST_INTR_USBHOST register is set. An interrupt is generated if the CMPIRQM bit is set to ‘1’.

- Interrupt on resume event

  The RWKIRQ bit of the USBHOST_INTR_USBHOST register is set when the host enters the resume state from suspend state. The RWKIRQ bit is set under the following conditions:

  - The SUSP bit of the USBHOST_HOST_STATUS register is set to ‘0’.
  - The D+/D– pins are placed in the K-state.
  - A device connection/disconnection is detected.

  An interrupt is generated on a resume event if the RWKIRQM bit of the USBHOST_INTR_USBHOST_MASK register is set to ‘1’.

- Interrupt on token cancellation

An interrupt is generated when a token send is canceled based on the setting of the CANCEL bit in the Host Control 2 register (USBHOST_HOST_CTL2). If the CANCEL bit is set and if the target token is written to the USBHOST_HOST_TOKEN register in the EOF area (specified in the Host EOF Setup register), the token send is canceled.

- Endpoint interrupts

  The USB host has two endpoints: Endpoint1 and Endpoint2. Each endpoint can generate two interrupts. When the packet transfer of an endpoint ends normally, the EPnDRQ (n=1or 2) bit of the USBHOST_INTR_HOST_EP register is set to '1'. An interrupt can be triggered if the EPnDRQM (n=1or 2) bit of the USBHOST_INTR_HOST_EP_MASK register is set to '1'. If the packet size does not match the packet size specified in the PKS bit of the HOST_EPn_CTL (n=1 or 2) register, the EPnSPK (n=1or 2) bit of the USBHOST_INTR_HOST_EP register is set. An interrupt can be configured by setting the EPnSPKM (n=1or 2) bit of the USBHOST_INTR_HOST_EP_MASK register.

34.3.9 DMA Transfer Function

Data handled by the USB host can be transferred via DMA between the send/receive buffer and the internal RAM. There are two modes of DMA transfer:

- Packet transfer mode
- Automatic buffer transfer mode

34.3.9.1 Packet Transfer Mode

The packet transfer mode transfers each packet according to the configured data size in DMA. This transfer mode can access each buffer of the endpoints.

In the packet transfer mode the OUT direction transfer (host to device) involves the following sequence of steps:

1. After the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register is set and the interrupt handling is entered, configure the DMA register settings relevant to the number of transfers and block size corresponding to the data size to be transferred in the next OUT packet; then, enable DMA to start the transfer.
2. After the DMA transfer, clear the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register and the interrupt cause flag of DMAC, and return from the interrupt handling.

Figure 34-12. Packet Transfer Mode: OUT Direction

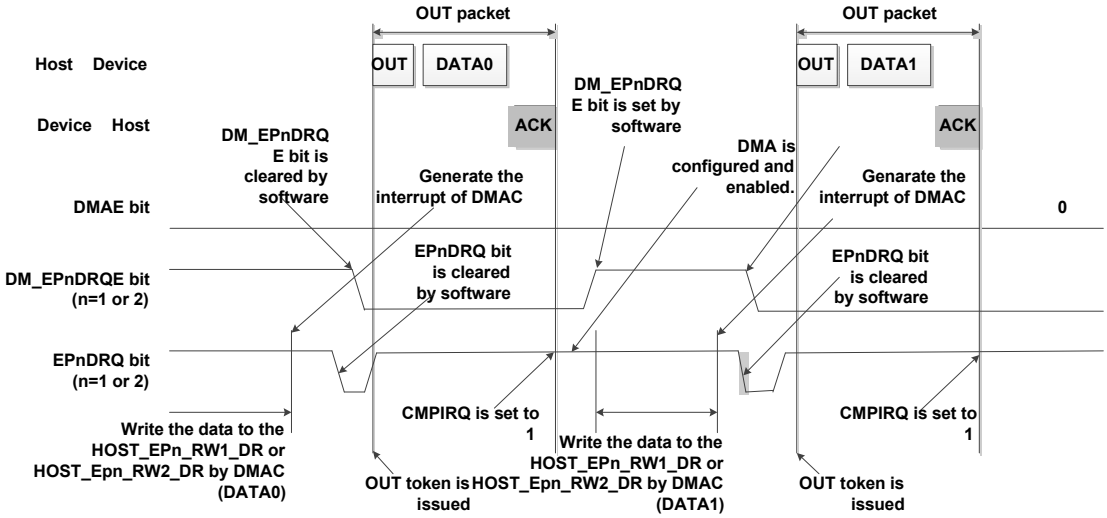
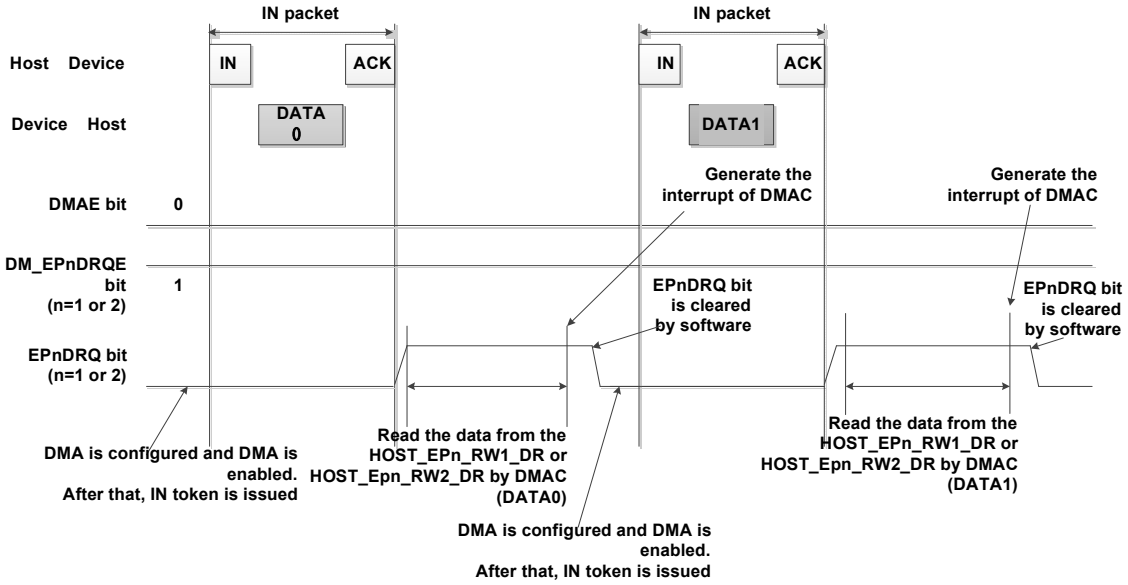


Figure 34-13. Packet Transfer Mode: IN Direction



The data transfer in the IN direction (device to host) involves the following sequence of steps:

1. After the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register is set and the interrupt handling is entered, check the transfer data size.
2. Configure the DMA register setting relevant to the number of transfers and block size corresponding to the transfer data size, and then enable DMA to start the transfer.
3. After the transfer, clear the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register and the interrupt source flag of DMAC, and return from the interrupt handling.

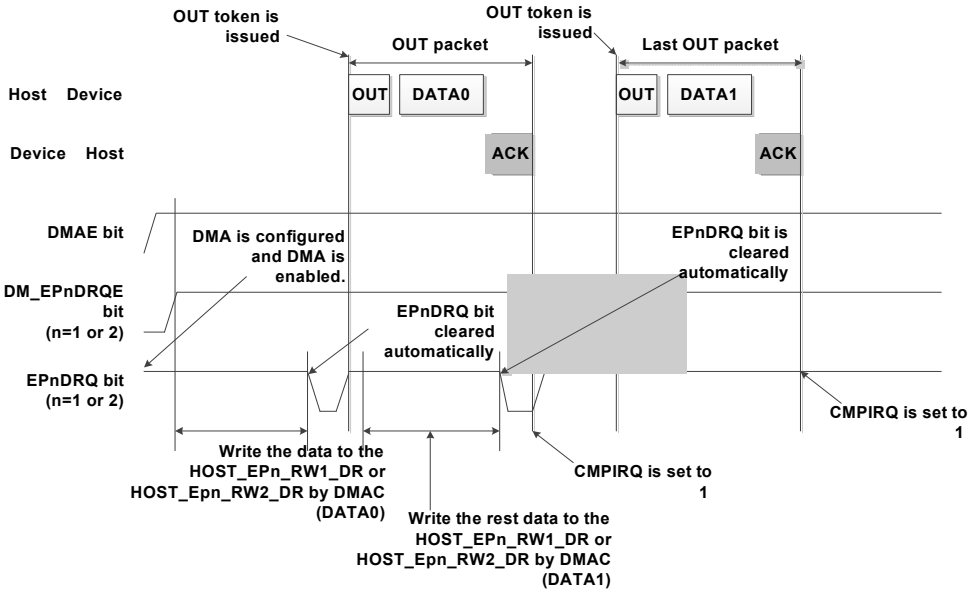
34.3.9.2 Automatic Data Transfer Mode

Automatic data transfer mode is enabled by setting the DMAE bit of the USBHOST_HOST_EPn_CTL register (n=1 or 2). When the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register is set while DMAE is enabled, the DMA request is automatically cleared when the data size corresponding to PKSn (n = 1, 2) bits of the USBHOST_HOST_EPn_CTL register (n=1 or 2) has been transferred. Later, the same sequence is repeated until the data size configured in the DMA is reached. In this mode, configuration by software is not required. Thus, this mode can transfer data automatically by a single setting. It can also transfer even bytes; to transfer odd bytes, a software transfer sequence is required.

The data transfer in the OUT direction (host to device) must be processed in the following sequence:

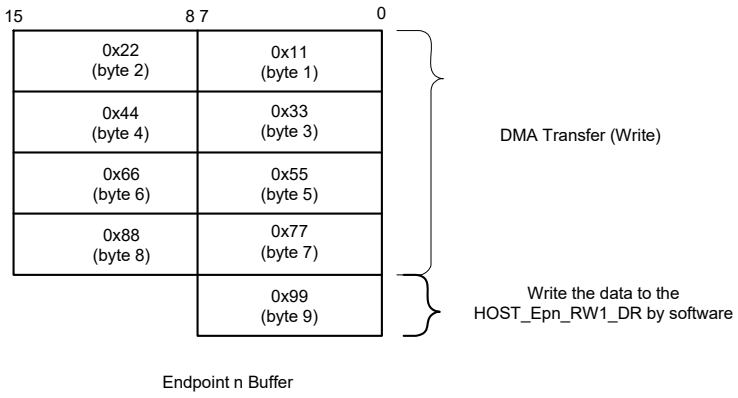
1. Configure the DMA register setting relevant to the number of transfers and block size corresponding to the total data size, and then enable DMA to start the transfer.
2. Set the packet size and total number of bytes for DMA transfer to PKS bits of the USBHOST_HOST_EPn_CTL (n=1 or 2) register and BLK_NUM bits of the USBHOST_HOST_EPn_BLK (n=1 or 2) register, respectively.
3. Enable DMAE bit of the USBHOST_HOST_EPn_CTL (n=1 or 2) register and EPnDRQE bit of the Host DMA Enable (USBHOST_HOST_DMA_ENBL) (n=1 or 2) register.
4. After the transfer, reconfigure the DMAC using an interrupt generated by the DMA controller, and clear the flag to return from interrupt handling.

Figure 34-14. Automatic Data Transfer Mode: OUT Direction



To transfer the odd bytes of data via DMA, write the last byte of data by software transfer.

Figure 34-15. Odd Bytes Transfer in the OUT Direction



The data transfer in the IN direction (device to host) must be processed in the following sequence:

1. Configure the DMA register setting relevant to the number of transfers and block size corresponding to the total data size, and then enable DMA to start the transfer.
2. Enable DMAE bit of the USBHOST_HOST_EPn_CTL (n = 1 or 2) register and EPnDRQE bit of the USBHOST_HOST_DMA_ENBL (n = 1 or 2) register.
3. After the transfer, reconfigure the DMA controller using an interrupt generated by the DMAC and clear the flag to return from interrupt handling.

Figure 34-16. Automatic Data Transfer Mode: IN Direction

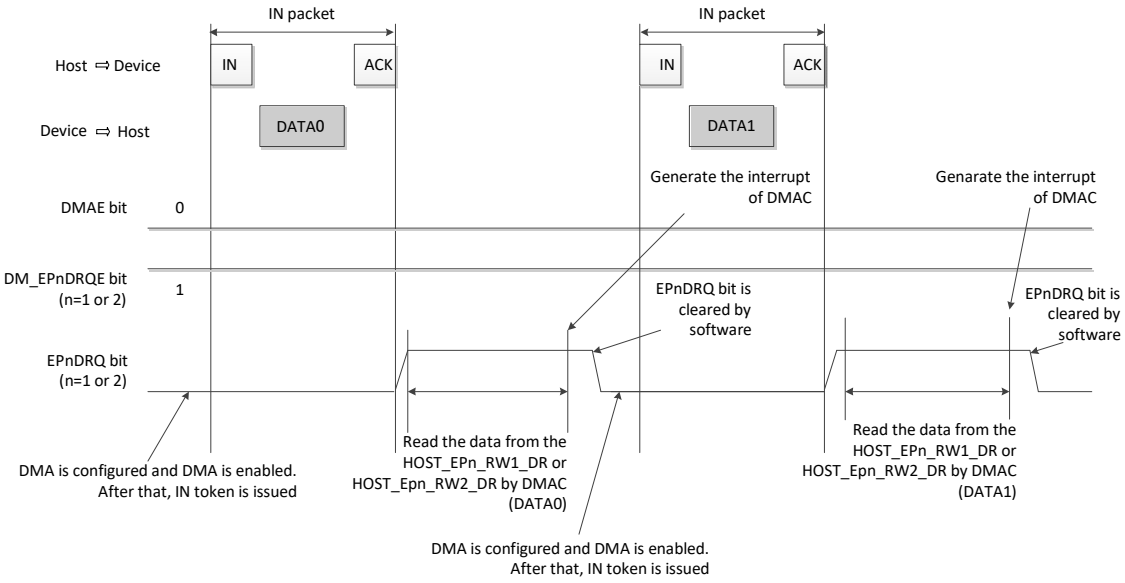
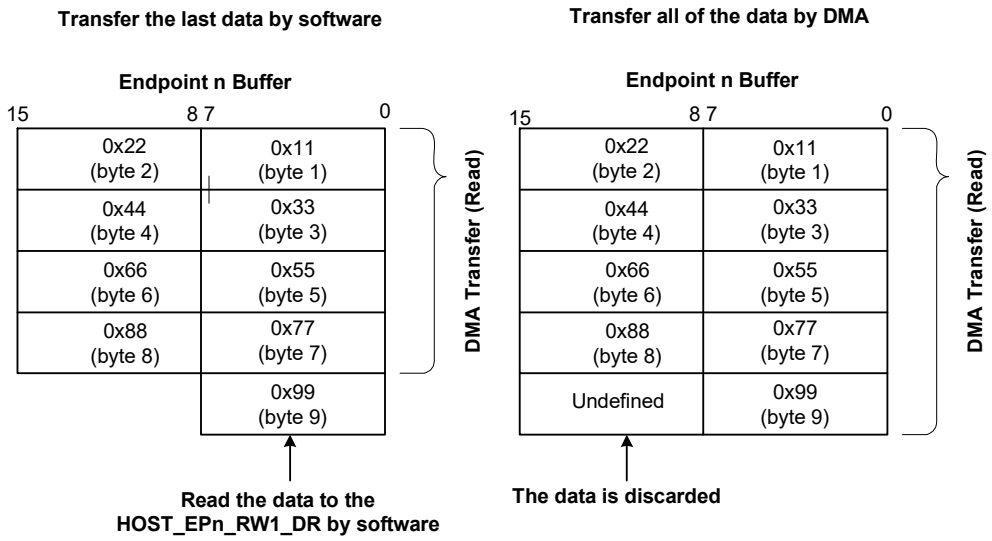


Figure 34-17. Odd Bytes Transfer in the IN Direction



To transfer the data size corresponding to odd number of bytes via DMA, either of the following methods can be used:

- Use the software transfer only for the last data, and read the low-order byte (HOST_EPn_RW1_DR: n=1 or 2).
- Transfer all the data + 1 byte via DMA, and discard the last data after an endian conversion.

34.3.10 Suspend and Resume Operations

The USB host supports suspend and resume operations.

34.3.10.1 Suspend Operation

If the SUSP bit of the USBHOST_HOST_STATUS register is set, the following procedure is performed, and the USB circuit is placed in the suspend state.

- The USB bus is placed in the high-impedance state.
- All circuit blocks with no clock required are stopped.

34.3.10.2 Resume Operation

The USB bus changes from the suspend state to the resume state to start processing when one of the following conditions is satisfied.

- SUSP bit of the USBHOST_HOST_STATUS register is set to ‘0’.
- The D+ or D– pin is placed in the K-state mode by the device.
- A device disconnection is detected.
- A device connection is detected.

The host can start issuing tokens when RWKIRQ bit of the USBHOST_INTR_USBHOST register is set to ‘1’.

34.3.11 Device Disconnection

The device disconnection timer starts when both the D+ and D– pins are set to LOW. If both D+ and D– remain at LOW for 2.5 µs or more, the device is considered to be disconnected. This then sets the CSTAT bit of the USBHOST_HOST_STATUS register as ‘0’ and the DIRQ bit of the USBHOST_INTR_USBHOST register as ‘1’. At this time, if the DIRQM bit of the USBHOST_INTR_USBHOST_MASK register is 1, an interrupt occurs. To clear this interrupt, write '1' to the DIRQ bit of the USBHOST_INTR_USBHOST register.

If the USB bus is reset, the device is considered to be disconnected. In this case, the CSTAT bit of the USBHOST_HOST_STATUS register is set to ‘0’, but the DIRQ bit of the USBHOST_INTR_USBHOST register is not set to 1.

34.4 USB Host Registers

Name	Description
USBHOST_HOST_CTL0	Host control 0 register
USBHOST_HOST_CTL1	Host control 1 register
USBHOST_HOST_CTL2	Host control 2 register
USBHOST_HOST_ERR	Host error status register
USBHOST_HOST_STATUS	Host status register
USBHOST_HOST_FCOMP	Host SOF interrupt frame compare register
USBHOST_HOST_RTIMER	Host retry timer setup register
USBHOST_HOST_ADDR	Host address register
USBHOST_HOST_EOF	Host EOF setup register
USBHOST_HOST_FRAME	Host frame setup register
USBHOST_HOST_TOKEN	Host token endpoint register
USBHOST_HOST_EP1_CTL	Host endpoint 1 control register
USBHOST_HOST_EP1_STATUS	Host endpoint 1 status register
USBHOST_HOST_EP1_RW1_DR	Host endpoint 1 data 1-byte register
USBHOST_HOST_EP1_RW2_DR	Host endpoint 1 data 2-byte register
USBHOST_HOST_EP2_CTL	Host endpoint 2 control register
USBHOST_HOST_EP2_STATUS	Host endpoint 2 status register
USBHOST_HOST_EP2_RW1_DR	Host endpoint 2 data 1-byte register
USBHOST_HOST_EP2_RW2_DR	Host endpoint 2 data 2-byte register
USBHOST_HOST_LVL1_SEL	Host interrupt level 1 selection register
USBHOST_HOST_LVL2_SEL	Host interrupt level 2 selection register
USBHOST_INTR_USBHOST_CAUSE_HI	Interrupt USB host cause high register
USBHOST_INTR_USBHOST_CAUSE_MED	Interrupt USB host cause medium register
USBHOST_INTR_USBHOST_CAUSE_LO	Interrupt USB host cause low register
USBHOST_INTR_HOST_EP_CAUSE_HI	Interrupt USB host endpoint cause high register
USBHOST_INTR_HOST_EP_CAUSE_MED	Interrupt USB host endpoint cause medium register
USBHOST_INTR_HOST_EP_CAUSE_LO	Interrupt USB host endpoint cause low register
USBHOST_INTR_USBHOST	Interrupt USB host register
USBHOST_INTR_USBHOST_SET	Interrupt USB host set register
USBHOST_INTR_USBHOST_MASK	Interrupt USB host mask register
USBHOST_INTR_USBHOST_MASKED	Interrupt USB host masked register
USBHOST_INTR_HOST_EP	Interrupt USB host endpoint register
USBHOST_INTR_HOST_EP_SET	Interrupt USB host endpoint set register
USBHOST_INTR_HOST_EP_MASK	Interrupt USB host endpoint mask register
USBHOST_INTR_HOST_EP_MASKED	Interrupt USB host endpoint masked register
USBHOST_HOST_DMA_ENBL	Host DMA enable register
USBHOST_HOST_EP1_BLK	Host endpoint 1 block register
USBHOST_HOST_EP2_BLK	Host endpoint 2 block register

35. LCD Direct Drive

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU Liquid Crystal Display (LCD) drive system is a highly configurable peripheral that allows the device to directly drive STN and TN segment LCDs.

35.1 Features

The PSoC 6 MCU LCD segment drive block has the following features:

- Supports up to 61 segments and 8 commons
- Supports Type A (standard) and Type B (low-power) drive waveforms
- Any GPIO can be configured as a common or segment
- Supports five drive methods:
  - Digital correlation
  - PWM at 1/2 bias
  - PWM at 1/3 bias
  - PWM at 1/4 bias
- Ability to drive 3-V displays from 1.8 V V_{DD} in Digital Correlation mode
- Operates in Active, Sleep, and Deep Sleep modes
- Digital contrast control

35.2 Architecture

35.2.1 LCD Segment Drive Overview

A segmented LCD panel has the liquid crystal material between two sets of electrodes and various polarization and reflector layers. The two electrodes of an individual segment are called commons (COM) or backplanes and segment electrodes (SEG). From an electrical perspective, an LCD segment can be considered as a capacitive load; the COM/SEG electrodes can be considered as the rows and columns in a matrix of segments. The opacity of an LCD segment is controlled by varying the root-mean-square (RMS) voltage across the corresponding COM/SEG pair.

The following terms/voltages are used in this chapter to describe LCD drive:

- **V_{RMSOFF}**: The voltage that the LCD driver can realize on segments that are intended to be off.
- **V_{RMSON}**: The voltage that the LCD driver can realize on segments that are intended to be on.

- **Discrimination Ratio (D)**: The ratio of V_{RMSON} and V_{RMSOFF} that the LCD driver can realize. This depends on the type of waveforms applied to the LCD panel. Higher discrimination ratio results in higher contrast.

Liquid crystal material does not tolerate long term exposure to DC voltage. Therefore, any waveforms applied to the panel must produce a 0-V DC component on every segment (on or off). Typically, LCD drivers apply waveforms to the COM and SEG electrodes that are generated by switching between multiple voltages. The following terms are used to define these waveforms:

- **Duty**: A driver is said to operate in 1/M duty when it drives 'M' number of COM electrodes. Each COM electrode is effectively driven 1/M of the time.
- **Bias**: A driver is said to use 1/B bias when its waveforms use voltage steps of (1/B) × VDRV. VDRV is the highest drive voltage in the system (equals V_{DD}). The PSoC 6 MCU supports 1/2, 1/3, and 1/4 biases in PWM drive modes.
- **Frame**: A frame is the length of time required to drive all the segments. During a frame, the driver cycles through the commons in sequence. All segments receive 0-V DC (but non-zero RMS voltage) when measured over the entire frame.

The PSoC 6 MCU supports two different types of drive waveforms in all drive modes. These are:

- **Type-A Waveform**: In this type of waveform, the driver structures a frame into M sub-frames. 'M' is the number of COM electrodes. Each COM is addressed only once during a frame. For example, COM[i] is addressed in sub-frame i.
- **Type-B Waveform**: The driver structures a frame into 2M sub-frames. The two sub-frames are inverses of each other. Each COM is addressed twice during a frame. For example, COM[i] is addressed in sub-frames i and M+i. Type-B waveforms are slightly more power efficient because it contains fewer transitions per frame.

35.2.2 Drive Modes

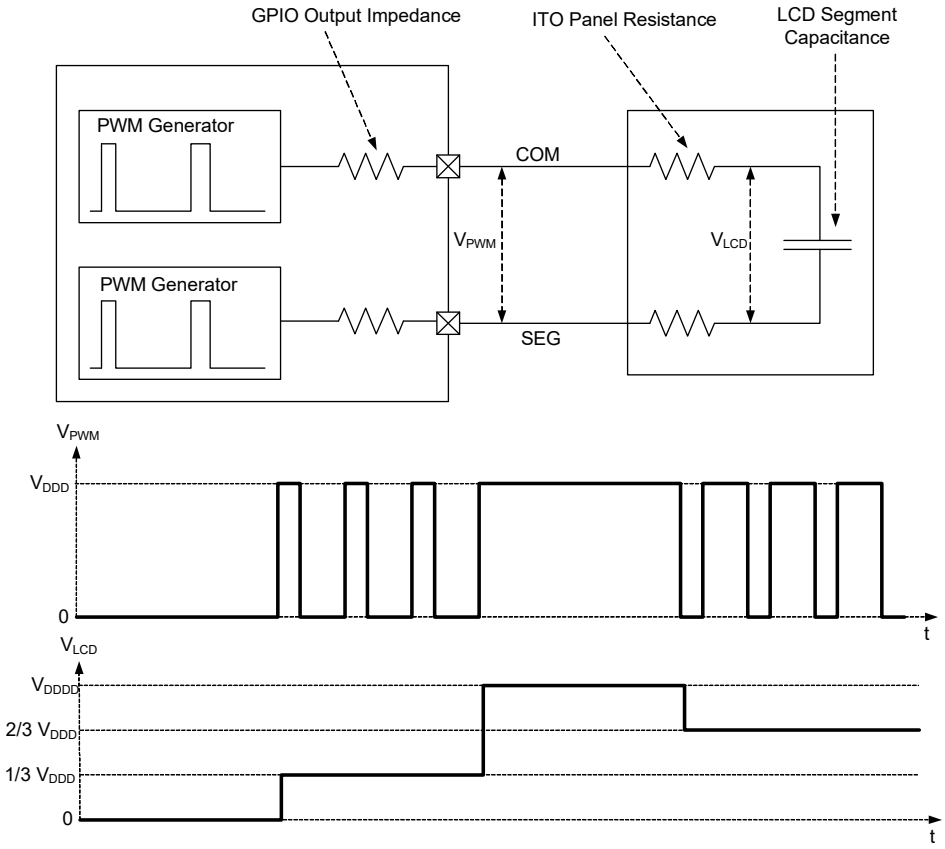
The PSoC 6 MCU supports the following drive modes.

- PWM drive at 1/2 bias
- PWM drive at 1/3 bias
- PWM drive at 1/4 bias with high-frequency clock input
- Digital correlation

35.2.2.1 PWM Drive

In PWM drive mode, multi-voltage drive signals are generated using a PWM output signal together with the intrinsic resistance and capacitance of the LCD. Figure 35-1 illustrates this.

Figure 35-1. PWM Drive (at 1/3 Bias)



The output waveform of the drive electronics is a PWM waveform. With the Indium Tin Oxide (ITO) panel resistance and the segment capacitance to filter the PWM, the voltage across the LCD segment is an analog voltage, as shown in Figure 35-1. This figure illustrates the generation of a 1/3 bias waveform (four commons and voltage steps of $V_{DD}/3$). See the Clocking System chapter on page 242 for details.

The PWM is derived from either CLK_LF (32 kHz, low-speed operation) or CLK_PERI (high-speed operation). See the Clocking System chapter on page 242 for more details of peripheral and low-frequency clocks. The filtered analog voltage across the LCD segments typically runs at low frequency for segment LCD driving.

Figure 35-2 and Figure 35-3 illustrate the Type A and Type B waveforms for COM and SEG electrodes for 1/2 bias and 1/4 duty. Only COM0/COM1 and SEG0/SEG1 are drawn for demonstration purpose. Similarly, Figure 35-4 and Figure 35-5 illustrate the Type A and Type B waveforms for COM and SEG electrodes for 1/3 bias and 1/4 duty.

Figure 35-2. PWM1/2 Type-A Waveform Example

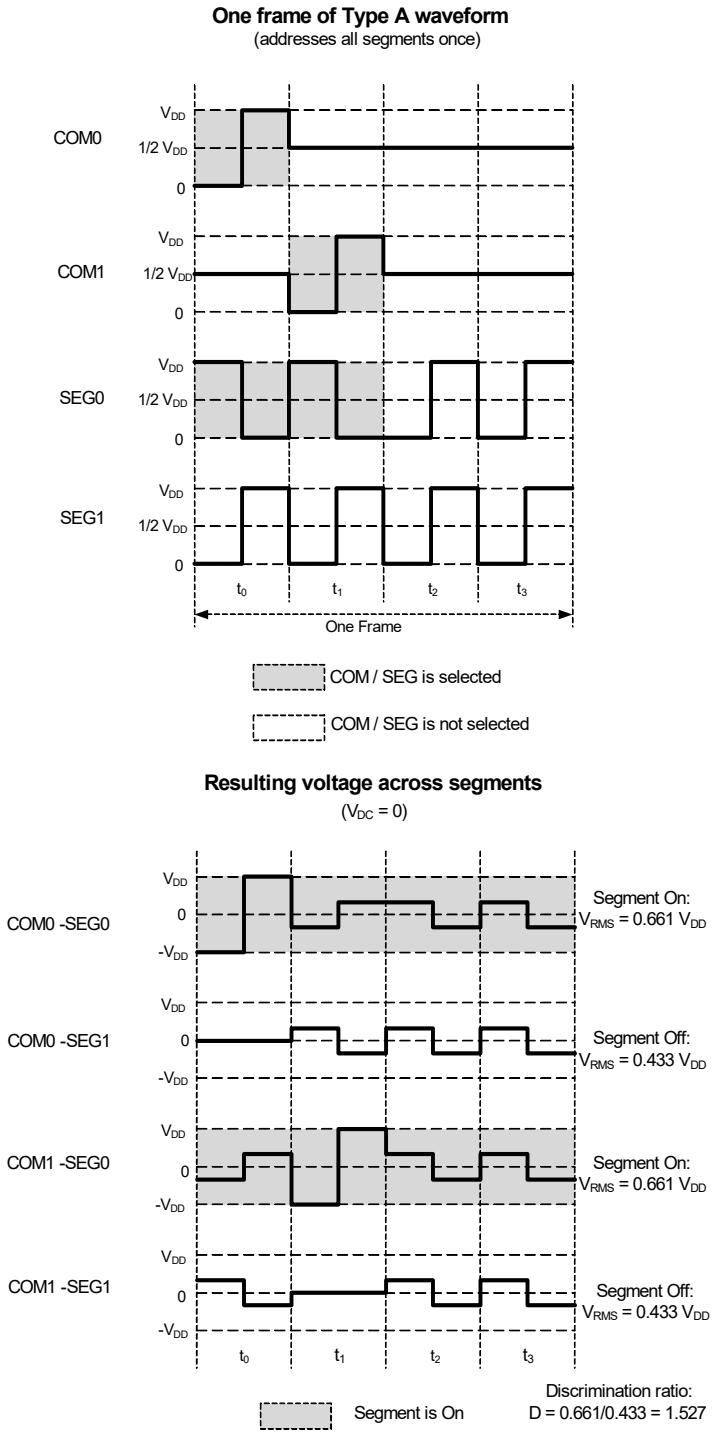


Figure 35-3. PWM1/2 Type-B Waveform Example

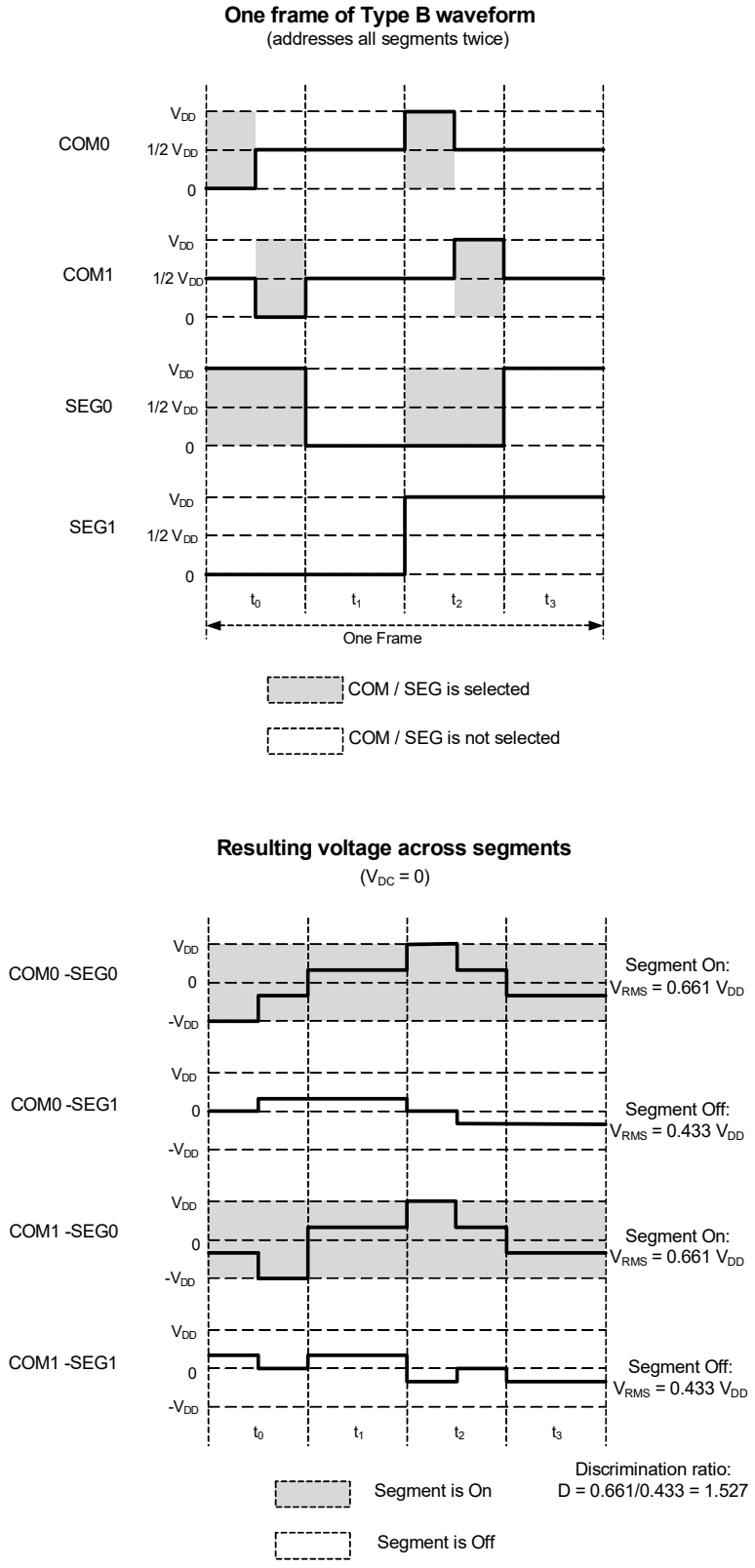


Figure 35-4. PWM1/3 Type-A Waveform Example

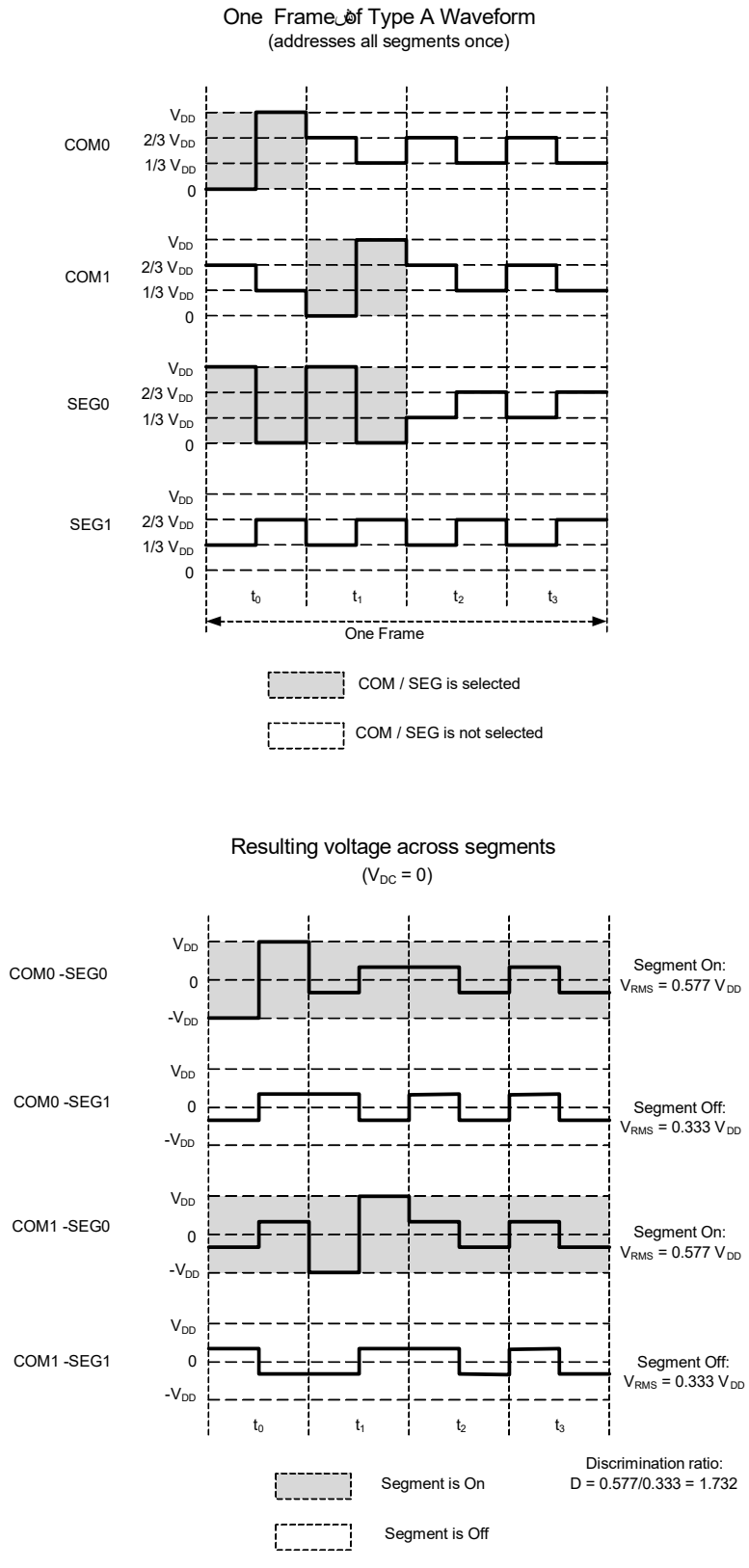
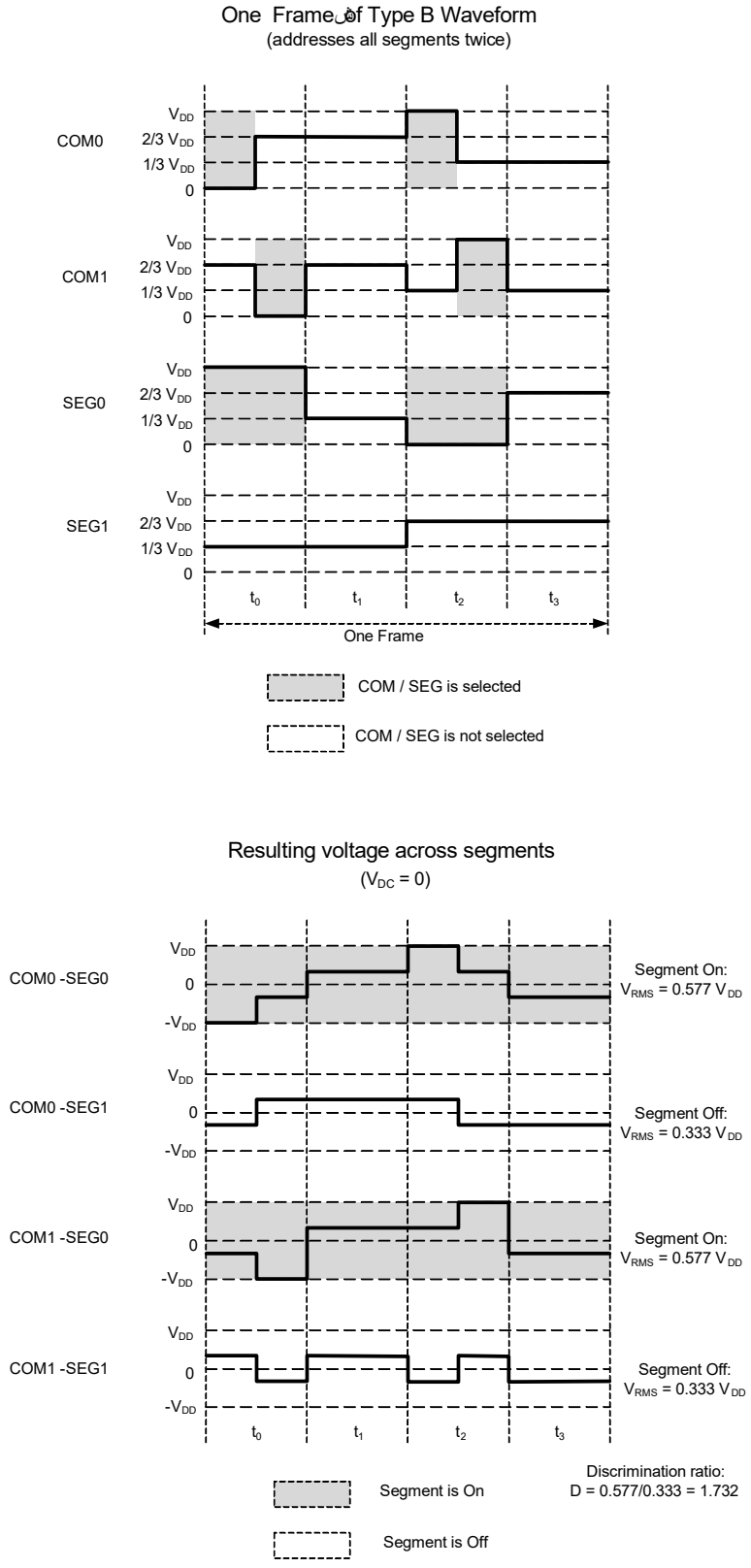


Figure 35-5. PWM1/3 Type-B Waveform Example



The effective RMS voltage for ON and OFF segments can be calculated easily using these equations:

$$V_{RMS(OFF)} = \sqrt{\frac{2(B-2)^2 + 2(M-1)}{2M}} x \left(\frac{V_{DRV}}{B}\right)$$

Equation 35-1

$$V_{RMS(ON)} = \sqrt{\frac{2B^2 + 2(M-1)}{2M}} x \left(\frac{V_{DRV}}{B}\right)$$

Equation 35-2

Where B is the bias and M is the duty (number of COMs).

For example, if the number of COMs is four, the resulting discrimination ratios (D) for 1/2 and 1/3 biases are 1.528 and 1.732, respectively. 1/3 bias offers better discrimination ratio in two and three COM drives also. Therefore, 1/3 bias offers better contrast than 1/2 bias and is recommended for most applications. 1/4 bias is available only in high-speed operation of the LCD. They offer better discrimination ratio especially when used with high COM designs (more than four COMs).

When the low-speed operation of LCD is used, the PWM signal is derived from the 32-kHz CLK_LF. To drive a low-capacitance display with acceptable ripple and rise/fall times using a 32-kHz PWM, additional external series resistances of 100 k-1 MΩ should be used. External resistors are not required for PWM frequencies greater than ~1 MHz. The ideal PWM frequency depends on the capacitance of the display and the internal ITO resistance of the ITO routing traces.

The 1/2 bias mode has the advantage that PWM is only required on the COM signals; the SEG signals use only logic levels, as shown in Figure 35-2 and Figure 35-3.

35.2.2.2 Digital Correlation

The digital correlation mode, instead of generating bias voltages between the rails, takes advantage of the characteristic of LCDs that the contrast of LCD segments is determined by the RMS voltage across the segments. In this approach, the correlation coefficient between any given pair of COM and SEG signals determines whether the corresponding LCD segment is on or off. Thus, by doubling the base drive frequency of the COM signals in their inactive sub-frame intervals, the phase relationship of the COM and SEG drive signals can be varied to turn segments on and off. This is different from varying the DC levels of the signals as in the PWM drive approach. Figure 35-8 and Figure 35-9 are example waveforms that illustrate the principles of operation.

Figure 35-6. Digital Correlation Type-A Waveform

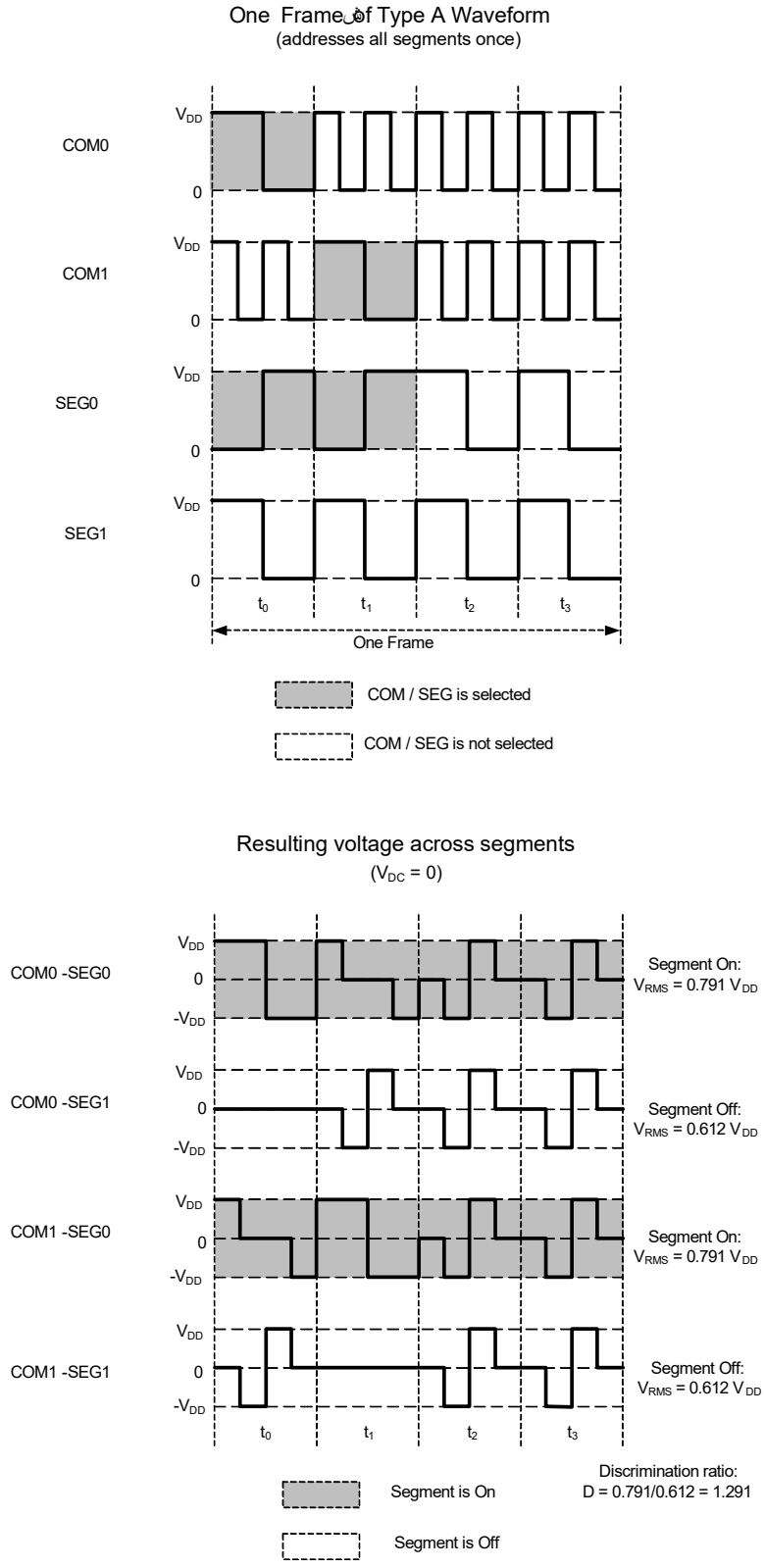
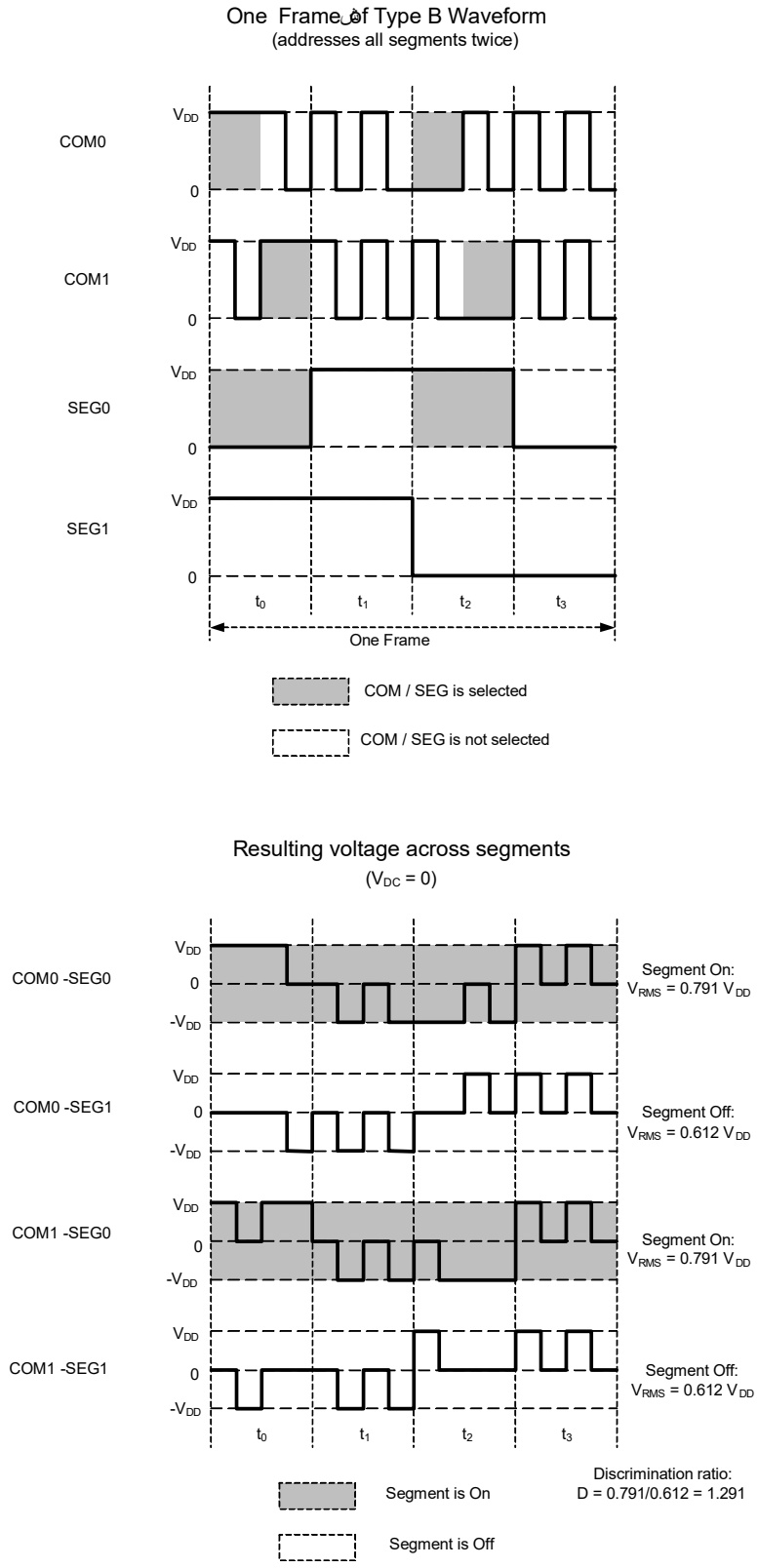


Figure 35-7. Digital Correlation Type-B Waveform



The RMS voltage applied to on and off segments can be calculated as follows:

$$V_{RMS(OFF)} = \sqrt{\frac{(M-1)}{2M}} x(V_{DD})$$

$$V_{RMS(ON)} = \sqrt{\frac{2+(M-1)}{2M}} x(V_{DD})$$

Where B is the bias and M is the duty (number of COMs). This leads to a discrimination ratio (D) of 1.291 for four COMs. Digital correlation mode also has the ability to drive 3-V displays from 1.8-V V_{DD}.

35.2.3 Recommended Usage of Drive Modes

The PWM drive mode has higher discrimination ratios compared to the digital correlation mode, as explained in 35.2.2.1 PWM Drive and 35.2.2.2 Digital Correlation. Therefore, the contrast in digital correlation method is lower than PWM method but digital correlation has lower power consumption because its waveforms toggle at low frequencies.

The digital correlation mode creates reduced, but acceptable contrast on TN displays, but no noticeable difference in contrast or viewing angle on higher contrast STN displays. Because each mode has strengths and weaknesses, recommended usage is as follows.

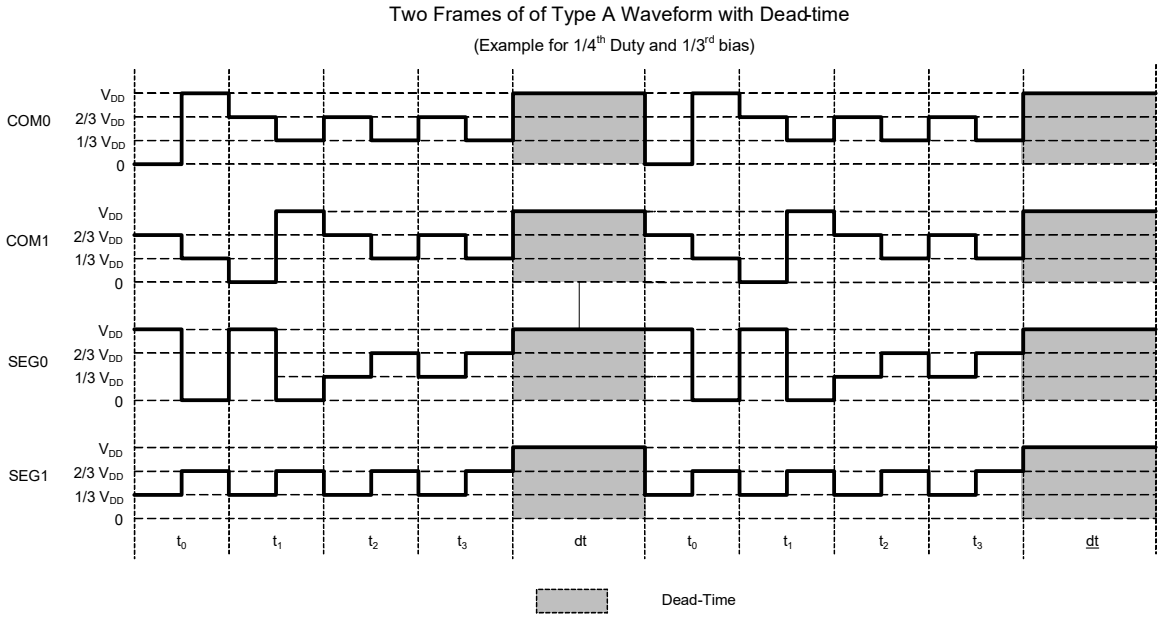
Table 35-1. Recommended Usage of Drive Modes

Display Type	Deep Sleep Mode	Sleep/Active Mode	Notes
Four COM TN Glass	Digital correlation	PWM 1/3 bias	Firmware must switch between LCD drive modes before going to deep sleep or waking up.
Four COM STN Glass	Digital correlation		No contrast advantage for PWM drive with STN glass.
Eight COM, STN	Not supported	PWM 1/4 bias	Supported only in the high-speed LCD mode. The low-speed CLK_LF is not fast enough to make the PWM work at high multiplex ratios.

35.2.4 Digital Contrast Control

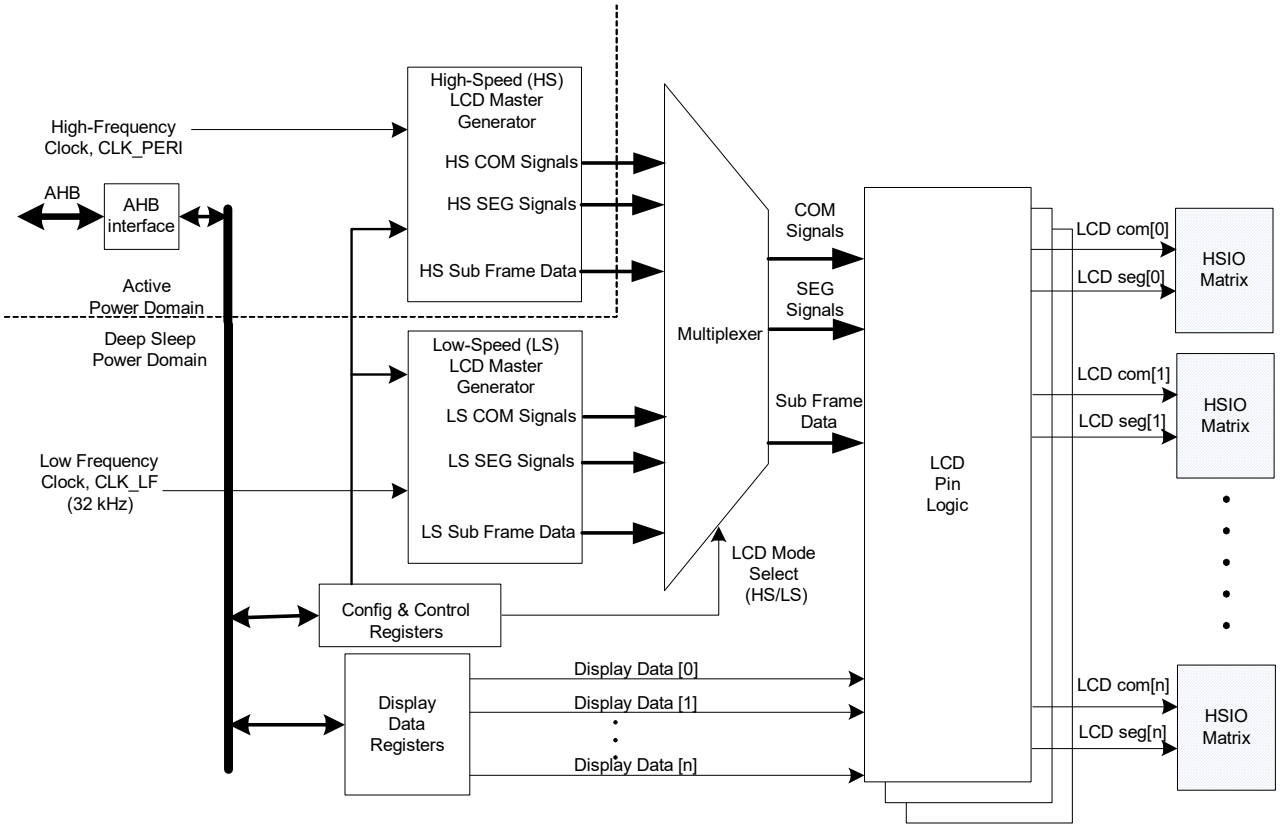
In all drive modes, digital contrast control can be used to change the contrast level of the segments. This method reduces contrast by reducing the driving time of the segments. This is done by inserting a 'Dead-Time' interval after each frame. During dead time, all COM and SEG signals are driven to a logic 1 state. The dead time can be controlled in fine resolution. Figure 35-8 illustrates the dead-time contrast control method for 1/3 bias and 1/4 duty implementation.

Figure 35-8. Dead-Time Contrast Control



35.3 PSoC 6 MCU Segment LCD Direct Drive

Figure 35-9. Block Diagram of LCD Direct Drive System



The LCD controller block contains two generators, one with a high-speed clock source CLK_PERI and the other with a low-speed clock source (32 kHz) derived from the CLK_LF. These are called high-speed LCD master generator and low-speed LCD master generator, respectively. Both the generators support PWM and digital correlation drive modes. PWM drive mode with low-speed generator requires external resistors, as explained in PWM Drive on page 484.

The multiplexer selects one of these two generator outputs to drive LCD, as configured by the firmware. The LCD pin logic block routes the COM and SEG outputs from the generators to the corresponding I/O matrices. Any GPIO can be used as either COM or SEG. This configurable pin assignment for COM or SEG is implemented in GPIO and I/O matrix; see High-Speed I/O Matrix. These two generators share the same configuration registers. These memory mapped I/O registers are connected to the system bus (AHB) using an AHB interface.

The LCD controller works in three device power modes: Active, Sleep, and Deep Sleep. High-speed operation is supported in Active and Sleep modes. Low-speed operation is supported in Active, Sleep, and Deep Sleep modes. The LCD controller is unpowered in Hibernate mode.

35.3.1 High-Speed and Low-Speed Master Generators

The high-speed and low-speed master generators are similar to each other. The only exception is that the high-speed version has larger frequency dividers to generate the frame and sub-frame periods. The high-speed generator is in the active power domain and the low-speed generator is in the Deep Sleep power domain. A single set of configuration registers is provided to control both high-speed and low-speed blocks. Each master generator has the following features and characteristics:

- Register bit configuring the block for either Type A or Type B drive waveforms (LCD_MODE bit in LCD0_CONTROL register).
- Register bits to select the number of COMs (COM_NUM field in LCD0_CONTROL register).
- Operating mode configuration bits enabled to select one of the following:
  - Digital correlation
  - PWM 1/2 bias
  - PWM 1/3 bias

- PWM 1/4 bias (not supported in low-speed generator)
- Off/disabled. Typically, one of the two generators will be configured to be Off

OP_MODE and BIAS fields in LCD0_CONTROL bits select the drive mode.

- A counter to generate the sub-frame timing. The SUBFR_DIV field in the LCD0_DIVIDER register determines the duration of each sub-frame. If the divide value written into this counter is C, the sub-frame period is 4 × (C+1). The low-speed generator has an 8-bit counter. This counter generates a maximum half sub-frame period of 8 ms from the 32-kHz CLK_LF. The high-speed generator has a 16-bit counter.
- A counter to generate the dead time period. These counters have the same number of bits as the sub-frame period counters and use the same clocks. DEAD_DIV field in the LCD0_DIVIDER register controls the dead time period.

35.3.2 Multiplexer and LCD Pin Logic

The multiplexer selects the output signals of either high-speed or low-speed master generator blocks and feeds it to the LCD pin logic. This selection is controlled by the configuration and control register. The LCD pin logic uses the sub-frame signal from the multiplexer to choose the display data. This pin logic will be replicated for each LCD pin.

35.3.3 Display Data Registers

Each LCD segment pin is part of an LCD port with its own display data register, LCD0_DATAx. The device has eight such LCD ports. Note that these ports are not real pin ports but the ports/connections available in the LCD hardware for mapping the segments to commons. Each LCD segment configured is considered as a pin in these LCD ports. The LCD0_DATAxx registers are 32-bit wide and store the ON/OFF data for all SEG-COM combination enabled in the design. For example, LCD0_DATA0x holds SEG-COM data for COM0 to COM3 and LCD0_DATA1x holds SEG-COM data for COM4 to COM7. The bits [4i+3:4i] (where 'i' is the pin number) of each LCD0_DATAxx register represent the ON/OFF data for Pin[i], as shown in Table 35-2. The LCD0_DATAxx register should be programmed according to the display data of each frame. The display data registers are Memory Mapped I/O (MMIO) and accessed through the AHB slave interface. See the PSoC 61 datasheet/PSoC 62 datasheet for the pin connections.

Table 35-2. SEG-COM Mapping Example of LCD0_DATA00 Register (each SEG is a pin of the LCD port)

BITS[31:28] = PIN_7[3:0]				BITS[27:24] = PIN_6[3:0]			
PIN_7-COM3	PIN_7-COM2	PIN_7-COM1	PIN_7-COM0	PIN_6-COM3	PIN_6-COM2	PIN_6-COM1	PIN_6-COM0
BITS[23:20] = PIN_5[3:0]				**BITS[19:16] = PIN_4[3:0]**			
PIN_5-COM3	PIN_5-COM2	PIN_5-COM1	PIN_5-COM0	PIN_4-COM3	PIN_4-COM2	PIN_4-COM1	PIN_4-COM0
BITS[15:12] = PIN_3[3:0]				**BITS[11:8] = PIN_2[3:0]**			
PIN_3-COM3	PIN_3-COM2	PIN_3-COM1	PIN_3-COM0	PIN_2-COM3	PIN_2-COM2	PIN_2-COM1	PIN_2-COM0
BITS[7:3] = PIN_1[3:0]				**BITS[3:0] = PIN_0[3:0]**			
PIN_1-COM3	PIN_1-COM2	PIN_1-COM1	PIN_1-COM0	PIN_0-COM3	PIN_0-COM2	PIN_0-COM1	PIN_0-COM0

35.4 Register List

Table 35-3. LCD Direct Drive Register List

Register Name	Description
LCD0_ID	This register includes the information of LCD controller ID and revision number
LCD0_DIVIDER	This register controls the sub-frame and dead-time period
LCD0_CONTROL	This register is used to configure high-speed and low-speed generators
LCD0_DATA0x	LCD port pin data register for COM0 to COM3
LCD0_DATA1x	LCD port pin data register for COM4 to COM7

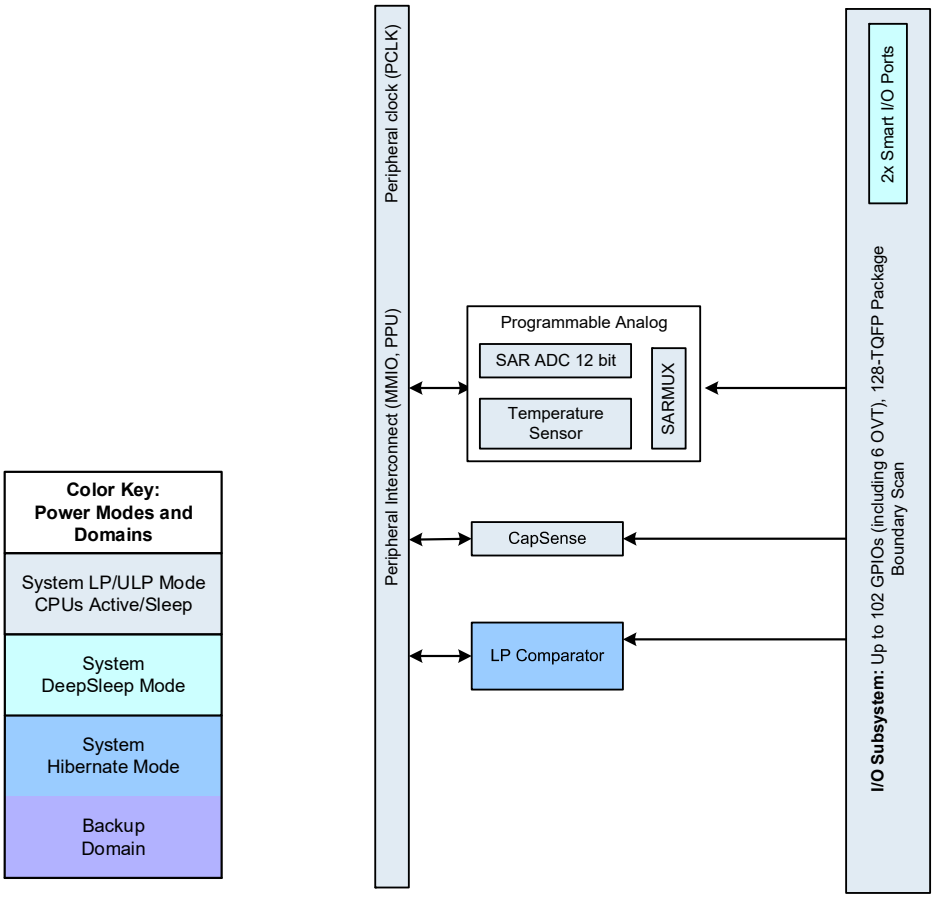
Section E: Analog Subsystem

This section encompasses the following chapters:

- Analog Reference Block chapter on page 497
- Low-Power Comparator chapter on page 501
- SAR ADC chapter on page 506
- Temperature Sensor chapter on page 520
- CapSense chapter on page 524

Top Level Architecture

Figure E-1. Analog System Block Diagram



36. Analog Reference Block

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The Analog Reference block (AREF) generates highly accurate reference voltage and currents needed by the programmable analog subsystem (PASS) and CapSense (CSD) blocks.

36.1 Features

- Provides accurate bandgap references for PASS and CSD subsystems
- 1.2-V voltage reference (V_{REF}) generator
- Option to output alternate voltage references routed from SRSS or from an external pin
- Zero dependency to absolute temperature (IZTAT) flat current reference generation, which is independent of temperature variations
- Option to generate IZTAT from SRSS generated current reference

36.2 Architecture

Figure 36-1. AREF Block Diagram

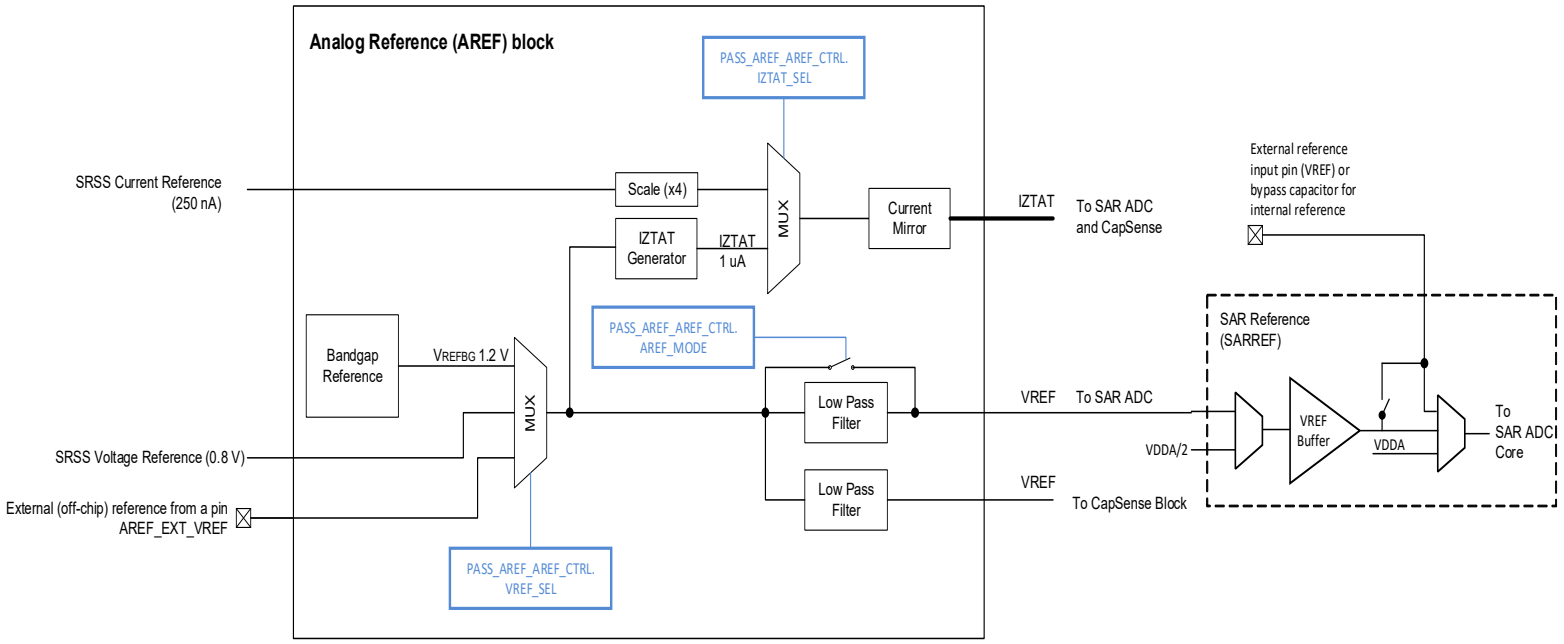


Figure 36-1 shows the architecture of the Analog Reference (AREF) block. It consists of a bandgap reference circuit, which generates 1.2 V voltage reference (VREFBG). Options for the voltage reference include 0.8 V reference from the SRSS block and an external reference from a dedicated pin (see the PSoC 61 datasheet/PSoC 62 datasheet for pin number). Application use cases of the selected voltage reference (V_{REF}) include setting the reference SAR ADC, and CapSense blocks.

The AREF block also generates zero-temperature coefficient IZTAT reference current (1 µA), which remains stable over temperature. The other option for IZTAT reference is the 1 µA current derived from the 250-nA reference current from SRSS. Selected IZTAT reference current is used in SAR ADC, and CapSense blocks.

Current mirror circuits are used to generate multiple current references to drive to different analog blocks.

The following sections explain the configurations in detail.

36.2.1 Bandgap Reference Block

The AREF block contains a local bandgap reference generator, which has tighter accuracy, temperature stability, and lower noise than the SRSS bandgap reference. The bandgap reference block provides a temperature stable voltage (V_{REFBG}). See the device datasheet to know the accuracy of V_{REFBG}.

The AREF block is enabled by writing '1' into the ENABLED bit of the PASS_AREF_AREF_CTRL register.

36.2.2 V_{REF} Reference Voltage Selection Multiplexer Options

The multiplexer in AREF is used to select the sources for the output voltage reference. The following options are available for selection using VREF_SEL bits of the PASS_AREF_AREF_CTRL register.

Table 36-1. Reference Voltage Multiplexer Options

VREF_SEL[1:0]	Description
00 (SRSS)	Routes the V_{REF} from SRSS (0.8 V)
01 (LOCAL)	Routes locally generated bandgap reference (1.2V)
02 (EXTERNAL)	Routes the reference from the external pin – AREF_EXT_VREF[a]

a. The AREF_EXT_VREF pin is different from the V_{REF} pin. V_{REF} pin is used by SAR ADC to take the reference directly bypassing the reference voltage from AREF.

36.2.3 Zero Dependency To Absolute Temperature Current Generator (I_{ZTAT})

The IZTAT current generator uses the output of the selected reference voltage (V_{REF}) to generate a precise current reference, which has a low variation over temperature.

Table 36-2. IZTAT Reference

IZTAT Reference Outputs	Value
IZTAT	1 µA

36.2.3.1 IZTAT Selection Multiplexer Options

IZTAT output current can be derived from the SRSS reference current (250 nA) or the reference current from the local IZTAT generator. The selection is made using IZTAT_SEL bit of the PASS_AREF_AREF_CTRL register.

Table 36-3. IZTAT Multiplexer Options

IZTAT_SEL	Description
0 (SRSS)	Uses SRSS current reference
1 (LOCAL)	Uses locally generated reference current

36.2.4 Startup Modes

AREF supports two startup modes, which provide a trade-off between wakeup time and noise performance when transitioning from the Deep Sleep to Active power mode. This is selected using the AREF_MODE bit of the PASS_AREF_AREF_CTRL register. The FAST_START mode enables faster wakeup, but with higher noise levels. Firmware can switch to NORMAL mode after the FAST_START settling time to achieve better noise performance.

Table 36-4. Startup Modes

AREF_MODE	Description
0 (NORMAL)	Normal startup mode
1 (FAST_START)	Fast startup mode

36.3 Registers

Table 36-5. List of AREF Registers

Register	Comment	Features
PASS_AREF_AREF_CTRL	AREF control register	Reference selection, startup time, and low-power mode.

37. Low-Power Comparator

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

PSoC 6 MCUs have two Low-Power comparators, which can perform fast analog signal comparison of internal and external analog signals in all system power modes. Low-Power comparator output can be inspected by the CPU, used as an interrupt/wakeup source to the CPU when in CPU Sleep mode, used as a wakeup source to system resources when in System Deep Sleep or Hibernate mode, or fed to GPIO or trigger multiplexer (see Trigger Multiplexer Block chapter on page 294) as an asynchronous or synchronous signal (level or pulse).

37.1 Features

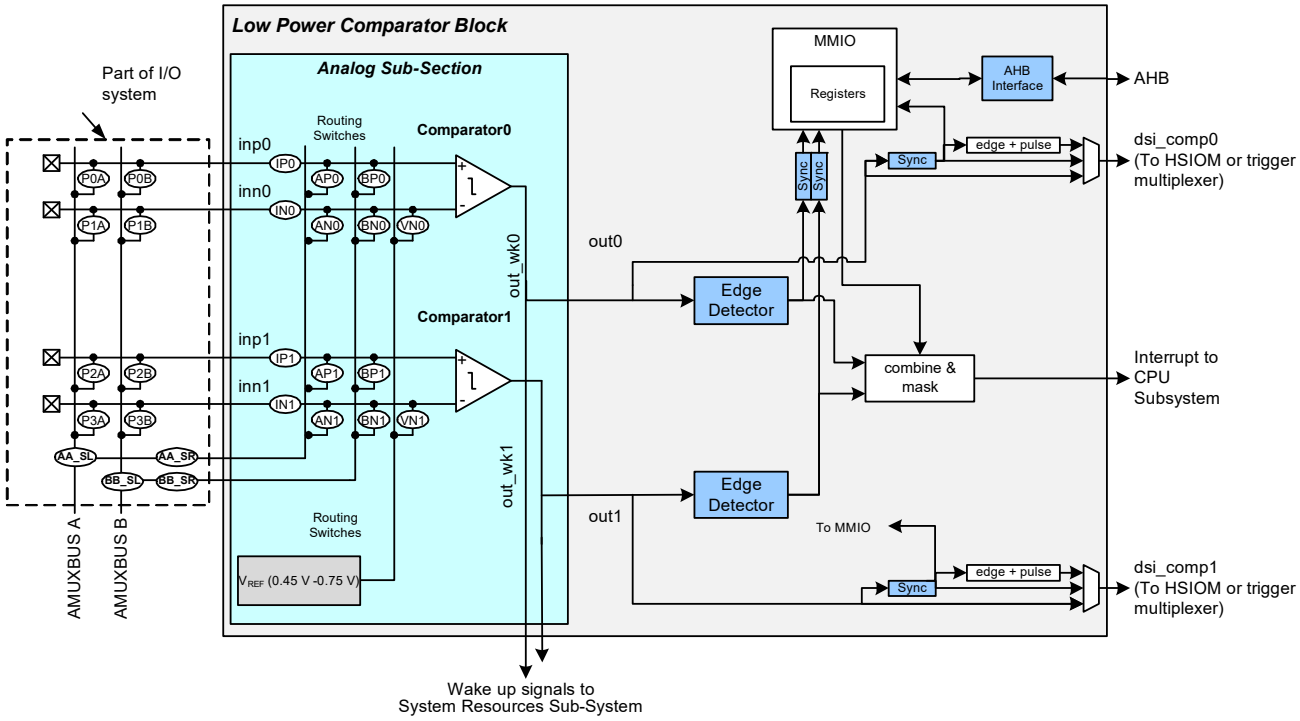
The PSoC 6 MCU comparators have the following features:

- Configurable input pins
- Programmable power and speed
- Ultra low-power mode support
- Each comparator features a one-sided hysteresis option
- Rising edge, falling edge, combined rising and falling edge detection at the comparator output
- Local reference voltage generation
- Wakeup source from low-power modes

37.2 Architecture

Figure 37-1 shows the block diagram for the Low-Power comparator.

Figure 37-1. Low-Power Comparator Block Diagram



The following sections describe the operation of the PSoC 6 MCU Low-Power comparator, including input configuration, power and speed modes, output and interrupt configuration, hysteresis, and wakeup from low-power modes.

37.2.1 Input Configuration

Low-Power comparators can operate with the following input options:

- Compare two voltages from external pins.
- Compare a voltage from an external pin against an internally generated analog-signal (through AMUXBUS).
- Compare two internal voltages through AMUXBUS-A/ AMUXBUS-B.
- Compare internal and external signals with a locally-generated reference voltage. Note that this voltage is not a precision reference and can vary from 0.45 V–0.75 V.

See the PSoC 61 datasheet/PSoC 62 datasheet for detailed specifications of the Low-Power comparator.

Note that AMUXBUS connections are not available in Deep Sleep and Hibernate modes. If Deep Sleep or Hibernate operation is required, the Low-Power comparator must be connected to the dedicated pins. This restriction also includes routing of any internally-generated signal, which uses the AMUXBUS for the connection. See the I/O System chapter on page 261 for more details on connecting the GPIO to AMUXBUS A/B or setting up the GPIO for comparator input.

Refer to the LPCOMP_CMP0_SW, LPCOMP_CMP1_SW, LPCOMP_CMP0_SW_CLEAR, and LPCOMP_CMP1_SW_CLEAR registers in the registers TRM to understand how to control the internal routing switches shown in Figure 37-1.

If the inverting input of a comparator is routed to a local voltage reference, the LPREF_EN bit in the LPCOMP_CONFIG register must be set to enable the voltage reference.

37.2.2 Output and Interrupt Configuration

Both Comparator0 and Comparator1 have hardware outputs available at dedicated pins. See the PSoC 61 datasheet/ PSoC 62 datasheet for the location of comparator output pins.

Firmware readout of Comparator0 and Comparator1 outputs are available at the OUT0 and OUT1 bits of the LPCOMP_STATUS register (Table 37-1). The output of each

comparator is connected to a corresponding edge detector block. This block determines the edge that triggers the interrupt. The edge selection and interrupt enable is configured using the INTTYPE0 and INTTYPE1 bitfields in the LPCOMP_CMP0_CTRL and LPCOMP_CMP1_CTRL registers for Comparator0 and Comparator1, respectively. Using the INTTYPE0 and INTTYPE1 bits, the interrupt type can be selected to disabled, rising edge, falling edge, or both edges, as described in Table 37-1.

Each comparator's output can also be routed directly to a GPIO pin through the HSIOM. See the I/O System chapter on page 261 for more details.

During an edge event, the comparator will trigger an interrupt. The interrupt request is registered in the COMP0 bit and COMP1 bit of the LPCOMP_INTR register for Comparator0 and Comparator1, respectively. Both Comparator0 and Comparator1 share a common interrupt signal output (see Figure 37-1), which is a logical OR of the two interrupts and mapped as the Low-Power comparator block's interrupt in the CPU NVIC. Refer to the Interrupts chapter on page 56 for details. If both the comparators are used in a design, the COMP0 and COMP1 bits of the LPCOMP_INTR register must be read in the interrupt service routine to know which one triggered the interrupt. Alternatively, COMP0_MASK bit and COMP1_MASK bit of the LPCOMP_INTR_MASK register can be used to mask the Comparator0 and Comparator1 interrupts to the CPU. Only the masked interrupts will be serviced by the CPU. After the interrupt is processed, the interrupt should be cleared by writing a '1' to the COMP0 and COMP1 bits of the LPCOMP_INTR register in firmware. If the interrupt to the CPU is not cleared, it stays active regardless of the next compare events and interrupts the CPU continuously. Refer to the Interrupts chapter on page 56 for details.

LPCOMP_INTR_SET register bits [1:0] can be used to assert an interrupt for firmware debugging.

In Deep Sleep mode, the wakeup interrupt controller (WIC) can be activated by a comparator edge event, which then wakes up the CPU. Similarly in Hibernate mode, the LPCOMP can wake up the system resources sub-system. Thus, the LPCOMP has the capability to monitor a specified signal in low-power modes. See the Power Supply and Monitoring chapter on page 218 and the Device Power Modes chapter on page 225 for more details.

Table 37-1. Output and Interrupt Configuration

Register[Bit_Pos]	Bit_Name	Description
LPCOMP_STATUS[0]	OUT0	Current/Instantaneous output value of Comparator0
LPCOMP_STATUS[16]	OUT1	Current/Instantaneous output value of Comparator1
LPCOMP_CMP0_CTRL[7:6]	INTTYPE0	Sets on which edge Comparator0 will trigger an IRQ 00: Disabled 01: Rising Edge 10: Falling Edge 11: Both rising and falling edges
LPCOMP_CMP1_CTRL[7:6]	INTTYPE1	Sets on which edge Comparator1 will trigger an IRQ 00: Disabled 01: Rising Edge 10: Falling Edge 11: Both rising and falling edges
LPCOMP_INTR[0]	COMP0	Comparator0 Interrupt: hardware sets this interrupt when Comparator0 triggers. Write a '1' to clear the interrupt
LPCOMP_INTR[1]	COMP1	Comparator1 Interrupt: hardware sets this interrupt when Comparator1 triggers. Write a '1' to clear the interrupt
LPCOMP_INTR_SET[0]	COMP0	Write a '1' to trigger the software interrupt for Comparator0
LPCOMP_INTR_SET[1]	COMP1	Write a 1 to trigger the software interrupt for Comparator1

37.2.3 Power Mode and Speed Configuration

The Low-Power comparators can operate in three power modes:

- Normal
- Low-power
- Ultra low-power

The power or speed setting for Comparator0 is configured using the MODE0 bitfield of the LPCOMP_CMP0_CTRL register. Similarly, the power or speed setting for Comparator1 is configured using the MODE1 bitfield of the LPCOMP_CMP1_CTRL register. The power consumption and response time vary depending on the selected power mode; power consumption is highest in fast mode and lowest in ultra-low-power mode, response time is fastest in fast mode and slowest in ultra-low-power mode. Refer to the PSoC 61 datasheet/PSoC 62 datasheet for specifications for the response time and power consumption for various power settings.

The comparators can also be enabled or disabled using these bitfields, as described in Table 37-2.

Note: The output of the comparator may glitch when the power mode is changed while comparator is enabled. To avoid this, disable the comparator before changing the power mode.

Table 37-2. Comparator Power Mode Selection Bits

Register[Bit_Pos]	Bit_Name	Description
LPCOMP_CMP0_CTRL[1:0]	MODE0	Compartor0 power mode selection 00: Off 01: Ultra low-power operating mode. This mode must be used when the device is in Deep Sleep or Hibernate mode. 10: Low-power operating mode 11: Normal, full-speed, full-power operating mode See the datasheet for electrical specifications in each power mode.
LPCOMP_CMP1_CTRL[1:0]	MODE1	Compartor1 power mode selection 00: Off 01: Ultra low-power operating mode. This mode must be used when the device is in Deep Sleep or Hibernate mode. 10: Low-power operating mode 11: Normal, full-speed, full-power operating mode See the datasheet for electrical specifications in each power mode.

Additionally, the entire Low-Power comparator system can be enabled or disabled globally using the LPCOMP_CONFIG[31] bit. See the registers TRM for details of these bitfields.

37.2.4 Hysteresis

For applications that compare signals close to each other and slow changing signals, hysteresis helps to avoid oscillations at the comparator output when the signals are noisy. For such applications, a fixed hysteresis may be enabled in the comparator block. See the PSoC 61 datasheet/PSoC 62 datasheet for the hysteresis voltage range.

The hysteresis level is enabled/disabled by using the HYST0 and HYST1 bitfields in the LPCOMP_CMP0_CTRL and LPCOMP_CMP1_CTRL registers for Comparator0 and Comparator1, as described in Table 37-3.

Table 37-3. Hysteresis Control Bits

Register[Bit_Pos]	Bit_Name	Description
LPCOMP_CMP0_CTRL[5]	HYST0	Enable/Disable hysteresis to Comparator0 1: Enable Hysteresis 0: Disable Hysteresis See the datasheet for hysteresis voltage range.
LPCOMP_CMP1_CTRL[5]	HYST1	Enable/Disable hysteresis to Comparator1 1: Enable Hysteresis 0: Disable Hysteresis See the datasheet for hysteresis voltage range.

37.2.5 Wakeup from Low-Power Modes

The comparator is operational in the device's low-power modes, including Sleep, Deep Sleep, and Hibernate modes. The comparator output can wake the device from Sleep and Deep Sleep modes. The comparator output can generate interrupts in Deep Sleep mode when enabled in the LPCOMP_CONFIG register, the INTTYPEx bits in the LPCOMP_CMPx_CTRL register should not be set to disabled, and the INTR_MASKx bit should be set in the LPCOMP_INTR_MASK register for the corresponding comparator to wake the device from low-power modes. Comparisons involving AMUXBUS connections are not available in Deep Sleep and Hibernate modes. Moreover, if the comparator is required in Deep Sleep or Hibernate modes, then it must be configured into Ultra Low-Power mode before the device is put into Deep Sleep or Hibernate mode.

In the Deep Sleep and Hibernate power modes, a compare event on either Comparator0 or Comparator1 output will generate a wakeup interrupt. The INTTYPEx bits in the LPCOMP_CONFIG register should be properly configured. The mask bits in the LPCOMP_INTR_MASK register is used to select whether one or both of the comparator's interrupt is serviced by the CPU. See the Device Power Modes chapter on page 225 for more details.

37.2.6 Comparator Clock

The comparator uses the system main clock SYSCLK as the clock for interrupt and output synchronization. See the Clocking System chapter on page 242 for more details.

37.3 Register List

Table 37-4. Low-Power Comparator Register Summary

Register	Function
LPCOMP_CONFIG	LPCOMP global configuration register
LPCOMP_INTR	LPCOMP interrupt register
LPCOMP_INTR_SET	LPCOMP interrupt set register
LPCOMP_INTR_MASK	LPCOMP interrupt request mask register
LPCOMP_INTR_MASKED	LPCOMP masked interrupt output register
LPCOMP_STATUS	Output status register
LPCOMP_CMP0_CTRL	Comparator0 configuration register
LPCOMP_CMP1_CTRL	Comparator1 configuration
LPCOMP_CMP0_SW	Comparator0 switch control
LPCOMP_CMP1_SW	Comparator1 switch control
LPCOMP_CMP0_SW_CLEAR	Comparator0 switch control clear
LPCOMP_CMP1_SW_CLEAR	Comparator1 switch control clear

38. SAR ADC

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

The PSoC 6 MCU has a 12-bit successive approximation register analog-to-digital converter (SAR ADC). The 12-bit, 1-Msps SAR ADC is designed for applications that require moderate resolution and high data rate.

38.1 Features

- Maximum sample rate of 1 Msps
- Sixteen individually configurable logical channels that can scan eleven unique input channels. Each channel has the following features:
  - Input from eight dedicated pins (eight single-ended mode or four differential inputs) or internal signals (AMUXBUS or temperature sensor)
  - Each channel may choose one of the four programmable acquisition times to compensate for external factors (such as high input impedance sources with long settling times)
  - Single-ended or differential measurement
  - Averaging and accumulation
  - Double-buffered results
- Result may be left- or right-aligned, or may be represented in 16-bit sign extended
- Scan can be triggered by firmware, trigger from other peripherals or pins
  - One-shot – periodic or continuous mode
- Hardware averaging support
  - First order accumulate
  - Supports 2, 4, 8, 16, 32, 64, 128, and 256 samples (powers of 2)
- Selectable voltage references
  - Internal V_{DDA} and $V_{DDA}/2$ references
  - Internal 1.2-V reference with buffer
  - External reference
- Interrupt generation
  - End of scan
  - Saturation detect and over-range (configurable) detect for every channel
  - Scan results overflow
  - Collision detect

38.2 Architecture

Figure 38-1. Block Diagram

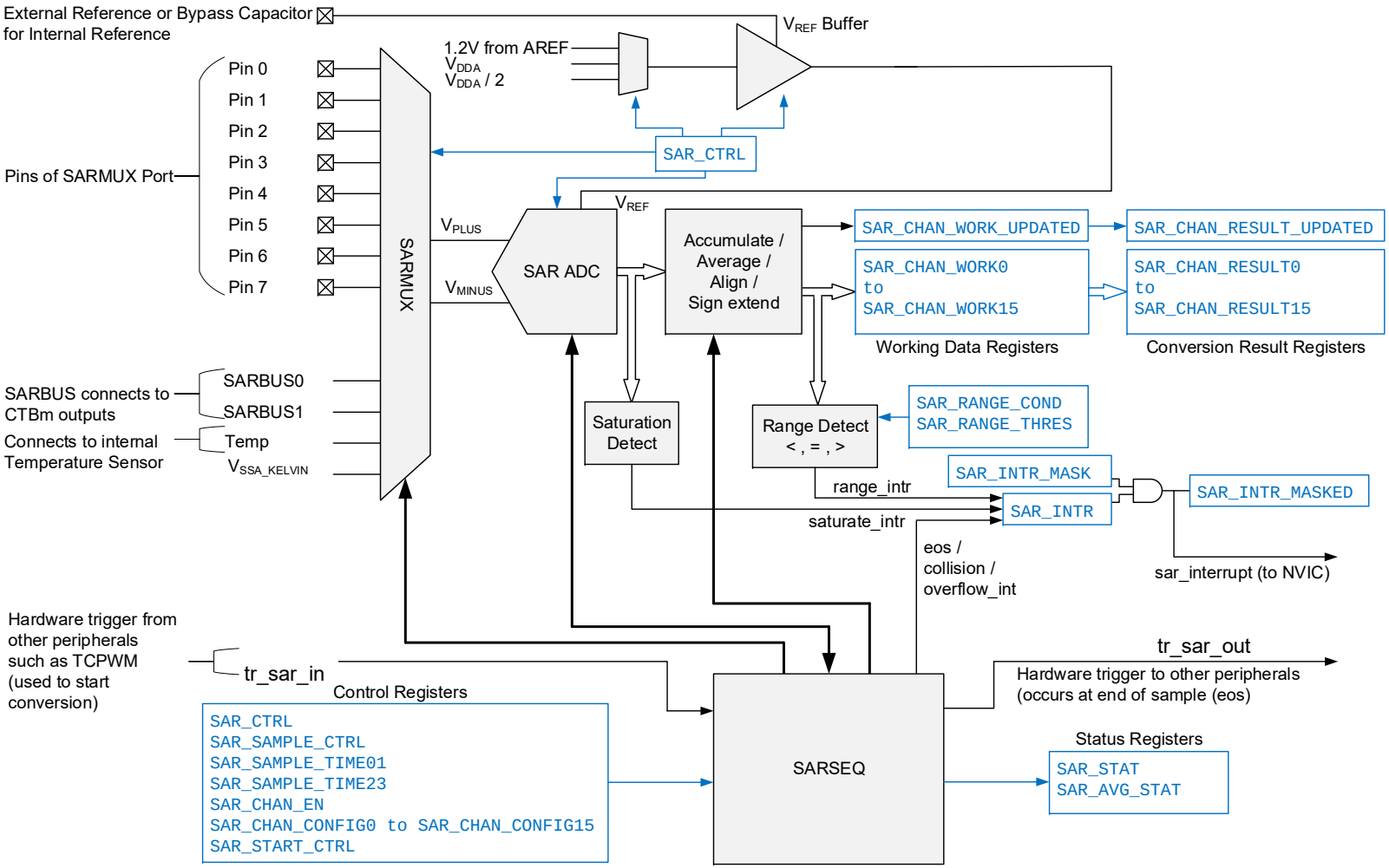


Figure 38-1 shows the simplified block diagram of PSoC 6 MCU SAR ADC system, with important registers shown in blue. Preceding the SAR ADC is the SARMUX, which can route external pins and internal signals (AMUXBUS A/ AMUXBUS B or temperature sensor output) to the internal channels of SAR ADC. The sequencer controller (SARSEQ) is used to control SARMUX and SAR ADC to do an automatic scan on all enabled channels without CPU intervention. SARSEQ also performs pre-processing such as averaging and accumulating the output data.

The result from each channel is double-buffered and a complete scan may be configured to generate an interrupt at the end of the scan. The sequencer may also be configured to flag overflow, collision, and saturation errors that can be configured to assert an interrupt.

38.2.1 SAR ADC Core

The PSoC 6 MCU SAR ADC core is a 12-bit SAR ADC. The maximum sample rate for this ADC is 1 Msps. The SAR ADC core has the following features:

- Fully differential architecture; also supports single-ended mode
- 12-bit resolution
- Four programmable acquisition times
- Seven programmable power levels
- Supports single and continuous conversion mode

SAR_CTRL register contains the bitfields that control the operation of SAR ADC core. See the registers TRM for more details of this register.

38.2.1.1 Single-ended and Differential Modes

The PSoC 6 MCU SAR ADC can operate in single-ended and differential modes. Differential or single-ended mode can be configured using the DIFFERENTIAL_EN bitfield in the channel configuration register, SAR_CHAN_CONFIGx, where x is the channel number (0–15).

SAR ADC gives full range output (0 to 4095) for differential inputs in the range of $-V_{REF}$ to $+V_{REF}$.

Note: The precise value of the input range in the differential mode is $-V_{REF}$ to $(+V_{REF} - (V_{REF}/2047))$. The positive input range is limited by the resolution of the ADC.

The single-ended mode options of negative input include V_{SSA}, V_{REF}, or an external input from P1, P3, P5, or P7 pins of SARMUX. See the PSoC 61 datasheet/PSoC 62 datasheet for the exact location of SARMUX pins. This mode is configured by the NEG_SEL bitfield in the global configuration register SAR_CTRL. When V_{MINUS} is connected to these SARMUX pins, the single-ended mode is equivalent to differential mode. However, when the odd pin of each differential pair is connected to the common alternate ground, these conversions are 11-bit because measured signal value cannot go below ground.

To get a single-ended conversion with 12 bits, you must connect V_{REF} to the negative input of the SAR ADC; then, the input range can be from 0 to $2 \times V_{REF}$.

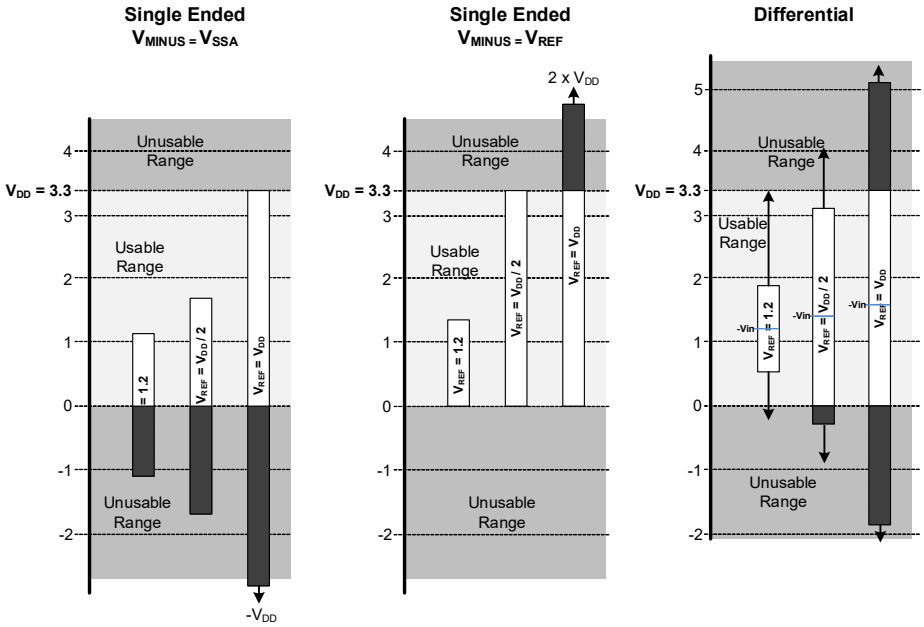
Note that temperature sensor can only be used in single-ended mode.

38.2.1.2 Input Range

All inputs should be in the range of V_{SSA} to V_{DDA}. Input voltage range is also limited by V_{REF}. If voltage on negative input is Vn and the ADC reference is V_{REF}, the range on the positive input is Vn ± V_{REF}. This criterion applies for both single-ended and differential modes. In single-ended mode, Vn is connected to V_{SSA}, V_{REF} or an external input.

Note that Vn ± V_{REF} should be in the range of V_{SSA} to V_{DDA}. For example, if negative input is connected to V_{SSA}, the range on the positive input is 0 to V_{REF}, not $-V_{REF}$ to V_{REF}. This is because the signal cannot go below V_{SSA}. Only half of the ADC range is usable because the positive input signal cannot swing below V_{SS}, which effectively only generates an 11-bit result.

Figure 38-2. Input Range



38.2.1.3 Result Data Format

Result data format is configurable from two aspects:

- Signed/unsigned
- Left/right alignment

When the result is considered signed, the most significant bit of the conversion is used for sign extension to 16 bits with MSb. For an unsigned conversion, the result is zero extended to 16-bits. The sample value can be either right-aligned or left-aligned within the 16 bits of the result register. By default, data is right-aligned in data[11:0], with sign extension to 16 bits, if required. Left-alignment will cause lower significant bits to be made zero.

Result data format can be controlled by DIFFERENTIAL_SIGNED, SINGLE_ENDED_SIGNED, and LEFT_ALIGN bitfields in the SAR_SAMPLE_CTRL register.

The result data format can be shown as follows.

Table 38-1. Result Data Format

Alignment	Signed/ Unsigned	Result Register															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Right	Unsigned	–	–	–	–	11	10	9	8	7	6	5	4	3	2	1	0
Right	Signed	11	11	11	11	11	10	9	8	7	6	5	4	3	2	1	0
Left	–	11	10	9	8	7	6	5	4	3	2	1	0	–	–	–	–

38.2.1.4 Negative Input Selection

The negative input connection choice affects the voltage range and effective resolution (Table 38-2). In single-ended mode, negative input of the SAR ADC can be connected to V_{SSA}, V_{REF}, or P1, P3, P5, or P7 pins of SARMUX. Vx is the common mode voltage.

Table 38-2. Negative Input Selection Comparison

Single-ended/Differential	Signed/Unsigned	V_{MINUS}	V_{PLUS} Range	Result Register
Single-ended	N/A[a]	V_{SSA}	$+V_{REF}$	0x7FF
			V_{SSA} = 0	0x000
Single-ended	Unsigned	V_{REF}	+2 × V_{REF}	0xFFF
			V_{REF}	0x800
			V_{SSA} = 0	0
Single-ended	Signed	V_{REF}	+2 × V_{REF}	0x7FF
			V_{REF}	0x000
			V_{SSA} = 0	0x800
Single-ended	Unsigned	Vx[b]	Vx + V_{REF}	0xFFF
			Vx	0x800
			Vx – V_{REF}	0
Single-ended	Signed	Vx	Vx + V_{REF}	0x7FF
			Vx	0x000
			Vx – V_{REF}	0x800
Differential	Unsigned	Vx	Vx + V_{REF}	0xFFF
			Vx	0x800
			Vx – V_{REF}	0
Differential	Signed	Vx	Vx + V_{REF}	0x7FF
			Vx	0x000
			Vx – V_{REF}	0x800

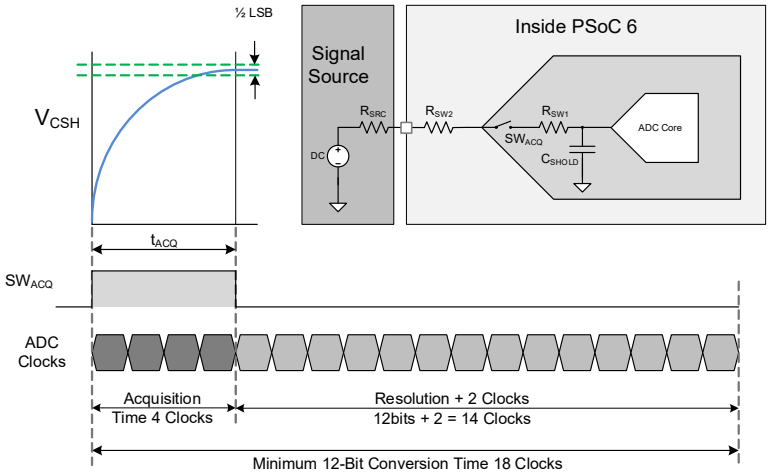
a. For single-ended mode with V_{MINUS} connected to V_{SSA}, conversions are effectively 11-bit because voltages cannot swing below V_{SSA} on any PSoC 6 MCU pin. Because of this, the global configuration bit SINGLE_ENDED_SIGNED (SAR_SAMPLE_CTRL[2]) will be ignored and the result is always (0x000-0x7FF).
b. Vx is the differential input common mode voltage.

Note that single-ended conversions with V_{MINUS} connected to the pins with SARMUX connectivity are electrically equivalent to differential mode. However, when the odd pin of each differential pair is connected to the common alternate ground, these conversions are 11-bit because measured signal value (SARMUX.vplus) cannot go below ground.

38.2.1.5 Acquisition Time

Acquisition time is the time taken by sample and hold (S/H) circuit inside SAR ADC to settle. After acquisition time, the input signal source is disconnected from the SARADC core, and the output of the S/H circuit will be used for conversion. Each channel can select one from four acquisition time options, from 4 to 1023 SAR clock cycles defined in global configuration registers SAR_SAMPLE_TIME01 and SAR_SAMPLE_TIME23. These two registers contain four programmable sample times ST0, ST1, ST2, and ST3. One of these four sample times can be selected for a particular channel by configuring the SAMPLE_TIME_SEL bitfield in the respective SAR_CHAN_CONFIGx register.

Figure 38-3. Acquisition Time



The acquisition time should be sufficient to charge the internal hold capacitor of the ADC through the resistance of the routing path, as shown in Figure 38-3. The recommended value of acquisition time is:

Tacq >= 9 × (Rsrc + Rmux + Racqsw) × Csh

Where:

- Rsrc = Source resistance
- Rmux = SARMUX switch resistance
- Racqsw = Sample and hold switch
- Rmux + Racqsw ~= 1000 ohms
- Csh ~= 10pF

This depends on the routing path (see Analog Routing on page 513 for details).

38.2.1.6 SAR ADC Clock

Note: The maximum SAR ADC clock frequency may be limited to less than 18 MHz by the PSoC 61 datasheet/PSoC 62 datasheet specification for sample rate maximum.

SAR ADC clock frequency must be between 1.8 MHz and 18 MHz, which comes from the peripheral clock (CLK_PERI) in the system resources subsystem (SRSS). See the Clocking System chapter on page 242 to know how to configure the peripheral clock.

38.2.1.7 SAR ADC Timing

As Figure 38-4 shows, an ADC conversion with the minimum acquisition time of four clocks requires 18 clocks to complete. Note that the minimum acquisition time of four clock cycles at 36 MHz is based on the minimum acquisition time supported by the SAR block (R_{SW1} and C_{SHOLD} in Figure 38-3 on page 511), which is 97 ns.

Total clock cycles for valid output are equal to:

4 clock cycles for sampling input (minimum acquisition time set by SAR_SAMPLE_TIME01 or SAR_SAMPLE_TIME23)

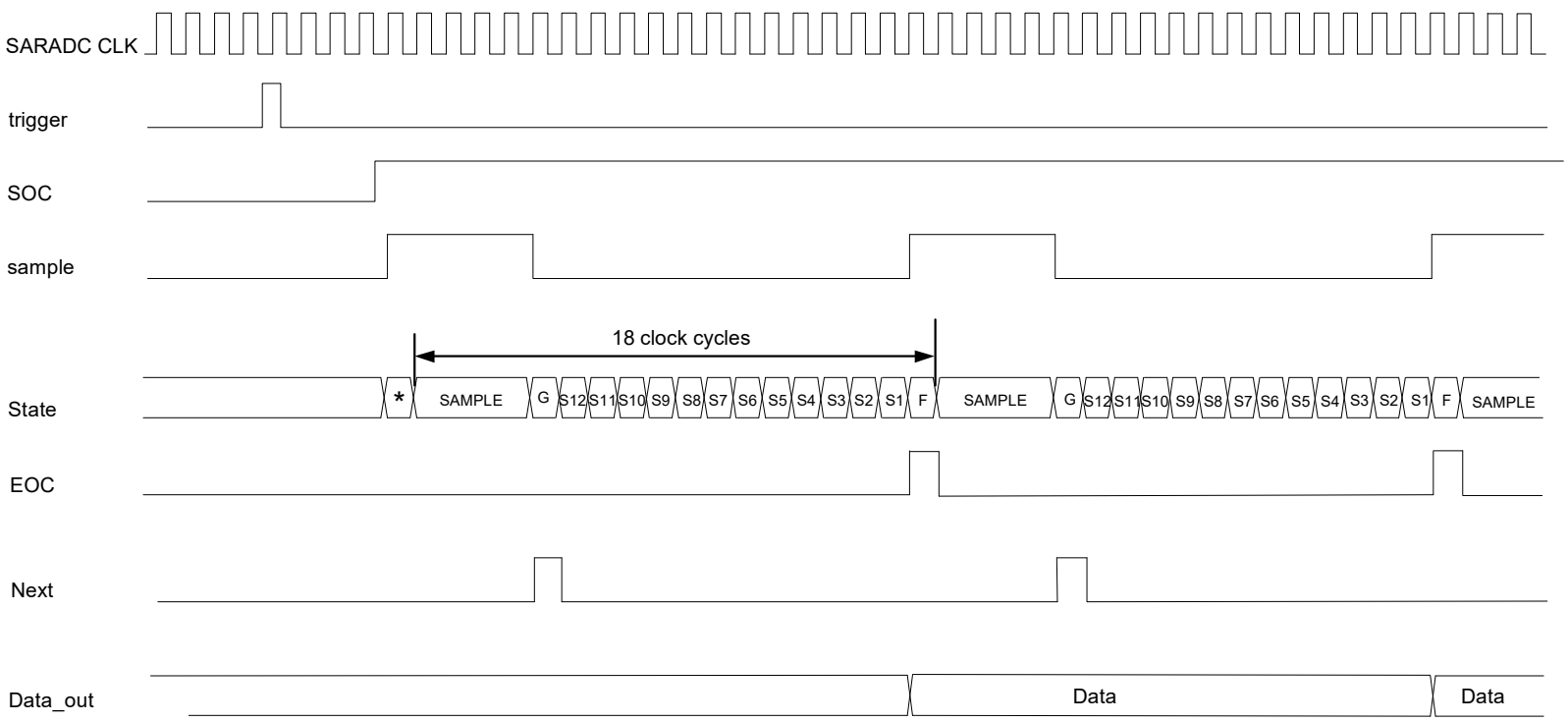
+ 12 clock cycle conversions (with 12-bit resolution)

+ 1 clock for EOF output signal

+ 1 clock for continuous conversion mode and Auto-zero

=18 clock cycles.

Figure 38-4. SAR ADC Timing



38.2.2 SARMUX

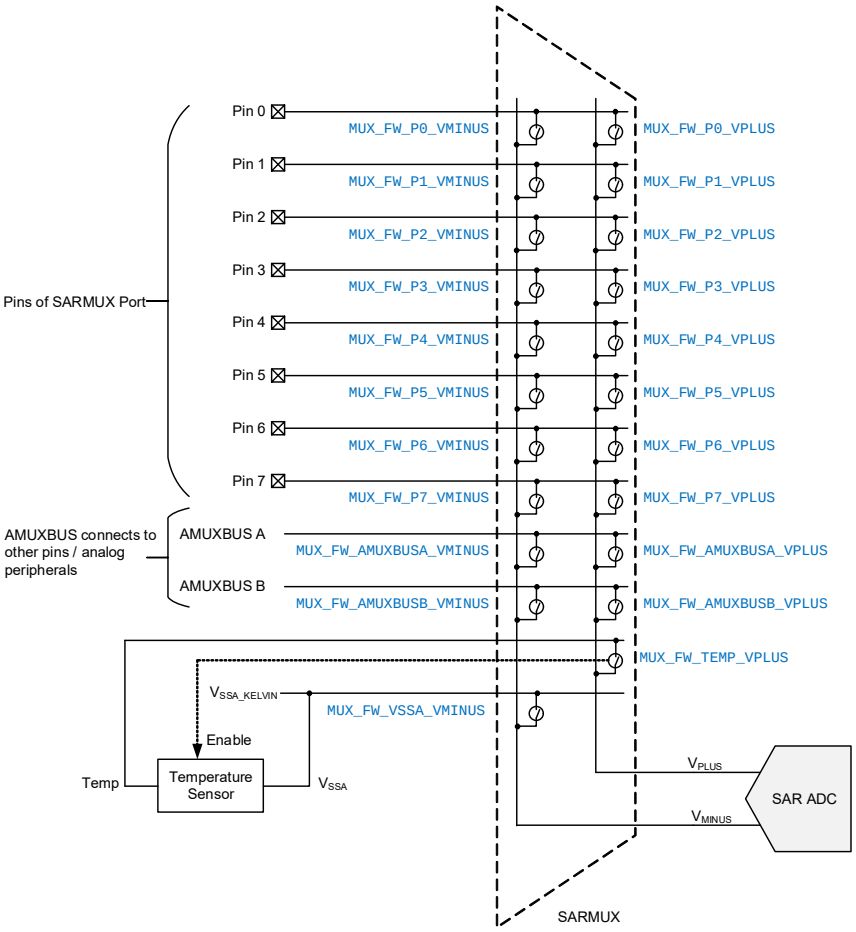
SARMUX is an analog dedicated programmable multiplexer. The main features of SARMUX are:

- Controlled by sequencer controller block (SARSEQ) or firmware
- Internal temperature sensor
- Multiple inputs:
  - Analog signals from pins (port 2)
  - Temperature sensor output (settling time for the temperature sensor is about 1 µs)
  - AMUXBUS A/AMUXBUS B

38.2.2.1 Analog Routing

SARMUX has many switches that may be controlled by SARSEQ block (sequencer controller) or firmware. Different control methods have different control capability on the switches. Figure 38-5 shows the SARMUX switches. See the PSoC 61 datasheet/PSoC 62 datasheet for the exact location of SARMUX pins.

Figure 38-5. SARMUX Switches



Sequencer control: In the sequencer control mode, the SARMUX switches are controlled by the SARSEQ block. After configuring each channel's analog routing, it enables multi-channel, automatic scan in a round-robin fashion, without CPU intervention. Not every switch in analog routing can be controlled by the sequencer, as Figure 38-5 shows.

SAR_MUX_SWITCH_SQ_CTRL register can be used to enable/disable SARSEQ control of SARMUX switches. See section 19.3.4 SARSEQ for more details of sequencer control.

Firmware control: In firmware control, registers are written by the firmware to connect required signals to V_{PLUS}/VMINUS terminals before starting the scan. Firmware can control every switch in SARMUX, as Figure 38-5 shows. However, firmware control needs continuous CPU intervention for multi-channel scans. The SAR_MUX_SWITCH0 register can be used by the firmware to control SARMUX switches. Note that additional register writes may be required to connect blocks outside the SARMUX.

The PSoC 6 MCU analog interconnection is very flexible. SAR ADC can be connected to multiple inputs via SARMUX, including both external pins and internal signals. It can also connect to non-SARMUX ports through AMUXBUS A/AMUXBUS B, at the expense of scanning performance (more parasitic coupling, longer RC time to settle).

Input from SARMUX Port

In this mode, sequencer and firmware control are possible. In addition to SARMUX switch configuration, the GPIOs must be configured properly to connect to SARMUX. See the I/O System chapter on page 261 for more details.

Input from Other Pins through AMUXBUS

Two pins that do not support SARMUX connectivity can be connected to SAR ADC as a differential pair. Additional switches are required to connect these two pins to AMUXBUS A and AMUXBUS B, and then connect AMUXBUS A and AMUXBUS B to the SAR ADC. See the I/O System chapter on page 261 for details of AMUXBUS connections.

The additional switches reduce the scanning performance (more parasitic coupling, longer RC time to settle). This is not recommended for external signals; the dedicated SARMUX port should be used, if possible. Moreover, sequencer control may not be available if more than one AMUXBUS channel is required.

Input from Temperature Sensor

On-chip temperature sensor can be used to measure the device temperature. See the Temperature Sensor chapter on page 520 for more details of this block. For temperature sensors, differential conversions are not available (conversion result is undefined), thus always use it in singled-ended mode. Temperature sensor can be routed to positive input of SAR ADC using a switch that can be controlled by the sequencer, or firmware. Setting the MUX_FW_TEMP_VPLUS bit in the SAR_MUX_SWITCH0 register can enable the temperature sensor and connect its output to V_{PLUS} of SAR ADC; clearing this bit will disable temperature sensor by cutting off its bias current.

38.2.3 SARREF

The main features of the SAR reference block (SARREF) are:

- Reference options: V_{DDA}, $V_{DDA}/2$, 1.2-V bandgap, and external reference
- Reference buffer and bypass capacitor to enhance internal reference drive capability

38.2.3.1 Reference Options

SARREF generates V_{DDA} and $V_{DDA}/2$ voltages. In addition, it can take in a 1.2-V bandgap reference from AREF (see the Analog Reference Block chapter on page 497 for details) or an external V_{REF} connected to a dedicated pin (see the PSoC 61 datasheet/PSoC 62 datasheet for details). External V_{REF} value should be between 1-V and V_{DDA}.

The external V_{REF} pin is also used to bypass the internal references. The V_{REF}_SEL bitfield in the SAR_CTRL register can be used to select which one of these references is connected to the SAR ADC.

38.2.3.2 Reference Buffer and Bypass Capacitors

The internal references, 1.2 V from bandgap and $V_{DDA}/2$, are buffered with the reference buffer. This reference may be routed to the external V_{REF} pin where a capacitor can be used to filter noise that may exist on the reference.

The VREF_BYP_CAP_EN bitfield in the SAR_CTRL register can be used to enable the bypass capacitor. REFBUF_EN and PWR_CTRL_VREF bitfields in the same register can be used to enable the buffer, and select one of seven available power levels of the reference buffer respectively.

SAR performance varies with the mode of reference and the V_{DDA} supply.

Table 38-3. Reference Modes

Reference Mode	Maximum SAR ADC Clock Frequency	Maximum Sample Rate
External Reference	18 MHz	1 Msps
Internal reference without bypass capacitor	1.8 MHz	100 ksps
Internal reference with bypass capacitor	18 MHz	1 Msps
V_{DDA} as reference	18 MHz	1 Msps

Reference buffer startup time varies with different bypass capacitor size. Table 38-4 lists two common values for the bypass capacitor and its startup time specification. If reference selection is changed between scans or when scanning after device low-power modes in which the ADC is not active (see the Device Power Modes chapter on page 225), make sure the reference buffer is settled before the SAR ADC starts sampling.

Table 38-4. Bypass Capacitor Values vs Startup Time

Capacitor Value	Startup Time
Internal reference with bypass capacitor (50 nF)	120 µs
Internal reference with bypass capacitor (100 nF)	210 µs
Internal reference without bypass capacitor	10 µs

38.2.3.3 Input Range versus Reference

All inputs should be between V_{SSA} and V_{DDA}. The ADCs input range is limited by V_{REF} selection. If negative input is Vn and the ADC reference is V_{REF}, the range on the positive input is Vn ± V_{REF}. These criteria applies for both single-ended and differential modes as long as both negative and positive inputs stay within V_{SSA} to V_{DDA}.

38.2.4 SARSEQ

SARSEQ is a dedicated control block that automatically sequences the input mux from one channel to the next while placing the result in an array of registers, one per channel. SARSEQ has the following functions:

- Controls SARMUX analog routing automatically without CPU intervention
- Controls SAR ADC core such as selecting acquisition times
- Receives data from SAR ADC and pre-process (average, range detect)
- Results are double-buffered; therefore, the CPU can safely read the results of the last scan while the next scan is in progress.

The features of SARSEQ are:

- Sixteen channels can be individually enabled as an automatic scan without CPU intervention that can scan eleven unique input channels
- Each channel has the following features:
  - Single-ended or differential mode
  - Input from external pin or internal signal (AMUXBUS/temperature sensor)
  - One of four programmable acquisition times
  - Result averaging and accumulation
- Scan triggering
  - One-shot, periodic, or continuous mode
  - Triggered by any digital signal or input from GPIO pin
  - Triggered by internal block
  - Software triggered
- Hardware averaging support
  - First order accumulate
  - 2, 4, 8, 16, 32, 64, 128, or 256 samples averaging (powers of 2)
  - Results in 16-bit representation
- Double buffering of output data
  - Left or right adjusted results
  - Results in working register and result register
- Interrupt generation
  - Finished scan conversion
  - Channel saturation detection
  - Range (configurable) detection
  - Scan results overflow
  - Collision detect

38.2.4.1 Channel Configuration

The SAR_CHAN_CONFIGx register contains the following bitfields, which control the behavior of respective channels during a SARSEQ scan:

- POS_PORT_ADDR and POS_PIN_ADDR select the connection to V_{PLUS} terminal of the ADC

- NEG_ADDR_EN, NEG_PORT_ADDR, and NEG_PIN_ADDR select the connection to V_{MINUS} terminal of the ADC
- SAMPLE_TIME_SEL selects the acquisition time for the channel
- AVG_EN enables the hardware averaging feature
- DIFFERENTIAL_EN selects single-ended/differential mode

SAR_CHAN_EN contains the enable bits that can be used to include or exclude a channel for the next SARSEQ scan.

38.2.4.2 Averaging

The SARSEQ block has a 20-bit accumulator and shift register to implement averaging. Averaging is performed after sign extension. The SAR_SAMPLE_CTRL register controls the global averaging settings.

Each channel configuration register (SAR_CHAN_CONFIG) has an enable bit (AVG_EN) to enable averaging.

The AVG_CNT bitfield in the SAR_SMAPLE_CTRL register specifies the number of accumulated samples (N) according to the formula:

N=2^(AVG_CNT+1) N range = [2..256]

For example, if AVG_CNT = 3, then N = 16.

Because the conversion result is 12-bit and the maximum value of N is 256 (left shift 8 bits), the 20-bit accumulator will never overflow.

The AVG_SHIFT bitfield in the SAR_SAMPLE_CTRL register is used to shift the accumulated result to get the averaged value. If this bit is set, the SAR sequencer performs sign extension and then accumulation. The accumulated result is then shifted right AVG_CNT + 1 bits to get the averaged result.

The AVG_MODE bitfield in the SAR_SAMPLE_CTRL can be used to select between accumulate-and-dump and interleaved averaging modes. If a channel is configured for accumulate-and-dump averaging, the SARSEQ will take N consecutive samples of the specified channel before moving to the next channel. In the interleaved mode, one sample is taken per channel and averaged over several scans.

38.2.4.3 Range Detection

The SARSEQ supports range detection to allow automatic detection of result values compared to two programmable thresholds without CPU involvement. Range detection is defined by the SAR_RANGE_THRES register. The RANGE_LOW field in the SAR_RANGE_THRES register defines the lower threshold and RANGE_HIGH field defines the upper threshold of the range.

The RANGE_COND bitfield in the SAR_RANGE_COND register define the condition that triggers a channel maskable range detect interrupt (RANGE_INTR). The following conditions can be selected:

0: result < RANGE_LOW (below the range)

1: RANGE_LOW ≤ result < RANGE_HIGH (inside the range)

2: RANGE_HIGH ≤ result (above the range)

3: result < RANGE_LOW || RANGE_HIGH ≤ result (outside range)

See Range Detection Interrupts on page 517 for details.

38.2.4.4 Double Buffer

Double buffering is used so that firmware can read the results of a complete scan while the next scan is in progress. The SAR ADC results are written to a set of working registers until the scan is complete, at which time the data is copied to a second set of registers where the data can be read by your application. This action allows sufficient time for the firmware to read the previous scan before the present scan is completed. All input channels are double buffered with 16 registers.

SAR_CHAN_WORKx registers contain the working data and SAR_CHAN_RESULTx contain the buffered results of the channels. The SAR_CHAN_WORK_UPDATED and SAR_CHAN_RESULT_UPDATED registers can be used to track if the working data and the result value of a channel is updated.

38.2.5 SAR Interrupts

SAR ADC can generate interrupts on these events:

- End of Scan – When scanning is complete for all the enabled channels.
- Overflow – When the result register is updated before the previous result is read.
- Collision – When a new trigger is received while the SAR ADC is still processing the previous trigger.
- Range Detection – When the channel result meets the threshold value.
- Saturation Detection – When the channel result is equal to the minimum or maximum value.

This section describes each interrupt in detail. These interrupts have an interrupt mask in the SAR_INTR_MASK register. By making the interrupt mask low, the corresponding interrupt source is ignored. The SAR interrupt is generated if the interrupt mask bit is high and the corresponding interrupt source is pending.

When servicing an interrupt, the interrupt service routine (ISR) can clear the interrupt source by writing a ‘1’ to the corresponding interrupt bit in the SAR_RANGE_INTR register, after reading the data.

The SAR_INTR_MASKED register is the logical AND between the interrupts sources and the interrupt mask. This register provides a convenient way for the firmware to determine the source of the interrupt.

For verification and debug purposes, a set bit (such as EOS_SET in the SAR_INTR_SET register) is used to trigger each interrupt. This action allows the firmware to generate an interrupt without the actual event occurring.

38.2.5.1 End-of-Scan Interrupt (EOS_INTR)

After completing a scan, the end-of-scan interrupt (EOS_INTR) is raised. Firmware should clear this interrupt after picking up the data from the RESULT registers.

EOS_INTR can be masked by making the EOS_MASK bit 0 in the SAR_INTR_MASK register. EOS_MASKED bit of the SAR_INTR_MASKED register is the logic AND of the interrupt flags and the interrupt masks. Writing a '1' to EOS_SET bit in SAR_INTR_SET register can set the EOS_INTR, which is intended for debug and verification.

38.2.5.2 Overflow Interrupt

If a new scan completes and the hardware tries to set the EOS_INTR and EOS_INTR is still high (firmware does not clear it fast enough), then an overflow interrupt (OVERFLOW_INTR) is generated by the hardware. This usually means that the firmware is unable to read the previous results before the current scan completes. In this case, the old data is overwritten.

OVERFLOW_INTR can be masked by making the OVERFLOW_MASK bit 0 in SAR_INTR_MASK register. OVERFLOW_MASKED bit of SAR_INTR_MASKED register is the logic AND of the interrupt flags and the interrupt masks, which are for firmware convenience. Writing a '1' to the OVERFLOW_SET bit in SAR_INTR_SET register can set OVERFLOW_INTR, which is intended for debug and verification.

38.2.5.3 Collision Interrupt

It is possible that a new trigger is generated while the SARSEQ is still busy with the scan started by the previous trigger. Therefore, the scan for the new trigger is delayed until after the ongoing scan is completed. It is important to notify the firmware that the new sample is invalid. This is done through the collision interrupt, which is raised any time a new trigger, other than the continuous trigger, is received.

The collision interrupt for the firmware trigger (FW_COLLISION_INTR) allows the firmware to identify which trigger collided with an ongoing scan.

The collision interrupts can be masked by making the corresponding bit '0' in the SAR_INTR_MASK register. The corresponding bit in the SAR_INTR_MASKED register is the logic AND of the interrupt flags and the interrupt masks. Writing a '1' to the corresponding bit in SAR_INTR_SET register can set the collision interrupt, which is intended for debug and verification.

38.2.5.4 Range Detection Interrupts

Range detection interrupt flag can be set after averaging, alignment, and sign extension (if applicable). This means it is not required to wait for the entire scan to complete to determine whether a channel conversion is over-range. The threshold values need to have the same data format as the result data.

Range detection interrupt for a specified channel can be masked by setting the SAR_RANGE_INTR_MASK register specified bit to '0'. Register SAR_RANGE_INTR_MASKED reflects a bitwise AND between the interrupt request and mask registers. If the value is not zero, then the SAR interrupt signal to the NVIC is high.

SAR_RANGE_INTR_SET can be used for debug/verification. Write a '1' to set the corresponding bit in the interrupt request register; when read, this register reflects the interrupt request register.

There is a range detect interrupt for each channel (RANGE_INTR and INJ_RANGE_INTR).

38.2.5.5 Saturate Detection Interrupts

The saturation detection is always applied to every conversion. This feature detects if a sample value is equal to the minimum or maximum value and sets a maskable interrupt flag for the corresponding channel. This action allows the firmware to take action, such as discarding the result, when the SAR ADC saturates. The sample value is tested right after conversion, before averaging. This means that the interrupt is set while the averaged result in the data register is not equal to the minimum or maximum.

Saturation interrupt flag is set immediately to enable a fast response to saturation, before the full scan and averaging. Saturation detection interrupt for specified channel can be masked by setting the SAR_SATURATE_INTR_MASK register specified bit to '0'. SAR_SATURATE_INTR_MASKED register reflects a bit-wise AND between the interrupt request and mask registers. If the value is not zero, then the SAR interrupt signal to the NVIC is high.

SAR_SARTURATE_INTR_SET can be used for debug/verification. Write a '1' to set the corresponding bit in the interrupt request register; when read, this register reflects the interrupt request register.

38.2.5.6 Interrupt Cause Overview

INTR_CAUSE register contains an overview of all the pending SAR interrupts. It allows the ISR to determine the cause of the interrupt. The register consists of a mirror copy of SAR_INTR_MASKED. In addition, it has two bits that aggregate the range and saturate detection interrupts of all channels. It includes a logical OR of all the bits in

RANGE_INTR_MASKED and SATURATE_INTR_MASKED registers (does not include INJ_RANGE_INTR and INJ_-SATURATE_INTR).

38.2.6 Trigger

A scan can be triggered in the following ways:

- A firmware or one-shot trigger is generated when the firmware writes to the FW_TRIGGER bit of the SAR_START_CTRL register. After the scan is completed, the SARSEQ clears the FW_TRIGGER bit and goes back to idle mode waiting for the next trigger. The FW_TRIGGER bit is cleared immediately after the SAR is disabled.
- Trigger from other peripherals through the trigger multiplexer (see the Trigger Multiplexer Block chapter on page 294 for more details).
- A continuous trigger is activated by setting the CONTINUOUS bit in SAR_SAMPLE_CTRL register. In this mode, after completing a scan the SARSEQ starts the next scan immediately; therefore, the SARSEQ is always BUSY. As a result, all other triggers are essentially ignored. Note that FW_TRIGGER will still get cleared by hardware on the next completion.

For firmware continuous trigger, it takes only one SAR ADC clock cycle before the sequencer tells the SAR ADC to start sampling (provided the sequencer is idle).

38.2.7 SAR ADC Status

The current SAR status can be observed through the BUSY and CUR_CHAN fields in the SAR_STATUS register. The BUSY bit is high whenever the SAR is busy sampling or converting a channel; the CUR_CHAN bits indicate the number of the current channel being sampled. SW_VREF_NEG bit indicates the current switch status, including register controls, of the switch in the SAR ADC that shorts NEG with V_{REF} input.

The CUR_AVG_ACCU and CUR_AVG_CNT fields in the SAR_AVG_STAT register indicate the current averaging accumulator contents and the current sample counter value for averaging (counts down).

The SAR_MUX_SWITCH_STATUS register gives the current switch status of MUX_SWITCH0 register. These status registers help to debug SAR behavior.

38.3 Registers

Name	Description
SAR_CTRL	Global configuration register. Analog control register
SAR_SAMPLE_CTRL	Global configuration register. Sample control register
SAR_SAMPLE_TIME01	Global configuration register. Sample time specification ST0 and ST1
SAR_SAMPLE_TIME23	Global configuration register. Sample time specification ST2 and ST3
SAR_RANGE_THRES	Global range detect threshold register
SAR_RANGE_COND	Global range detect mode register
SAR_CHAN_EN	Enable bits for the channels
SAR_START_CTRL	Start control register (firmware trigger)
SAR_CHAN_CONFIGx	Channel configuration register. There are 16 such registers with x = 0 to 15
SAR_CHAN_WORKx	Channel working data register. There are 16 such registers with x = 0 to 15
SAR_CHAN_RESULTx	Channel result data register. There are 16 such registers with x = 0 to 15
SAR_CHAN_WORK_UPDATED	Channel working data register: updated bits
SAR_CHAN_RESULT_UPDATED	Channel result data register: updated bits
SAR_STATUS	Current status of internal SAR registers (for debug)
SAR_AVG_STAT	Current averaging status (for debug)
SAR_INTR	Interrupt request register
SAR_INTR_SET	Interrupt set request register
SAR_INTR_MASK	Interrupt mask register
SAR_INTR_MASKED	Interrupt masked request register
SAR_SATURATE_INTR	Saturate interrupt request register
SAR_SATURATE_INTR_SET	Saturate interrupt set request register
SAR_SATURATE_INTR_MASK	Saturate interrupt mask register
SAR_SATURATE_INTR_MASKED	Saturate interrupt masked request register
SAR_RANGE_INTR	Range detect interrupt request register
SAR_RANGE_INTR_SET	Range detect interrupt set request register
SAR_RANGE_INTR_MASK	Range detect interrupt mask register
SAR_RANGE_INTR_MASKED	Range interrupt masked request register
SAR_INTR_CAUSE	Interrupt cause register
SAR_MUX_SWITCH0	SARMUX firmware switch controls
SAR_MUX_SWITCH_CLEAR0	SARMUX firmware switch control clear
SAR_MUX_SWITCH_SQ_CTRL	SARMUX switch sequencer control
SAR_MUX_SWITCH_STATUS	SARMUX switch status

39. Temperature Sensor

This PSoC 6 MCU technical reference manual (TRM) provides comprehensive and detailed information about the functions of the PSoC 6 MCU device hardware. It is divided into two books: architecture TRM and registers TRM. The TRM is not recommended for those new to the PSoC 6 MCU, nor as a guide for developing PSoC 6 MCU applications. Use these documents instead:

- PSoC 61 datasheet, PSoC 62 datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

PSoC 6 MCUs have an on-chip temperature sensor that is used to measure the internal die temperature. The sensor consists of a transistor connected in diode configuration.

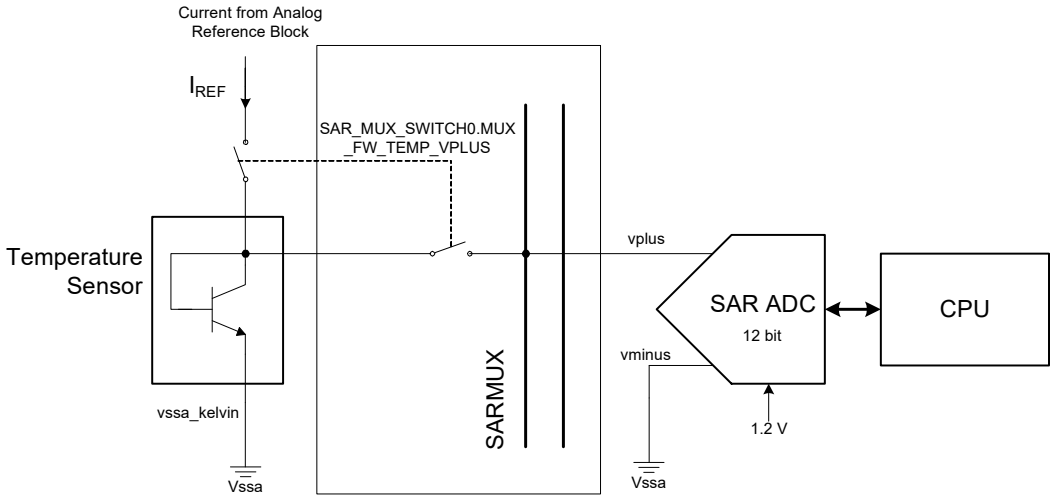
39.1 Features

- Measures device temperature
- Voltage output can be internally connected to SAR ADC for digital readout
- Factory calibrated parameters

39.2 Architecture

The temperature sensor consists of a single bipolar junction transistor (BJT) in the form of a diode. The transistor is biased using a reference current I_{REF} from the analog reference block (see the Analog Reference Block chapter on page 497 for more details). Its base-to-emitter voltage (V_{BE}) has a strong dependence on temperature at a constant collector current and zero collector-base voltage. This property is used to calculate the die temperature by measuring the V_{BE} of the transistor using the SAR ADC, as shown in Figure 39-1.

Figure 39-1. Temperature Sensing Mechanism



The analog output from the sensor (V_{BE}) is measured using the SAR ADC. Die temperature in °C can be calculated from the ADC results as given in the following equation:

$$Temp = (A \times SAR_{out} + 2^{10} \times B) + T_{adjust}$$

Equation 39-1

- Temp is the slope compensated temperature in °C represented as Q16.16 fixed point number format.
- 'A' is the 16-bit multiplier constant. The value of A is determined using the PSoC 6 MCU characterization data of two point slope calculation. It is calculated as given in the following equation.

$$A = (\text{signed int})\left(2^{16}\left(\frac{100°C - (-40°C)}{SAR_{100°C} - SAR_{-40°C}}\right)\right)$$

Equation 39-2

  Where,

  SAR_{100C} = ADC counts at 100 °C

  SAR_{-40C} = ADC counts at –40 °C

  Constant 'A' is stored in a register SFLASH_SAR_TEMP_MULTIPLIER. See the registers TRM for more details.
- 'B' is the 16-bit offset value. The value of B is determined on a per die basis by taking care of all the process variations and the actual bias current (I_{REF}) present in the chip. It is calculated as given in the following equation.

$$B = (\text{unsigned int})\left(2^{6}\text{x}100°C - \left(\frac{A \times SAR_{100C}}{2^{10}}\right)\right)$$

Equation 39-3

  Where,

  SAR_{100C} = ADC counts at 100 °C

  Constant 'B' is stored in a register SFLASH_SAR_TEMP_OFFSET. See the registers TRM for more details.
- T_{adjust} is the slope correction factor in °C. The temperature sensor is corrected for dual slopes using the slope correction factor. It is evaluated based on the result obtained without slope correction, which is given by the following equation:

$$T_{initial} = (A \times SAR_{OUT} + 2^{10} \times B)$$

Equation 39-4

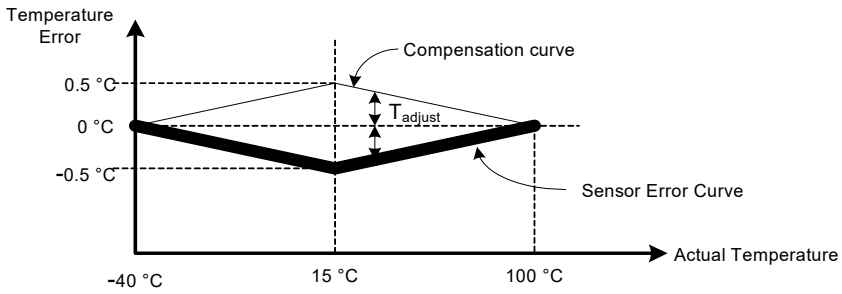
If $T_{initial}$ is greater than the center value (15 °C), then T_{adjust} is given by the following equation.

$$T_{adjust} = \left(\frac{0.5°C}{100°C - 15°C} \times (100°C \times 2^{16} - T_{initial})\right)$$ **Equation 39-5**

If $T_{initial}$ is less than center value, then T_{adjust} is given by the following equation.

$$T_{adjust} = \left(\frac{0.5°C}{40°C + 15°C} \times (40°C \times 2^{16} + T_{initial})\right)$$ **Equation 39-6**

Figure 39-2. Temperature Error Compensation



Note: A and B are 16-bit constants stored in flash during factory calibration. These constants are valid only with a specific SAR ADC configuration. See 39.3 SAR ADC Configuration for Measurement for details.

39.3 SAR ADC Configuration for Measurement

The temperature sensor is routed to the SAR ADC for measurement. Configure the SAR ADC channel to connect to the temperature sensor by writing '1' to the MUX_FW_TEMP_VPLUS bit of the SAR_MUX_SWITCH0 register. This also routes the reference current to the sensor. Note that if the SAR sequencer is used, the SAR_MUX_SWITCH_SQ_CTRL register also needs to be written. Configure the selected channel to a single-ended mode with negative input connected to V_{SS}.

Use the following SAR ADC parameters:

1. 12-bit resolution
2. Internal 1.2-V V_{REF}
3. Right-aligned result
4. 100 ksps sample rate (for best performance)
5. Averaging can be enabled to reduce the noise. However, averaging count, averaging shift, and averaging mode should be set so as to get a 12-bit result after averaging.

For details about the SAR ADC parameters and its configuration, see the SAR ADC chapter on page 506.

39.4 Algorithm

1. Get the digital output from the SAR ADC.
2. Fetch 'A' from SFLASH_SAR_TEMP_MULTIPLIER and 'B' from SFLASH_SAR_TEMP_OFFSET.
3. Calculate the die temperature using the following equation:

$$Temp = (A \times SAR_{OUT} + 2^{10} \times B) + T_{ADJUST}$$

   For example, let A = 0xBC4B and B = 0x65B4. Assume that the output of SAR ADC (V_{BE}) is 0x595 at a given temperature. Firmware does the following calculations:

   a. Multiply A and V_{BE}: 0xBC4B × 0x595 = $(-17333)_{10} \times (1429)_{10} = (-24768857)_{10}$
   b. Multiply B and 1024: 0x65B4 × 0x400 = $(26036)_{10} \times (1024)_{10} = (26660864)_{10}$
   c. Add the result of steps 1 and 2 to get $T_{initial}$: $(-24768857)_{10} + (26660864)_{10} = (1892007)_{10}$ = 0x1CDEA7

d. Calculate T_{adjust} using $T_{initial}$ value: $T_{initial}$ is the upper 16 bits multiplied by 2^{16}, that is, 0x1C00 = $(1835008)_{10}$. It is greater than 15°C (0x1C - upper 16 bits). Use Equation 4 to calculate T_{adjust}. It comes to 0x6B1D = $(27421)_{10}$
e. Add T_{adjust} to $T_{initial}$: $(1892007)_{10}$ + $(27421)_{10}$ = $(1919428)_{10}$ = 0x1D49C4
f. The integer part of temperature is the upper 16 bits = 0x001D = $(29)_{10}$
g. The decimal part of temperature is the lower 16 bits = 0x4B13 = $(0.18884)_{10}$
h. Combining the result of steps f and g, Temp = 29.18884 °C ~ 29.2°C

39.5 Registers

Name	Description
SFLASH_SAR_TEMP_MULTIPLIER	Multiplier constant 'A' as defined in Equation 39-1.
SFLASH_SAR_TEMP_OFFSET	Constant 'B' as defined in Equation 39-1.

40. CapSense

The CapSense system can measure the self-capacitance of an electrode or the mutual capacitance between a pair of electrodes. In addition to capacitive sensing, the CapSense system can function as an ADC to measure voltage on any GPIO pin that supports the CapSense functionality.

The CapSense touch sensing method in PSoC 6 MCUs, which senses self-capacitance, is known as CapSense Sigma Delta (CSD). Similarly, the mutual-capacitance sensing method is known as CapSense Cross-point (CSX). The CSD and CSX touch sensing methods provide the industry's best-in-class signal-to-noise ratio (SNR), high touch sensitivity, low-power operation, and superior EMI performance.

CapSense touch sensing is a combination of hardware and firmware techniques. Therefore, use the CapSense component provided by the ModusToolbox IDE to implement CapSense designs. See the *PSoC 4 and PSoC 6 MCU CapSense Design Guide* for more details.