PSoC™ 4000T MCU architecture

Reference manual

About this document

Scope and purpose

This document provides the reader with detailed descriptions about the features of the PSoC™ 4000T MCU device, its functional units and their interaction.

Intended audience

This reference manual is addressed to embedded hardware and software developers.

Table of contents

Table of contents

Section A: Overview

This section encompasses the following chapters:

Revision history

Revision	Issue date	Description of change
**	2022-01-19	Initial version of PSoC™ 4000T MCU architecture TRM.
*A	2022-05-12	Chapter 1. Introduction: Updated section 1.2 Features. Updated section 1.5.1 CAPSENSE™. Chapter 5 Interrupts Updated Table 5-2: Updated MSCv3 to MSC and MSCv3LP to MSCLP. Chapter 7 I/O system Updated section 7.6: Added a paragraph on Port Interrupt Cause Register. Updated section 7.8 Registers: Removed references to Smart I/O. Chapter 8: Clocking system Updated the number of 16 bit clock divider, 16.5 bit divider, 24.5 bit divider and peripheral clocks. Added external clock source to the list of internal clock sources. Updated figure 8-1 Clocking system block diagram. Updated section 8.3.4: Updated the examples demonstrating the bit field values for the peripheral clock divider configuration. Updated sections 8.4, 8.5 and 8.6: Updated the "ENABLE" to "EN". Updated Table 8-8: Updated the "SEL_DIV" bit from 5:0 to 0. Chapter 9: Power supply and monitoring Update section 9.3.1.1: Updated VDDD voltage range "1.8 V to 5 V" to "2 V to 5 V". Chapter 10 Power modes Updated Table 10-1 and Table 10-2. Updated section 10.3 Deep-Sleep mode. Chapter 12 Watchdog timer Updated section 12.3: Reset asserted by WDT – "hardware" to "system", WDT program period "2048 ms" to "1638.4 ms". Updated Table 12-1: Added "SRSS_INTR_MASK" register and its description. Chapter 13 Trigger multiplexer block Updated description of PERI_TR_CTL register in Table 13-1. Chapter 19 Nonvolatile memory programming Updated section 19.5.1: Updated Silicon ID range "3600-36FF" to "0x3600 to 0x36FF".
*B	2023-07-12	Fixed links in Serial communications block (SCB) chapter. Fixed links in section 2.2.1. Updated CAPSENSE™ chapter. Updated System resource subsystem block diagram.

1 Introduction

PSoC™ 4 is a family of scalable MCU with an Arm® Cortex®-M0+ CPU. It combines a high-performance capacitive sensing subsystem, programmable and reconfigurable analog and digital blocks. The new PSoC™ 4000T series provides upgrade path for PSoC™ 4000 and PSoC™ 4000S based designs to fifth-generation HMI technology with software and package compatibility.

PSoC™ 4000T is a member of the PSoC™ 4 MCU family with fifth-generation CAPSENSE™ and MULTI-SENSE technology offering ultra-low power touch HMI solution based on an integrated “Always-On” sensing technology, improved performance to enable modern sleek user interface solutions with superior liquid tolerance and provides robust and reliable touch HMI solution for harsh environments.

PSoC™ 4000T is a microcontroller with standard communication, timing peripherals and Infineon’ fifth-generation CAPSENSE™ and MULTI-SENSE HMI technology purpose built for varieties of low power applications including wearable, hearable and smart connected IoT products that needs low power operation and improved performance to enable next generation of user experience.

1.1 Top level architecture

Figure 1-1 shows the major components of the PSoC™ 4000T MCU architecture.

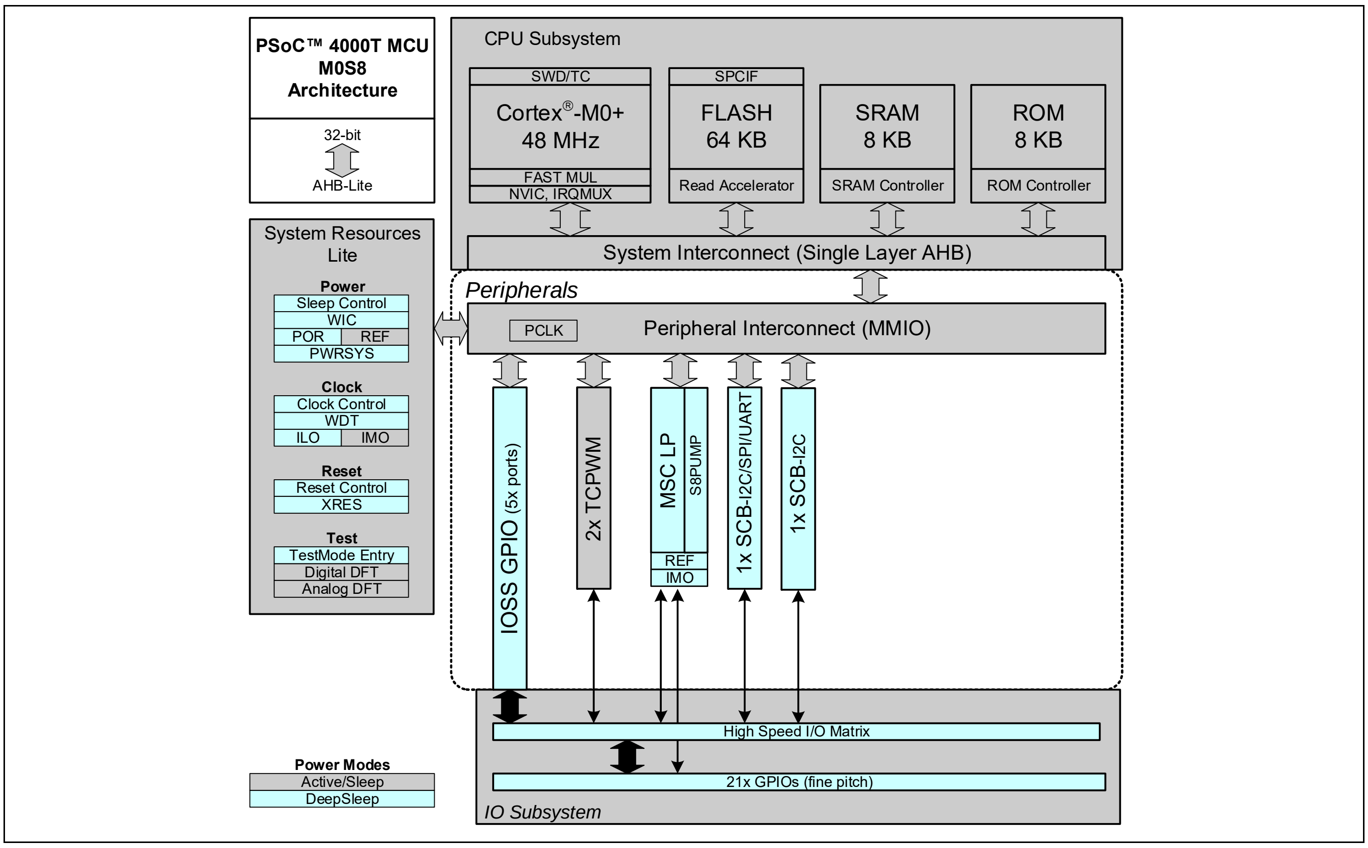


Figure 1-1. PSoC™ 4000T MCU block diagram

1.2 Features

- 32-bit MCU subsystem
  - 48-MHz Arm® Cortex®-M0+ CPU with single-cycle multiply
  - Up to 64 KB of flash with read accelerator
  - Up to 8 KB of SRAM
- Low-power 1.71 V to 5.5 V operation
  - Deep sleep mode with 5 µA always-on touch sensing
  - Active touch detection and tracking with 200 µA (average)
- Fifth-generation CAPSENSE™ sensing
  - All-new ratio-metric sensing architecture in multi-sense converter (MSC) provides best-in-class signal-to-noise ratio (SNR) (>5:1) and liquid tolerance for capacitive sensing.
  - "Always-On" sensing in operation deep sleep mode with hardware-based wake on touch detection for ultra-low power operation in standby mode.
  - Autonomous channel scanning in without assistance from MCU core for low power optimization with active touch detection and tracking
  - Advanced proximity sensing with directivity with machine learning based algorithms
  - Infineon-supplied software middleware makes capacitive sensing design easy
  - Automatic hardware tuning (SmartSense)
- Serial communication
  - Two independent runtime reconfigurable serial communication blocks (SCBs) with re-configurable I2 C, SPI,
    or UART functionality in one block with master/slave I2 C functionality in the other.
- Timing and pulse-width modulation
  - Two 16-bit timer/counter/pulse-width modulator (TCPWM) blocks
  - Center-aligned, edge, and pseudo-random modes
  - Comparator-based triggering of kill signals
  - Quadrature decoder
- Clock sources
  - ±2% Internal main oscillator (IMO)
  - 40 kHz Internal low-power oscillator (ILO)
- Up to 21 programmable GPIO pins
  - 25-pin WLCSP (0.35 mm pitch), 24-pin QFN (0.5 mm pitch), and a 16-pin QFN package (0.5 mm pitch).
  - GPIO pins can have sensing or digital functionality
- ModusToolbox™ software
  - Comprehensive collection of multi-platform tools and software libraries
  - Includes board support packages (BSPs), peripheral driver library (PDL), and middleware such as CAPSENSE™
- Industry-standard tool compatibility
  - After configuration, development can be done with Arm®-based industry-standard development tools

1.3 CPU system

1.3.1 Processor

The heart of the PSoC™ is a 32-bit Cortex®-M0+ CPU core running up to 48 MHz. It is optimized for low-power operation with extensive clock gating. It uses 16-bit instructions and executes a subset of the Thumb-2 instruction set. This instruction set enables fully compatible binary upward migration of the code to higher performance processors such as Cortex®-M3 and -M4.

The CPU has a hardware multiplier that provides a 32-bit result in one cycle.

1.3.2 Interrupt controller

The CPU subsystem includes a nested vectored interrupt controller (NVIC) with 13 interrupt inputs and a wakeup interrupt controller (WIC), which can wake the processor from Deep-Sleep mode.

1.3.3 Memory

1.3.3.1 Flash

The PSoC™ 4 memory subsystem has a flash module with a flash accelerator tightly coupled to the CPU, to improve average access times from the flash block. The flash accelerator delivers 85 percent of single-cycle SRAM access performance on an average.

1.3.3.2 SRAM

The PSoC™ 4 memory subsystem provides SRAM, which is retained in all power modes of the device.

1.4 System-wide resources

1.4.1 Clocking system

The clocking system consists of the internal main oscillator (IMO) and internal low-speed oscillator (ILO) as internal clocks and has provision for an external clock source.

The IMO with an accuracy of ±2 percent is the primary source of internal clocking in the device. The default IMO frequency is 24 MHz and can be adjusted between 24 MHz and 48 MHz in steps of 4 MHz. Multiple clock derivatives are generated from the main clock frequency to meet various application needs.

The ILO is a low-power, less accurate oscillator and is used as a source for LFCLK, to generate clocks for peripheral operation in Deep-Sleep mode. Its clock frequency is 40 kHz with ±60 percent accuracy.

An external clock source ranging from 1 MHz to 48 MHz can be used to generate the clock derivatives for the functional blocks instead of the IMO.

1.4.2 Power system

The device operates with a single external supply in the range 1.71 V to 5.5 V. It provides multiple power supply domains – V_{DDD} to power the digital section, and V_{DDA} for noise isolation of the analog section. V_{DDD} and V_{DDA} should be shorted externally.

The device has two low-power modes – Sleep and Deep-Sleep – in addition to the default Active mode. In Active mode, the CPU runs with all the logic powered. In Sleep mode, the CPU is powered off and SRAM is in retention, with all other peripherals functional. In Deep-Sleep mode, the CPU, SRAM, and high-speed logic are in retention; the main system clock is OFF while the low-frequency clock is ON and the low-frequency peripherals are in operation.

Multiple internal regulators are available in the system to support power supply schemes in different power modes.

1.4.3 GPIO

Every GPIO has the following characteristics:

- Eight drive strength modes
- Individual control of input and output disables
- Hold mode for latching previous state
- Selectable slew rates
- Interrupt generation – edge triggered

The pins are organized in a port that are 8-bit wide. A high-speed I/O matrix is used to multiplex between various signals that may connect to an I/O pin. Pin locations for fixed-function peripherals are also fixed.

1.4.4 Watchdog timers

The PSoC™ 4 device has one 16-bit watchdog timer, which is capable of automatically resetting the device in the event of an unexpected firmware execution path or a brownout that compromises the CPU functionality.

1.4.5 Timer/counter/PWM block

The Timer/Counter/PWM block consists of up to two16-bit counters with user-programmable period length. The TCPWM block has a capture register, period register, and compare register. The block supports complementary, dead-band programmable outputs. It also has a kill input to force outputs to a predetermined state. Other features of the block include center-aligned PWM, clock prescaling, pseudo random PWM, and quadrature decoding.

1.4.6 Serial communication blocks

The device has two SCBs.

The features of SCB[0] include:

- Standard I^2C multi-master and slave function
- Standard SPI master and slave function with Motorola, Texas Instruments, and National (MicroWire) modes
- Standard UART transmitter and receiver function with SmartCard reader (ISO7816), IrDA protocol, and LIN
- Standard LIN slave with LIN v1.3 and LIN v2.1/2.2 specification compliance
- EZ function mode support with 32-byte buffer
- Standard I2C multi-master and slave function

The features of SCB[1] include standard I2C multi-master and slave function only.

1.5 Special function peripherals

1.5.1 CAPSENSE™

PSoC™ 4000T device features Fifth-generation CAPSENSE™ and MULTI-SENSE technology. The key features of Fifth-generation CAPSENSE™ sensing are as follows:

- All-new ratio-metric sensing architecture in multi-sense converter (MSCLP) provides best-in-class signal-to-noise ratio (SNR) (>5:1) and liquid tolerance for capacitive sensing.
- "Always-On" sensing in operation Deep Sleep mode with hardware-based wake-on-touch (WoT) detection for ultra-low-power operation in Standby mode.
- Autonomous channel scanning in without assistance from MCU core for low power optimization with active touch detection and tracking

MSCLP block has built-in voltage reference and clock sources (called MSC IMO and MSC ILO) for WoT operation. The MSCLP does not depend on the SRSS and is fully operational even when the SRSS is powered OFF and device is in Deep Sleep.

1.6 Program and debug

PSoC™ 4 devices support programming and debugging features of the device via the on-chip SWD interface. The ModusToolbox™ IDE provides fully integrated programming and debugging support. The SWD interface is also fully compatible with industry standard third-party tools.

1.7 Device feature summary

Table 1-1 shows the PSoC™ 4000T MCU devices summary.

Table 1-1. PSoC™ 4000T device summary

Features	**PSoC™ 4000T MCU**
Maximum CPU Frequency	48 MHz
Flash	64 KB
SRAM	8 KB
GPIOs (max)	21
CAPSENSE™	1 channel
Timer, Counter, PWM (TCPWM)	2
Serial Communication Block (SCB)	2
Power Modes	Active, Sleep, and Deep-Sleep

2 Getting started

2.1 Support

Free support for PSoC™ 4 products is available online at **www.infineon.com/psoc4**. Resources include training seminars, discussion forums, application notes, PSoC™ consultants, CRM technical support email, knowledge base, and application support engineers.

For application assistance, visit **www.infineon.com/support** or call 1-800-541-4736.

2.2 Development ecosystem

2.2.1 PSoC™ 4 MCU resources

Infineon® provides a wealth of data at **www.infineon.com** to help you select the right PSoC™ device and quickly and effectively integrate it into your design. The following is an abbreviated, hyperlinked list of resources for PSoC™ 4 MCU:

- **Overview: PSoC™ portfolio**
- Product Selectors: **PSoC™ 4 MCU**
- **Application notes** cover a broad range of topics, from basic to advanced level, and include the following:
  - **AN79953**: Getting started With PSoC™ 4
  - **AN64846**: Getting started with CAPSENSE™
  - **AN92239**: Proximity sensing with CAPSENSE™
  - **AN85951**: PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ design guide
  - **AN88619**: PSoC™ 4 hardware design considerations
  - **AN219207** - Inductive sensing design guide
  - **AN86233**: PSoC™ 4 MCU power reduction techniques
  - **AN73854**: PSoC™ - Introduction to bootloaders
  - **AN57821**: Mixed signal circuit board layout
  - **AN89610**: Arm® Cortex® code optimization
  - **AN91184**: PSoC™ 4-BLE - Designing Bluetooth® Low Energy applications
- **Code examples** demonstrate product features and usage, and are also available on **GitHub repositories**.
- **Datasheets** describe and provide electrical specifications for each family.
- **PSoC™ 4 MCU programming specification** provides the information necessary to program PSoC™ 4 MCU nonvolatile memory.
- Development Tools
  - **ModusToolbox™ software** enables cross platform code development with a robust suite of tools and software libraries.
  - CY8CKIT-040T Evaluation kit is a low-power hardware platform that enables design and debug of the PSoC™ 4000T MCU device.
  - **MiniProg4** and **MiniProg3** all-in-one development programmers and debuggers.
  - **PSoC™ 4 MCU CAD libraries** provide footprint and schematic support for common tools. **IBIS models** are also available.
  - **Training videos** are available on a wide range of topics including the **PSoC™ 4 MCU 101 series**.
  - **Infineon developer community** enables connection with fellow PSoC™ developers around the world, 24 hours a day, 7 days a week, and hosts a dedicated **PSoC™ 4 MCU community**.

2.2.2 ModusToolbox™ software

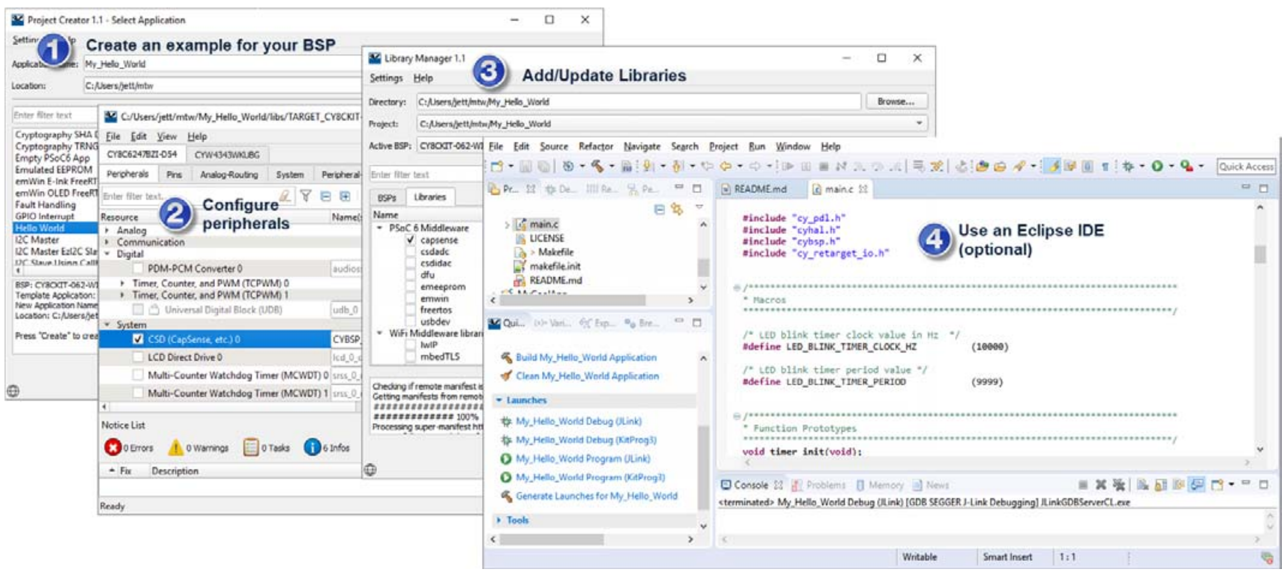
ModusToolbox software is Infineon®' comprehensive collection of multi-platform tools and software libraries that enable an immersive development experience for creating converged MCU and wireless systems. It is:

- Comprehensive - it has the resources you need
- Flexible - you can use the resources in your own workflow
- Atomic - you can get just the resources you want

Infineon® provides a large collection of code **repositories on GitHub**, including:

- Board Support Packages (BSPs) aligned with Infineon® kits
- Low-level resources, including a peripheral driver library (PDL)
- Middleware enabling industry-leading features such as CAPSENSE™
- An extensive set of thoroughly tested **code example applications**

ModusToolbox™ Software is IDE-neutral and easily adaptable to your workflow and preferred development environment. It includes a project creator, peripheral and library configurators, a library manager, as well as the optional Eclipse IDE for ModusToolbox™, as **Figure 2-1** shows. For information on using Infineon® tools, refer to the documentation delivered with ModusToolbox™ Software, and **AN79953: Getting started with PSoC™ 4**.


1 Create an example for your BSP
2 Configure peripherals
3 Add/Update Libraries
4 Use an Eclipse IDE (optional)

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "cy_retarget_io.h"

/*******************************************************************************
 * Macros
 *******************************************************************************/

/* LED blink timer clock value in Hz  */
#define LED_BLINK_TIMER_CLOCK_HZ          (10000)

/* LED blink timer period value */
#define LED_BLINK_TIMER_PERIOD            (9999)

/*******************************************************************************
 * Function Prototypes
 *******************************************************************************/
void timer init(void);
```

```
<terminated> My_Hello_World Debug (JLink) [GDB SEGGER J-Link Debugging] JLinkGDBServerCL.exe
```

Figure 2-1. ModusToolbox™ software tools

3 Document construction

This document includes the following sections:

- **"CPU system"** on page 22
- **"System resources subsystem (SRSS)"** on page 43
- **"Digital system"** on page 78
- **"Analog system"** on page 180
- **"Program and debug"** on page 182

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.
- 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.

3.3 Register conventions

Register conventions are detailed in the PSoC™ 4000T MCU registers technical reference manual.

3.4 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.

3.5 Units of measure

Table 3-1. Units of measure

Abbreviation	Unit of measure
bps	bits per second
°C	degrees Celsius
dB	decibels
fF	femtofarads
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
s	sigma: one standard deviation
V	volts

3.6 Acronyms

Table 3-2. Acronyms

Acronym	Definition
ABUS	analog output bus
AC	alternating current
ADC	analog-to-digital converter
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
BOD	brownout detect
BOM	bill of materials
BR	bit rate
BRA	bus request acknowledge
BRQ	bus request
CBC	cipher block chaining
CFB	cipher feedback
CI	carry in
CMP	compare
CO	carry out
COM	LCD common signal
CPU	central processing unit
CRC	cyclic redundancy check
CT	continuous time
CTB	continuous time block
CTBm	continuous time block mini
CTR	counter
DAC	digital-to-analog converter
DAP	debug access port
DAS	digitized analog signal
DC	direct current
DI	digital or data input
DMA	direct memory access
DMIPS	Dhrystone million instructions per second
DO	digital or data output
DSI	digital signal interface
DSM	deep-sleep mode
DW	data wire

Table 3-2. Acronyms *(continued)*

Acronym	Definition
ECB	electronic code book
ECO	external crystal oscillator
EEPROM	electrically erasable programmable read only memory
EMIF	external memory interface
FB	feedback
FIFO	first in first out
FSR	full scale range
GPIO	general purpose I/O
HCI	host-controller interface
HFCLK	high-frequency clock
HSIOM	high-speed I/O matrix
I^2C	inter-integrated circuit
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
IRES	initial power on reset
IRA	interrupt request acknowledge
IRQ	interrupt request
ISR	interrupt service routine
IVR	interrupt vector read
LCD	liquid crystal display
LFCLK	low-frequency clock
LFSR	linear feedback shift registers
LFT	loof for touch
LPCOMP	low-power comparator
LRb	last received bit
LRB	last received byte
LSb	least significant bit
LSB	least significant byte
LUT	lookup table
MISO	master-in-slave-out
MMIO	memory mapped input/output
MOSI	master-out-slave-in

Table 3-2. Acronyms *(continued)*

Acronym	Definition
MPU	memory protection unit
MSb	most significant bit
MSB	most significant byte
MSC	multi sense converter
MSP	main stack pointer
NMI	non-maskable interrupt
NVIC	nested vectored interrupt controller
OFB	output feedback
PC	program counter
PCB	printed circuit board
PCH	program counter high
PCL	program counter low
PD	power down
PGA	programmable gain amplifier
PM	power management
PMA	PSoC memory arbiter
POR	power-on reset
PPOR	precision power-on reset
PRNG	pseudo random number generator
PRS	pseudo random sequence
PSoC™	Programmable System-on-Chip
PSP	process stack pointer
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
ROM	read only memory
RMS	root mean square
RTC	real-time clock
RW	read/write
SAR	successive approximation register
SEG	LCD segment signal
SC	switched capacitor
SCB	serial communication block
SIE	serial interface engine
SIO	special I/O

Table 3-2. Acronyms *(continued)*

Acronym	Definition
SE0	single-ended zero
SHA	secure hash algorithm
SNR	signal-to-noise ratio
SOF	start of frame
SOI	start of instruction
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
SSADC	single slope ADC
SSC	supervisory system call
SYSCLK	system clock
SWD	single wire debug
TC	terminal count
TCPWM	timer, counter, PWM
TD	transaction descriptors
TIA	trans-impedance amplifier
TRNG	true random number generator
TX	transmitter
UART	universal asynchronous receiver/transmitter
UDB	universal digital block
USB	universal serial bus
USBIO	USB I/O
VTOR	vector table offset register
WCO	watch crystal oscillator
WDT	watchdog timer
WDR	watchdog reset
WoT	wake on touch
XRES	external reset
XRES_N	external reset, active low

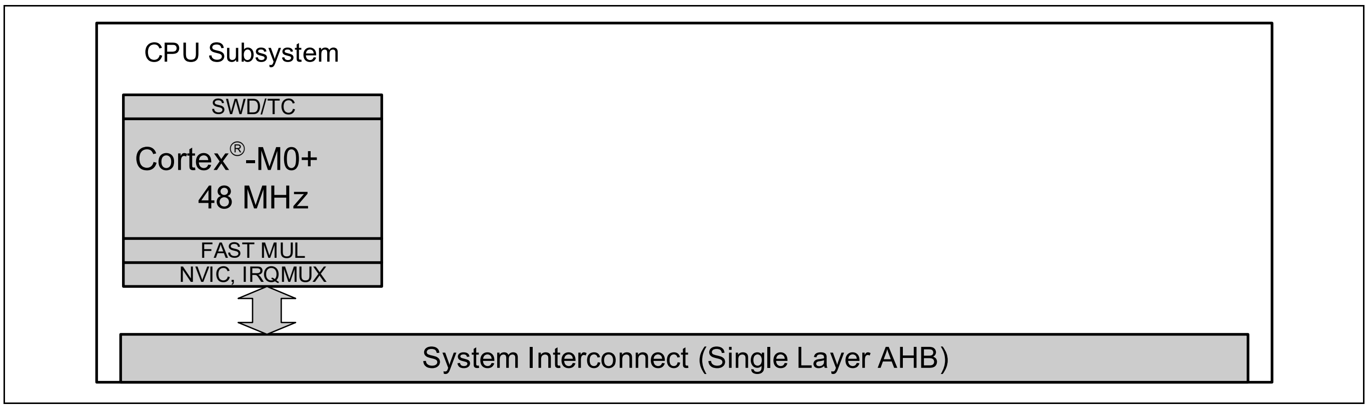
Section B: CPU system

This section encompasses the following chapters:

- **"Cortex®-M0+ CPU"** on page 23
- **"Interrupts"** on page 30
- **"Device security"** on page 41

Top level architecture

CPU system block diagram



4 Cortex®-M0+ CPU

The PSoC™ 4 Arm® Cortex®-M0+ core is a 32-bit CPU optimized for low-power operation. It has an efficient two-stage pipeline, a fixed 4-GB memory map, and supports the ARMv6-M a subset of the Thumb-2 instruction set. The Cortex®-M0+ also features a single-cycle 32-bit multiply instruction and low-latency interrupt handling. Other subsystems tightly linked to the CPU core include a nested vectored interrupt controller (NVIC), a SYSTICK timer, and debug.

This section gives an overview of the Cortex®-M0+ processor. For more details, see the Arm® Cortex®-M0+ user guide or technical reference manual, both available at **www.arm.com**.

4.1 Features

The PSoC™ 4 Cortex®-M0+ has the following features:

- Easy to use, program, and debug, ensuring easier migration from 8- and 16-bit processors
- Operates at up to 0.95-1.36 DMIPS/MHz; this helps to increase execution speed or reduce power
- Supports the Thumb instruction set for improved code density, ensuring efficient use of memory. Most instructions are 16-bits in length and it executes a subset of the Thumb-2 instruction set.
- NVIC unit to support interrupts and exceptions for rapid and deterministic interrupt response
- Supports unprivileged and privileged mode execution
- Supports optional Vector Table Offset Register (VTOR)
- Extensive debug support including:
  - SWD port
  - Breakpoints
  - Watchpoints

4.2 Block diagram

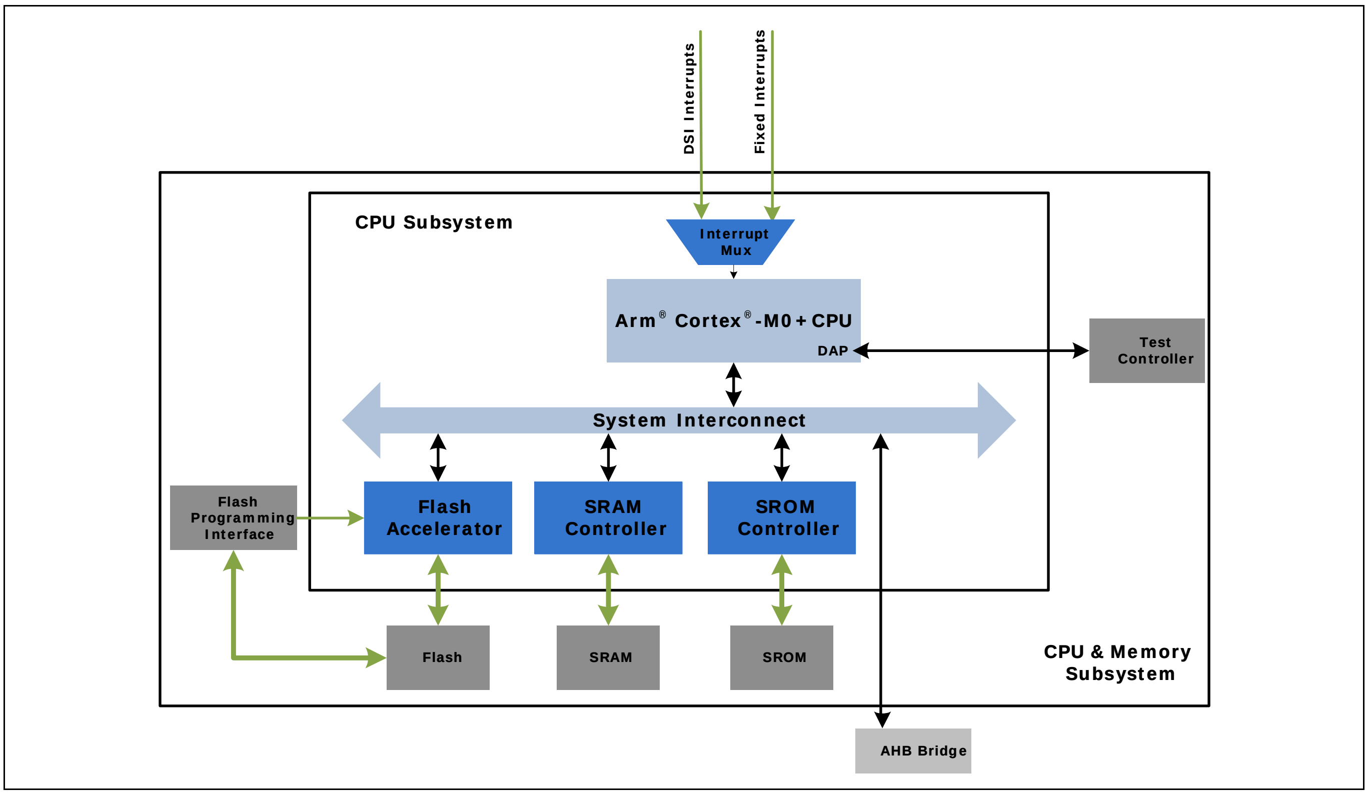


Figure 4-1. CPU subsystem block diagram

4.3 How it works

The Cortex®-M0+ is a 32-bit processor with a 32-bit data path, 32-bit registers, and a 32-bit memory interface. It supports most 16-bit instructions in the Thumb instruction set and some 32-bit instructions in the Thumb-2 instruction set.

The processor supports two operating modes (see **"Operating modes"** on page 26). It has a single-cycle 32-bit multiplication instruction.

4.4 Address map

The Arm® Cortex®-M0+ has a fixed address map allowing access to memory and peripherals using simple memory access instructions. 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. Cortex®-M0+ address map

Address range	Name	Use
0x00000000 - 0x1FFFFFFF	Code	Program code region. You can also place data here. Includes the exception vector table, which starts at address 0.
0x20000000 - 0x3FFFFFFF	SRAM	Data region. You can also execute code from this region.
0x40000000 - 0x5FFFFFFF	Peripheral	All peripheral registers. You cannot execute code from this region.
0x60000000- 0x9FFFFFFF	External RAM	Executable region for data.
0xA0000000- 0xDFFFFFFF	External device	External device memory.
0xE0000000 - 0xE00FFFFF	PPB	Peripheral registers within the CPU core.
0xE0100000 - 0xFFFFFFFF	Device	PSoC™ 4 implementation-specific.

4.5 Registers

The Cortex®-M0+ has sixteen 32-bit registers, as **Table 4-2** shows:

- 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 4-2. 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)	RW	[0x00000000]	The stack pointer (SP) is register R13. In thread mode, bit[1] of the CONTROL register indicates which 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 0x00000000 and the PSP state is indeterminate.
PSP (R13)			
LR (R14)	RW	Undefined	The link register (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions.

Table 4-2. Cortex®-M0+ registers *(continued)*

Name	Type[a)]	Reset value	Description
PC (R15)	RW	[0x00000004]	The program counter (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value from address 0x00000004. Bit[0] of the value is loaded into the EPSR T-bit at reset and must 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	[0x00000004].0	On reset, EPSR is loaded with the value bit[0] of the register [0x00000004].
IPSR	RO	0	The IPSR contains the exception number of the current ISR.
PRIMASK	RW	0	The PRIMASK register prevents activation of all exceptions with configurable priority.
CONTROL	RW	0	The CONTROL register controls the stack used when the processor is in thread mode.

a) Describes access type during program execution in thread mode and handler mode. Debug access can differ.

Table 4-3 shows how the PSR bits are assigned.

Table 4-3. 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.

Table 4-3. Cortex®-M0+ PSR bit assignments *(continued)*

Bit	PSR Register	Name	Usage
23 – 6	–	–	Reserved
5 – 0	IPSR	N/A	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 … 28 = IRQ12

Use the MSR or CPS instruction to set or clear bit 0 of the PRIMASK register. If the bit is 0, exceptions are enabled. If the bit is 1, all exceptions with configurable priority, that is, all exceptions except HardFault, NMI, and Reset, are disabled. See the **"Interrupts"** on page 30 for a list of exceptions.

4.6 Operating modes

The Cortex®-M0+ processor supports two operating modes:

- Thread mode – used by all normal applications. In this mode, the MSP or PSP can be used. The CONTROL register bit 1 determines which stack pointer is used:
  - 0 = MSP is the current stack pointer
  - 1 = PSP is the current stack pointer
- Handler mode – used to execute exception handlers. The MSP is always 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 action ensures that instructions after the ISB execute using the new stack pointer.

In Handler mode, explicit writes to the CONTROL register are ignored, because the MSP is always used. The exception entry and return mechanisms automatically update the CONTROL register.

4.7 Instruction set

The Cortex®-M0+ implements a version of the Thumb instruction set, as **Table 4-4** shows. For details, see the Cortex®-M0+ Generic User Guide.

An instruction operand can be an Arm® register, a constant, or another instruction-specific parameter. Instructions act on the operands and often store the result in a destination register. Many instructions are unable to use, or have restrictions on using, the PC or SP for the operands or destination register.

Table 4-4. Thumb instruction set

Mnemonic	Brief description
ADCS	Add with carry
ADD{S}[a)]	Add
ADR	PC-relative address to register
ANDS	Bit wise AND
ASRS	Arithmetic shift right
B{cc}	Branch {conditionally}
BICS	Bit clear
BKPT	Breakpoint
BL	Branch with link
BLX	Branch indirect with link
BX	Branch indirect
CMN	Compare negative
CMP	Compare
CPSID	Change processor state, disable interrupts
CPSIE	Change processor state, enable interrupts
DMB	Data memory barrier
DSB	Data synchronization barrier
EORS	Exclusive OR
ISB	Instruction synchronization barrier
LDM	Load multiple registers, increment after
LDR	Load register from PC-relative address
LDRB	Load register with word
LDRH	Load register with half-word
LDRSB	Load register with signed byte
LDRSH	Load register with signed half-word
LSLS	Logical shift left
LSRS	Logical shift right
MOV{S}[a]	Move
MRS	Move to general register from special register
MSR	Move to special register from general register
MULS	Multiply, 32-bit result

Table 4-4. Thumb instruction set *(continued)*

Mnemonic	Brief description
MVNS	Bit wise NOT
NOP	No operation
ORRS	Logical OR
POP	Pop registers from stack
PUSH	Push registers onto stack
REV	Byte-reverse word
REV16	Byte-reverse packed half-words
REVSH	Byte-reverse signed half-word
RORS	Rotate right
RSBS	Reverse subtract
SBCS	Subtract with carry
SEV	Send event
STM	Store multiple registers, increment after
STR	Store register as word
STRB	Store register as byte
STRH	Store register as half-word
SUB{S}[a]	Subtract
SVC	Supervisor call
SXTB	Sign extend byte
SXTH	Sign extend half-word
TST	Logical AND-based test
UXTB	Zero extend a byte
UXTH	Zero extend a half-word
WFE	Wait for event
WFI	Wait for interrupt

a) The ‘S’ qualifier causes the ADD, SUB, or MOV instructions to update APSR condition flags.

4.7.1 Address alignment

An aligned access is an operation where a word-aligned address is used for a word or multiple word access, or where a half-word-aligned address is used for a half-word access. Byte accesses are always aligned.

No support is provided for unaligned accesses on the Cortex®-M0+ processor. Any attempt to perform an unaligned memory access operation results in a HardFault exception.

4.7.2 Memory endianness

The Cortex®-M0+ uses the little-endian format, where the least-significant byte of a word is stored at the lowest address and the most significant byte is stored at the highest address.

4.8 Systick timer

The Systick timer is integrated with the NVIC and generates the SYSTICK interrupt. This interrupt can be used for task management in a real-time system. The timer has a reload register with 24 bits available to use as a countdown value. The Systick timer uses either the Cortex®-M0+ internal clock or the low-frequency clock (LFCLK) as the source.

4.9 Debug

PSoC™ 4 contains a debug interface based on SWD; it features four breakpoint (address) comparators and two watchpoint (data) comparators.

5 Interrupts

The Arm® Cortex®-M0+ (CM0+) CPU in PSoC™ 4 supports interrupts and exceptions. 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. The device provides a unified exception vector table for both interrupt handlers/ISR and exception handlers.

5.1 Features

PSoC™ 4 supports the following interrupt features:

- Supports 13 interrupts
- Nested vectored interrupt controller (NVIC) integrated with CPU core, yielding low interrupt latency
- Vector table may be placed in either flash or SRAM
- Configurable priority levels from 0 to 3 for each interrupt
- Level-triggered and pulse-triggered interrupt signals

5.2 How it works

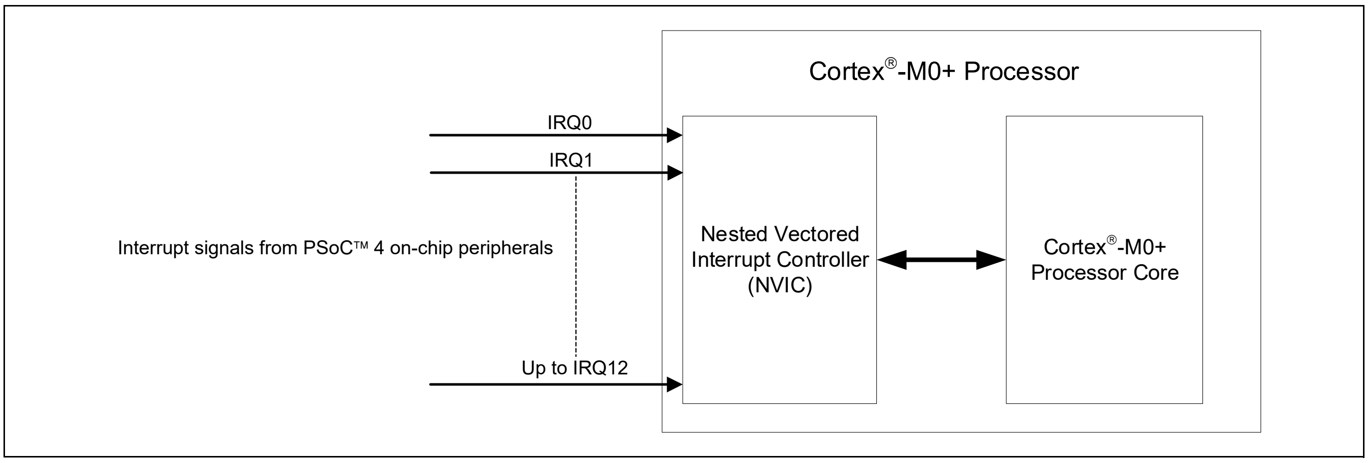

Cortex®-M0+ Processor
IRQ0
IRQ1
Interrupt signals from PSoC™ 4 on-chip peripherals
Up to IRQ12
Nested Vectored Interrupt Controller (NVIC)
Cortex®-M0+ Processor Core

Figure 5-1. PSoC™ 4 interrupts block diagram

Figure 5-1 shows the interaction between interrupt signals and the Cortex®-M0+ CPU. PSoC™ 4 has up to 13 interrupts; these interrupt signals are processed by the NVIC. The NVIC takes care of enabling/disabling individual interrupts, priority resolution, and communication with the CPU core. The exceptions are not shown in **Figure 5-1** because they are part of CM0+ core generated events, unlike interrupts, which are generated by peripherals external to the CPU.

5.3 Interrupts and exceptions - Operation

5.3.1 Interrupt/exception handling

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

1. Assuming that all the interrupt signals are initially low (idle or inactive state) and the processor is executing the main code, a rising edge on any one of the interrupt lines is registered by the NVIC. The interrupt line is now in a pending state waiting to be serviced by the CPU.
2. On detecting the interrupt request 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 5-1**. 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.

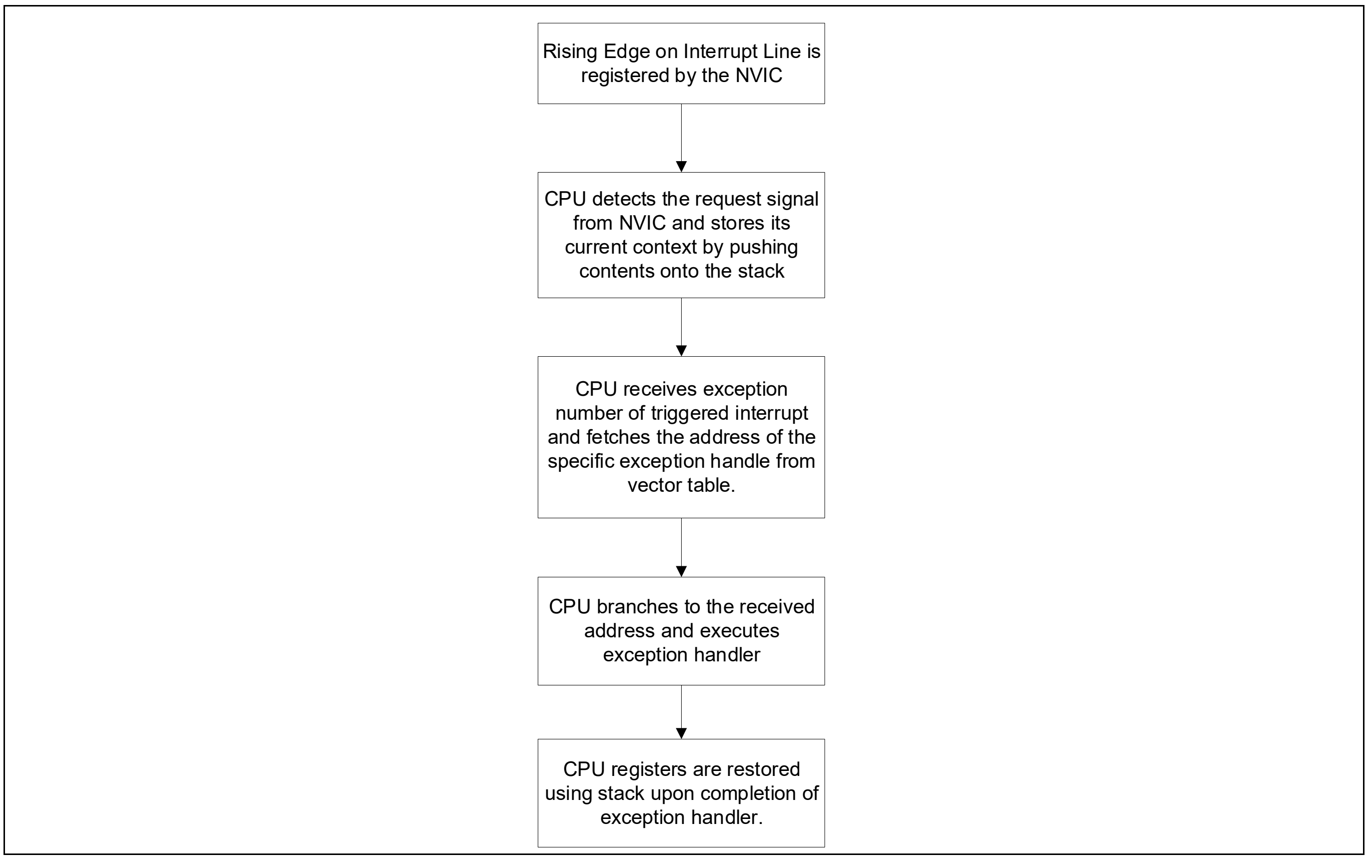

Rising Edge on Interrupt Line is registered by the NVIC
CPU detects the request signal from NVIC and stores its current context by pushing contents onto the stack
CPU receives exception number of triggered interrupt and fetches the address of the specific exception handle from vector table.
CPU branches to the received address and executes exception handler
CPU registers are restored using stack upon completion of exception handler.

Figure 5-2. 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.

5.3.2 Level and pulse interrupts

NVIC supports both level and pulse signals on the interrupt lines (IRQ0 to IRQ12). The classification of an interrupt as level or pulse is based on the interrupt source.

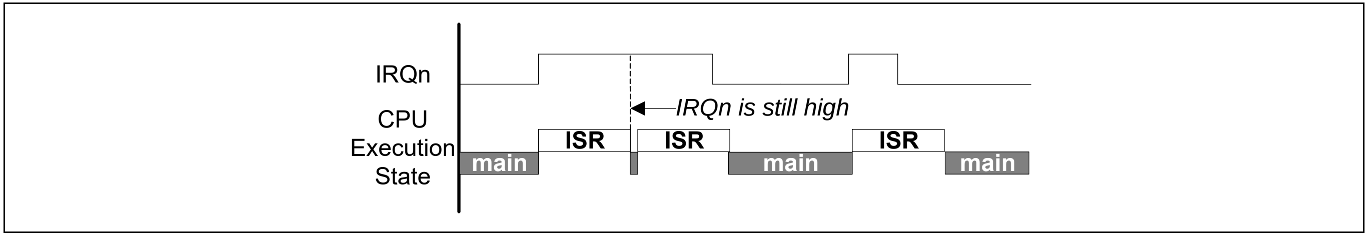


Figure 5-3. Level interrupts

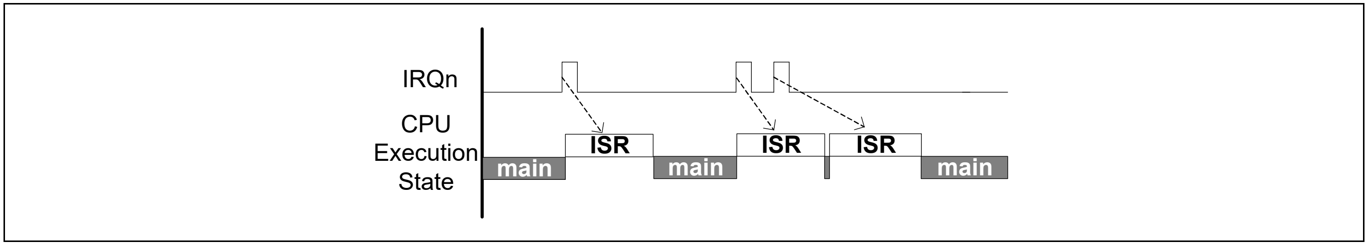


Figure 5-4. Pulse interrupts

Figure 5-3 and **Figure 5-4** 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. 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 5-4** for pulse interrupts).
4. If the interrupt signal is still high after completing the ISR, it will be pending and the ISR is executed again. **Figure 5-3** illustrates this for level triggered interrupts, where the ISR is executed as long as the interrupt signal is high.

5.3.3 Exception vector table

The exception vector table (**Table 5-1**), stores the entry point addresses for all exception handlers. The CPU fetches the appropriate address based on the exception number.

Table 5-1. Exception vector table

Exception number	Exception	Exception priority	Vector address
–	Initial Stack Pointer Value	Not applicable (NA)	Base_Address - 0x00000000 (start of flash memory) or 0x20000000 (start of SRAM)
1	Reset	–3, the highest priority	Base_Address + 0x04
2	Non Maskable Interrupt (NMI)	–2	Base_Address + 0x08
3	HardFault	–1	Base_Address + 0x0C
4-10	Reserved	NA	Base_Address + 0x10 to Base_Address + 0x28
11	Supervisory Call (SVCall)	Configurable (0 - 3)	Base_Address + 0x2C
12-13	Reserved	NA	Base_Address + 0x30 to Base_Address + 0x34
14	PendSupervisory (PendSV)	Configurable (0 - 3)	Base_Address + 0x38
15	System Timer (SysTick)	Configurable (0 - 3)	Base_Address + 0x3C
16	External Interrupt(IRQ0)	Configurable (0 - 3)	Base_Address + 0x40
…	…	Configurable (0 - 3)	…
28	External Interrupt(IRQ12)	Configurable (0 - 3)	Base_Address + 0x70

In **Table 5-1**, 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. The vector table can be located anywhere in the memory map (flash or SRAM) by modifying the Vector Table Offset Register (VTOR). This register is part of the System Control Space of CM0+ located at 0xE000ED08. This register takes bits 31:8 of the vector table address; bits 7:0 are reserved. Therefore, the vector table address should be 256 bytes aligned.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.

Reads of flash addresses 0x00000000 and 0x00000004 are redirected to the first eight bytes of SROM to fetch the stack pointer and reset vectors, unless the DIS_RESET_VECT_REL bit of the CPUSS_SYSREQ register is set. The default value of this bit at reset is 0 ensuring that reset vector is always fetched from SROM. To allow flash read from addresses 0x00000000 and 0x00000004, the DIS_RESET_VECT_REL bit should be set to '1'. The stack pointer vector holds the address that the stack pointer is loaded with on reset. The reset vector holds the address of the boot sequence. This mapping is done to use the default addresses for the stack pointer and reset vector from SROM when the device reset is released. For reset, boot code in SROM is executed first and then the CPU jumps to address 0x00000004 in flash to execute the handler in flash. The reset exception address in the SRAM vector table is never used.

Also, when the SYSCALL_REQ bit of the CPUSS_SYSREQ register is set, reads of flash address 0x00000008 are redirected to SROM to fetch the NMI vector address instead of from flash. Reset CPUSS_SYSREQ to read the flash at address 0x00000008.

The exception sources (exception numbers 1 to 15) are explained in **"Exception sources"** on page 34. The exceptions marked as Reserved in **Table 5-1** are not used, although they have addresses reserved for them in the vector table. The interrupt sources (exception numbers 16 to 28) are explained in **"Interrupt Sources"** on page 36.

5.4 Exception sources

This section explains the different exception sources listed in **Table 5-1** (exception numbers 1 to 15).

5.4.1 Reset exception

Device reset is treated as an exception in PSoC™ 4. It is always enabled with a fixed priority of –3, the highest priority exception. 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 out of supervisory read-only memory (SROM). The boot code and other data in SROM memory are programmed by Infineon®, and are not read/write accessible to external users. After completing the SROM boot sequence, the CPU code execution jumps to flash memory. Flash memory address 0x00000004 (Exception#1 in **Table 5-1**) 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.

5.4.2 Non-maskable interrupt (NMI) exception

Non-maskable interrupt (NMI) is the highest priority exception other than reset. It is always enabled with a fixed priority of –2. There are two ways to trigger an NMI exception in the device:

- **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 register (CM0P_ICSR register). Setting this bit will execute the NMI handler pointed to by the active vector table (flash or SRAM vector table).
- **System Call NMI exception:** This exception is used for nonvolatile programming operations such as flash write operation and flash checksum operation. It is triggered by setting the SYSCALL_REQ bit in the CPUSS_SYSREQ register. An NMI exception triggered by SYSCALL_REQ bit always executes the NMI exception handler code that resides in SROM. Flash or SRAM exception vector table is not used for system call NMI exception. The NMI handler code in SROM is not read/write accessible because it contains nonvolatile programming routines that should not be modified by the user.

5.4.3 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. 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 CM0+ 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.

5.4.4 Supervisor Call (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 a supervisor call that requires privileged access to the system. Note that the CM0+ in PSoC™ 4 uses a privileged mode for the system call NMI exception, which is not related to the SVCall exception (see the **"Chip operational modes"** on page 69 for details on privileged mode.) There is no other privileged mode support for SVCall at the architecture level in the device. The application developer must define the SVCall exception handler according to the end application requirements.

The priority of a SVCall exception can be configured to a value between 0 and 3 by writing to the two bit fields PRI_11[31:30] of the System Handler Priority Register 2 (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 (SHCSR) can be used to check or modify the pending status of the SVCall exception.

5.4.5 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. The PendSV exception is triggered by setting the PENDSVSET bit in the Interrupt Control State Register, CM0P_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, CM0P_ICSR. The priority of a PendSV exception can be configured to a value between 0 and 3 by writing to the two bit fields PRI_14[23:22] of the System Handler Priority Register 3 (CM0P_SHPR3). See the **Armv6-M Architecture Reference Manual** for more details.

5.4.6 SysTick exception

CM0+ CPU in PSoC™ 4 supports a system timer, referred to as SysTick, as part of its 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 SysTick exception. The exception is enabled by setting the TICKINT bit in the SysTick Control and Status Register (CM0P_SYST_CSR). The priority of a SysTick exception can be configured to a value between 0 and 3 by writing to the two bit fields PRI_15[31:30] 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 PENDSTSETb bit in the Interrupt Control State Register, CM0P_ICSR. 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, CM0P_ICSR.

5.5 Interrupt Sources

PSoC™ 4 supports up to 13 interrupts (IRQ0 to IRQ12 or exception numbers 16 – 28) from peripherals. The source of each interrupt is listed in **Table 5-2**. PSoC™ 4 provides flexible sourcing options for each interrupt line. The interrupts include standard interrupts from the on-chip peripherals such as TCPWM and serial communication block. The interrupt generated is usually the logical OR of the different peripheral states. The peripheral status register should be read in the ISR to detect which condition generated the interrupt. interrupts are usually level interrupts, which require that the peripheral status register be read in the ISR to clear the interrupt. If the status register is not read in the ISR, the interrupt will remain asserted and the ISR will be executed continuously. See the **"I/O system"** on page 44 for details on GPIO interrupts.

Table 5-2. PSoC™ 4000T MCU interrupt sources

Interrupt	Cortex®-M0+ exception no.	Interrupt source
NMI	2	SYSCALL_REQ
IRQ0	16	GPIO interrupt - Port 0
IRQ1	17	GPIO interrupt - Port 1
IRQ2	18	GPIO interrupt - Port 2
IRQ3	19	GPIO interrupt - Port 3
IRQ4	20	GPIO interrupt - All port
IRQ5	21	Watchdog timer (WDT)
IRQ6	22	Serial communication block 0 (SCB0)
IRQ7	23	Serial communication block 1 (SCB1)
IRQ8	24	MSCLP (Low power CAPSENSE™)
IRQ9	25	SPCIF interrupt
IRQ10	26	MSC (CAPSENSE™)
IRQ11	27	Timer/counter/PWM0 (TCPWM0)
IRQ12	28	Timer/counter/PWM1 (TCPWM1)

5.6 Exception priority

Exception priority is useful for exception arbitration when there are multiple exceptions that need to be serviced by the CPU. PSoC™ 4 provides 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™ 4, 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.

PSoC™ 4 supports 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 active exception. The CPU resumes execution of the lower priority exception handler after servicing the higher priority exception. The CM0+ CPU in PSoC™ 4 allows nesting of up to four 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 34.

The priority of the 13 interrupts (IRQ0 to IRQ12) can be configured by writing to the Interrupt Priority registers (CM0P_IPR). This is a group of 32-bit registers with each register storing the priority values of four interrupts, as given in **Table 5-3**. The other bit fields in the register are not used.

Table 5-3. Interrupt priority register bit definitions

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.

5.7 Enabling and disabling interrupts

The NVIC provides registers to individually enable and disable the 13 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_ISER) and the Interrupt Clear-Enable Register (CM0P_ICER) are used to enable and disable the interrupts respectively. These are 32-bit wide registers and each bit corresponds to the same numbered interrupt. These registers can also be read in software to get the enable status of the interrupts. **Table 5-4** shows the register access properties for these two registers. Note that writing zero to these registers has no effect.

Table 5-4. Interrupt enable/disable registers

Register	Operation	Bit value	Comment
Interrupt Set Enable Register (CM0P_ISER)	Write	1	To enable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled
Interrupt Clear Enable Register (CM0P_ICER)	Write	1	To disable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled

The CM0P_ISER and CM0P_ICER registers are applicable only for interrupts IRQ0 to IRQ12. 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 34.

The PRIMASK register in Cortex®-M0+ (CM0+) CPU 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 5-1**. They can be configured to a priority level between 0 and 3, 0 being the highest priority and 3 being the lowest priority. 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.

5.8 Exception states

Each exception can be in one of the following states.

Table 5-5. Exception states

Exception state	Meaning
Inactive	The exception is not active or pending. Either the exception is disabled or the enabled exception has not been triggered.
Pending	The exception request is 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 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_ICSR) contains status bits describing the various exceptions states.

- The VECTACTIVE bits ([8:0]) in the CM0P_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 bit fields 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 CM0P_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 CM0P_ICSR indicates if a NVIC generated interrupt (IRQ0 to IRQ12) is in a pending state.

5.8.1 Pending 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 13 interrupt lines by providing separate register bits for setting and clearing the pending states of the interrupts. The Interrupt Set-Pending register (CM0P_ISPR) and the Interrupt Clear-Pending register (CM0P_ICPR) are used to set and clear the pending status of the interrupt lines. These are 32-bit wide registers and each bit corresponds to the same numbered interrupt.

Table 5-6 shows the register access properties for these two registers. Note that writing zero to these registers has no effect.

Table 5-6. Interrupt set pending/clear pending registers

Register	Operation	Bit Value	Comment
Interrupt Set-Pending Register (CM0P_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 (CM0P_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 CM0P_ISER register.

Note that the CM0P_ISPR and CM0P_ICPR registers are used only for the 13 peripheral interrupts (exception numbers 16–28). 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 34.

5.9 Stack usage for 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. Cortex®-M0+ has 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 in Cortex®-M0+ 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 **"Cortex®-M0+ CPU"** on page 23 for details.

The Cortex®-M0+ uses two techniques, tail chaining and late arrival, to reduce latency in servicing exceptions. These techniques are not visible to the external user and are part of the internal processor architecture. For information on tail chaining and late arrival mechanism, visit the **Arm Infocenter**.

5.10 Interrupts and low-power modes

PSoC™ 4 allows device 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 device 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) instruction, executed by the CM0+ CPU, triggers the transition into Sleep and Deep-Sleep modes. The sequence of entering the different low-power modes is detailed in the **"Power modes"** on page 65. Chip low-power modes have two categories of fixed-function interrupt sources:

- Fixed-function interrupt sources that are available only in the Active and Deep-Sleep modes (watchdog timer interrupt,)
- Fixed-function interrupt sources that are available only in the Active mode (all other fixed-function interrupts)

5.11 Exceptions – Initialization and configuration

This section covers the different steps involved in initializing and configuring exceptions in PSoC™ 4.

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 by writing bits 31:28 of the VTOR register with the value of the flash or SRAM address at which the vector table will reside This register write is done as part of device initialization code.
   It is recommended that the vector table be available in SRAM if the application needs to 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. PSoC™ 4 Peripheral Device Library (PDL) uses the vector table in SRAM by default.
2. Configuring Individual Exceptions: The next step is to configure individual exceptions required in an application.
   a) Configure the exception or interrupt source; this includes setting up the interrupt generation conditions. The register configuration depends on the specific exception required.
   b) Define the exception handler function and write the address of the function to the exception vector table. **Table 5-1** gives the exception vector table format; the exception handler address should be written to the appropriate exception number entry in the table.
   c) Set up the exception priority, as explained in **“Exception priority”** on page 36.
   d) Enable the exception, as explained in **“Enabling and disabling interrupts”** on page 37.

5.12 Registers

Register name	Description
CM0P_ISER	Interrupt Set-Enable Register
CM0P_ICER	Interrupt Clear Enable Register
CM0P_ISPR	Interrupt Set-Pending Register
CM0P_ICPR	Interrupt Clear-Pending Register
CM0P_IPR	Interrupt Priority Registers
CM0P_ICSR	Interrupt Control State Register
CM0P_AIRCR	Application Interrupt and Reset Control Register
CM0P_SCR	System Control Register
CM0P_CCR	Configuration and Control Register
CM0P_SHPR2	System Handler Priority Register 2
CM0P_SHPR3	System Handler Priority Register 3
CM0P_SHCSR	System Handler Control and State Register
CM0P_SYST_CSR	Systick Control and Status Register
CPUSS_CONFIG	CPU Subsystem Configuration Register
CPUSS_SYSREQ	System Request Register

5.13 Associated documents

Armv6-M architecture reference manual – This document explains the Arm® Cortex®-M0+ architecture, including the instruction set, NVIC architecture, and CPU register descriptions.

6 Device security

PSoC™ 4 offers a number of options for protecting user designs from unauthorized access or copying. Disabling debug features and enabling flash protection provide a high level of security.

The debug circuits are enabled by default and can only be disabled in firmware. If disabled, the only way to re-enable them is to erase the entire device, clear flash protection, and reprogram the device with new firmware that enables debugging. Additionally, all device interfaces can be permanently disabled for applications concerned about phishing attacks due to a maliciously reprogrammed device or attempts to defeat security by starting and interrupting flash programming sequences. Permanently disabling interfaces is not recommended for most applications because the designer cannot access the device. For more information, as well as a discussion on flash row and chip protection, see the **CY8C4xxx, CYBLxxxx programming specifications**.

Note: *Because all programming, debug, and test interfaces are disabled when maximum device security is enabled, PSoC™ 4 devices with full device security enabled may not be returned for failure analysis.*

6.1 Features

The PSoC™ 4 device security system has the following features:

- User-selectable levels of protection.
- In the most secure case provided, the chip can be "locked" such that it cannot be acquired for test/debug and it cannot enter erase cycles. Interrupting erase cycles is a known way for hackers to leave chips in an undefined state and open to observation.
- CPU execution in a privileged mode by use of the non-maskable interrupt (NMI). When in privileged mode, NMI remains asserted to prevent any inadvertent return from interrupt instructions causing a security leak.

In addition to these, the device offers protection for individual flash row data.

6.2 How it works

6.2.1 Device security

The CPU operates in normal user mode or in privileged mode, and the device operates in one of four protection modes: BOOT, OPEN, PROTECTED, and KILL. Each mode provides specific capabilities for the CPU software and debug. You can change the mode by writing to the CPUSS_PROTECTION register.

- **BOOT mode**: The device comes out of reset in BOOT mode. It stays there until its protection state is copied from supervisor flash to the protection control register (CPUSS_PROTECTION). The debug-access port is stalled until this has happened. BOOT is a transitory mode required to set the part to its configured protection state. During BOOT mode, the CPU always operates in privileged mode.
- **OPEN mode**: This is the factory default. The CPU can operate in user mode or privileged mode. In user mode, flash can be programmed and debugger features are supported. In privileged mode, access restrictions are enforced.
- **PROTECTED mode**: The user may change the mode from OPEN to PROTECTED. This mode disables all debug access to user code or memory. In protected mode, only few registers are accessible; debug access to registers to reprogram flash is not available. The mode can be set back to OPEN but only after completely erasing the flash.
- **KILL mode**: The user may change the mode from OPEN to KILL. This mode removes all debug access to user code or memory, and the flash cannot be erased. Access to most registers is still available; debug access to registers to reprogram flash is not available. The part cannot be taken out of KILL mode; devices in KILL mode may not be returned for failure analysis.

6.2.2 Flash security

The PSoC™ 4 devices 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 rows. You can assign one of two protection levels to each row; see **Table 6-1**. Flash protection levels can only be changed by performing a complete flash erase.

For more details, see the **“Nonvolatile memory programming”** on page 192.

Table 6-1. Flash protection levels

Protection setting	Allowed	Not allowed
Unprotected	External read and write, Internal read and write	–
Full Protection	External read[a)] Internal read	External write, Internal write

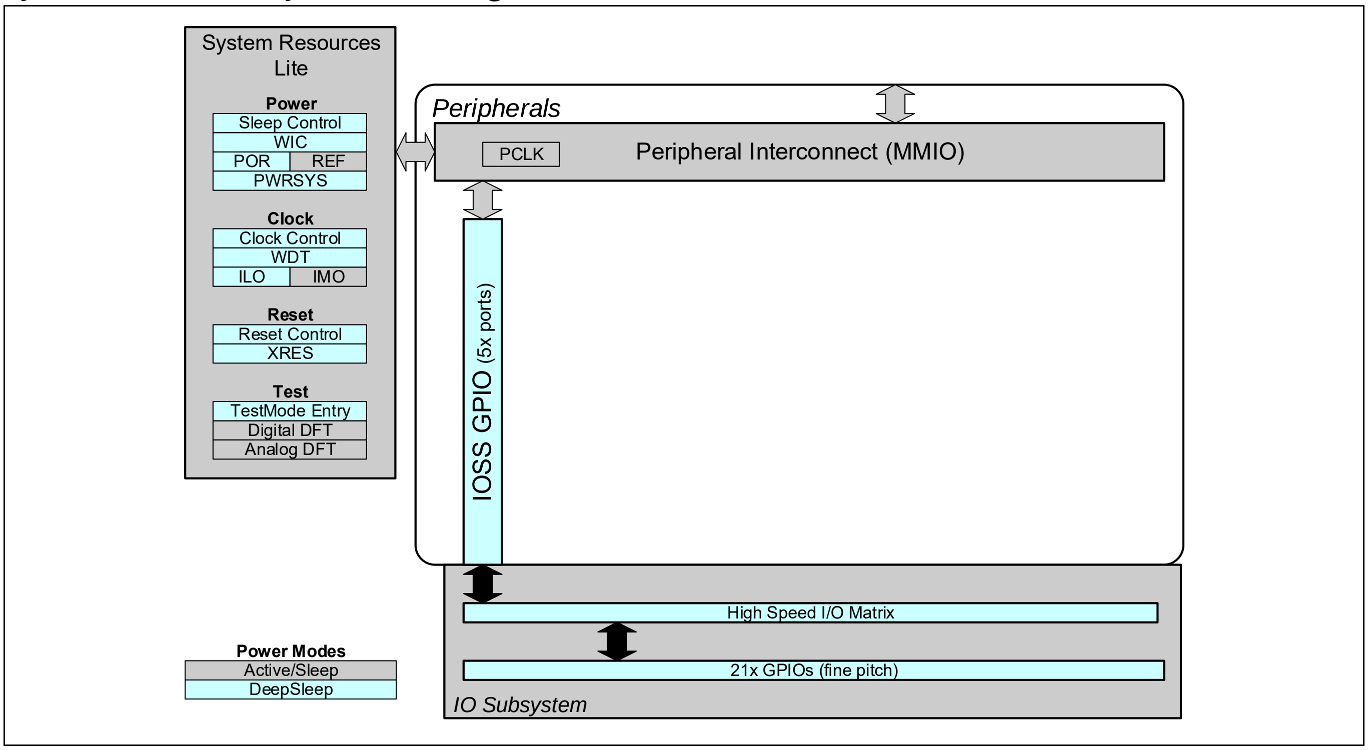
a) To protect the device from external read operations, you should change the device protection settings to PROTECTED.

Section C: System resources subsystem (SRSS)

This section encompasses the following chapters:

Top level architecture

System resource subsystem block diagram



7 I/O system

This chapter explains the PSoC™ 4 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™ 4 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™ 4 MCU family are grouped into ports; a port can have a maximum of eight GPIO pins. The PSoC™ 4000T MCU device has a maximum of 21 GPIOs arranged in 5 ports.

7.1 Features

The PSoC™ 4 GPIOs have these features:

- Analog and digital input and output capabilities
- Eight drive strength modes
- Separate port read and write registers
- Separate I/O supplies and voltages for groups of I/O ports
- Edge-triggered interrupts on rising edge, falling edge, or on both the edges, on pin basis
- Slew rate control
- Hold mode for latching previous state (used for retaining I/O state in Deep-Sleep mode)
- Selectable CMOS and low-voltage LVTTL input buffer mode
- CAPSENSE™ support blocks

7.2 GPIO interface overview

PSoC™ 4 is equipped with analog and digital peripherals. **Figure 7-1** shows an overview of the routing between the peripherals and pins.

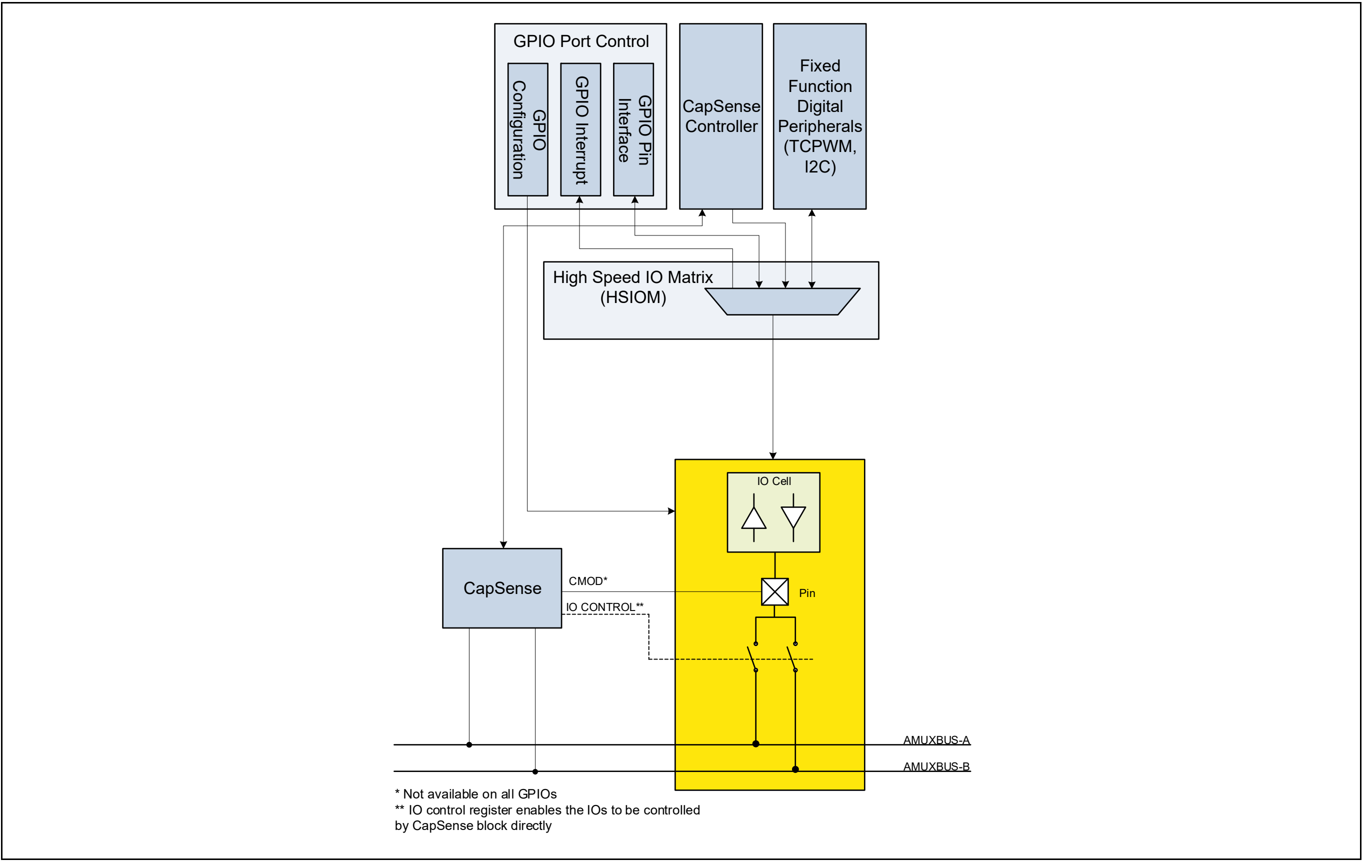

GPIO Port Control
GPIO Configuration
GPIO Interrupt
GPIO Pin Interface
CapSense Controller
Fixed Function Digital Peripherals (TCPWM, I2C)
High Speed IO Matrix (HSIOM)
IO Cell
CapSense
CMOD*
IO CONTROL**
Pin
AMUXBUS-A
AMUXBUS-B
* Not available on all GPIOs
** IO control register enables the IOs to be controlled by CapSense block directly

Figure 7-1. 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). HSIOM contains multiplexers to connect between a peripheral selected by the user and the pin. The CAPSENSE™ block is connected to the GPIO pins through the AMUX buses.

7.3 I/O cell architecture

Figure 7-2 shows the I/O cell architecture present in every GPIO pin. 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.

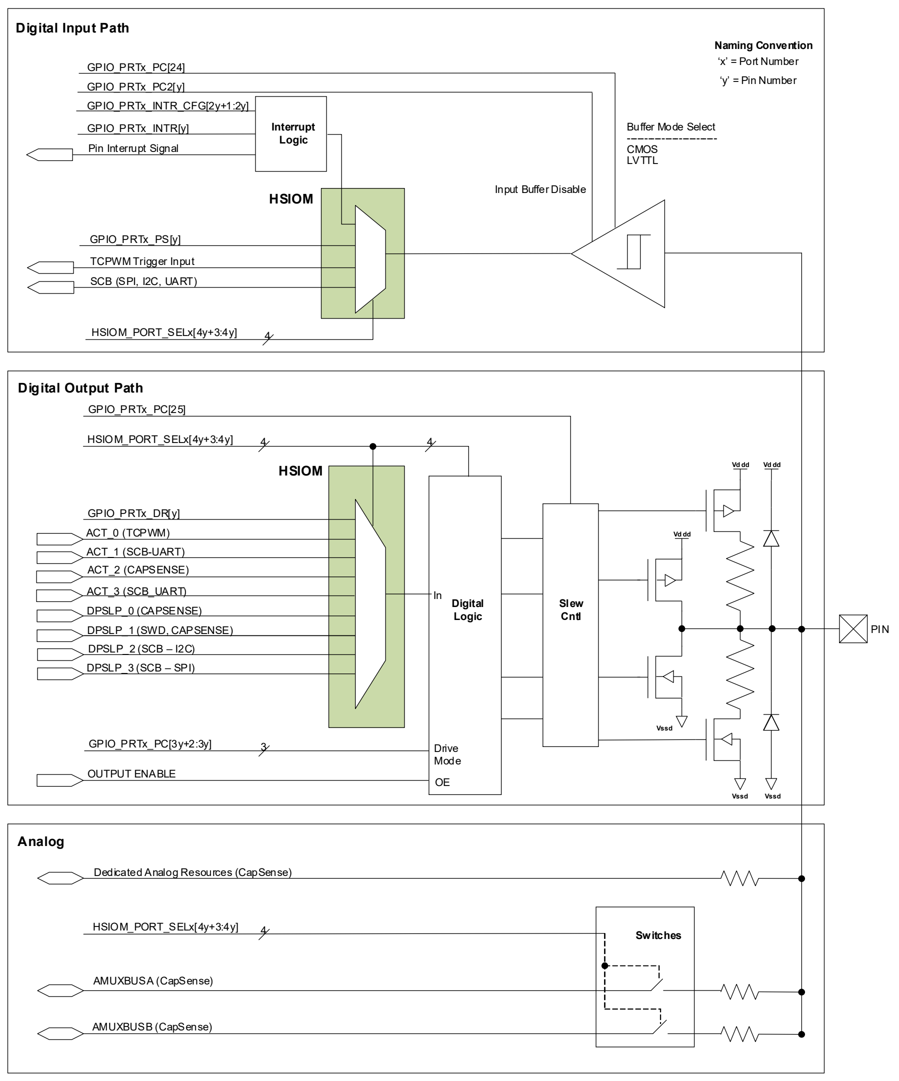


Figure 7-2. GPIO block diagram

7.3.1 Digital input buffer

The digital input buffer provides a high-impedance buffer for the external digital input. The buffer is enabled and disabled by the INP_DIS bit of the Port Configuration Register 2 (GPIO_PRTx_PC2, 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™ 4 MCU: PSoC™ 4000T 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. The buffer is configurable for the following modes:

- CMOS
- LVTTL

These buffer modes are selected by the PORT_VTRIP_SEL bit (GPIO_PRTx_PC[24])of the Port Configuration register.

Table 7-1. Input buffer modes

PORT_VTRIP_SEL	Input buffer mode
0b	CMOS
1b	LVTTL

The threshold values for each mode can be obtained from the device datasheet. The output of the input buffer is connected to the HSIOM for routing to the selected peripherals. Writing to the HSIOM port select register (HSIOM_PORT_SELx) selects the peripheral. The digital input peripherals connected to the HSIOM, shown in **Figure 7-2**, are pin dependent. See the device datasheet to know the functions available for each pin.

7.3.2 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 three primary types of control sources are CPU registers, configurable digital peripherals instantiated in the programmable UDB/DSI fabric, 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 device datasheet.

In PSoC™ 4, I/Os are driven with V_{DDD} supply. Each GPIO pin has ESD diodes to clamp the pin voltage to the V_{DDD} source. Ensure that the voltage at the pin does not exceed the I/O supply voltage V_{DDD} and drop below V_{SSD}. For the absolute maximum and minimum GPIO voltage, see the device datasheet.

The digital output driver can be enabled and disabled using the DSI signal from the peripheral or data register (GPIO_PRTx_DR) associated with the output pin. See **"High-speed I/O matrix"** on page 49 to know about the peripheral source selection for the data and to enable or disable control source selection.

7.3.2.1 Drive modes

Each I/O is individually configurable into one of eight drive modes using the Port Configuration register, GPIO_PRTx_PC. **Table 7-2** lists the drive modes. **Figure 7-3** is a simplified output driver diagram that shows the pin view based on each of the eight drive modes.

Table 7-2. Drive mode settings

GPIO_PRTx_PC ('x' denotes port number and 'y' denotes pin number)				
Bits	**Drive mode**	**Value**	**Data = 1**	**Data = 0**
3y+2: 3y	SEL'y'	Selects Drive Mode for Pin 'y' (0 ≤ y ≤ 7)		
	High-Impedance Analog	0	High Z	High Z
	High-impedance Digital	1	High Z	High Z
	Resistive Pull Up	2	Weak 1	Strong 0
	Resistive Pull Down	3	Strong 1	Weak 0
	Open Drain, Drives Low	4	High Z	Strong 0
	Open Drain, Drives High	5	Strong 1	High Z
	Strong Drive	6	Strong 1	Strong 0
	Resistive Pull Up and Down	7	Weak 1	Weak 0

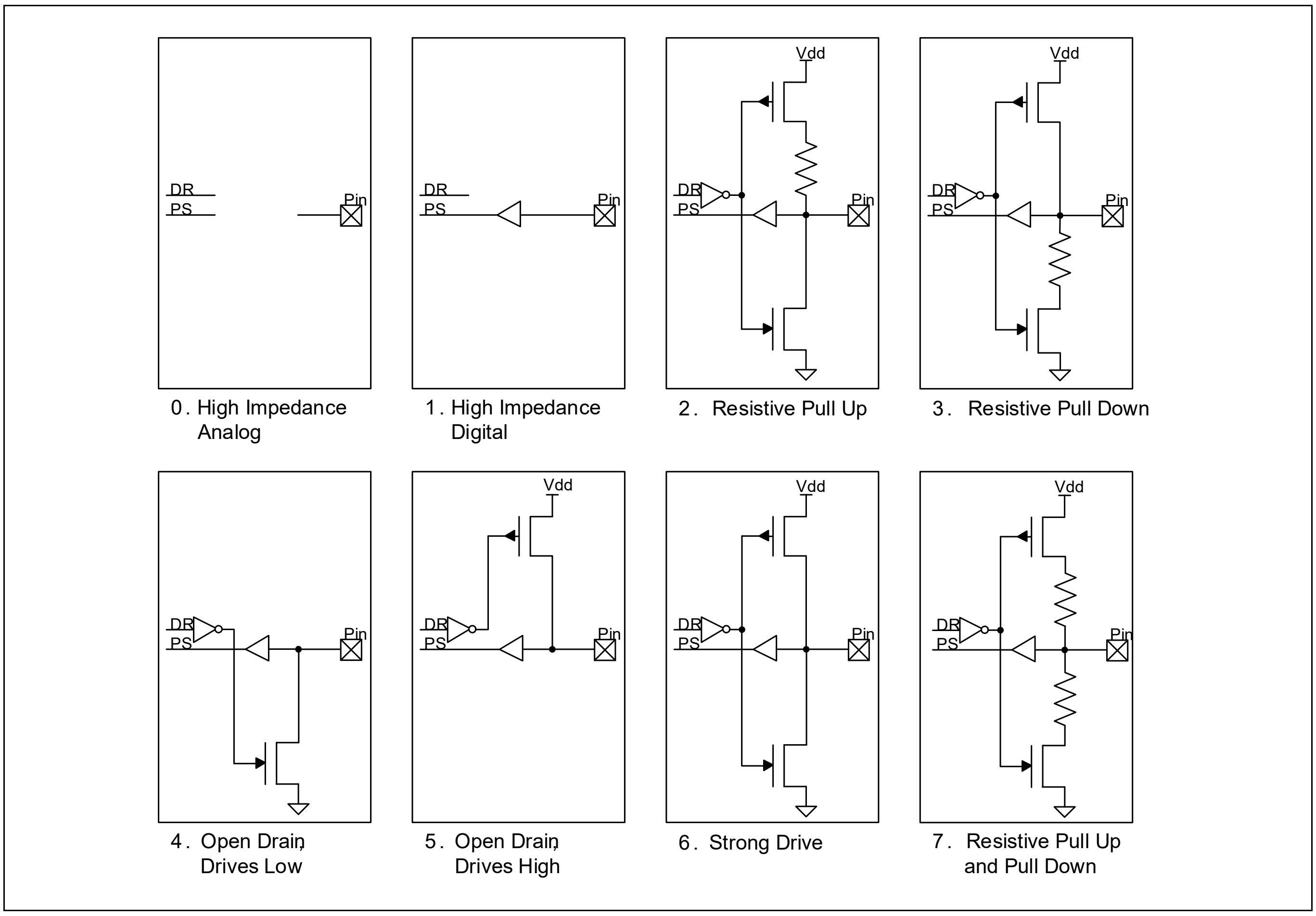

DR
PS
Pin
Vdd
0 . High Impedance Analog
1 . High Impedance Digital
2 . Resistive Pull Up
3 . Resistive Pull Down
4 . Open Drain Drives Low
5 . Open Drain Drives High
6 . Strong Drive
7 . Resistive Pull Up and Pull Down

Figure 7-3. I/O drive mode block diagram

- High-impedance analog

High-impedance analog mode is the default reset state; both output driver and digital input buffer are turned off. This state prevents an external voltage from causing a current to flow into the digital input buffer. This drive mode is recommended for pins that are floating or that support an analog voltage. High-impedance analog pins cannot be used for digital inputs. Reading the pin state register returns a 0x00 regardless of the data register value. To achieve the lowest device current in low-power modes, unused GPIOs must be configured to the high-impedance analog mode.

- High-impedance digital

High-impedance digital mode is the standard high-impedance (High Z) state recommended for digital inputs. In this state, the input buffer is enabled for digital input signals.

- 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 input is open drain low and resistive pull-down is used when 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. The pins can be used as digital input or output in these modes. Therefore, these modes are widely used in bi-directional digital communication. Open drain drive high mode is used when signal is externally pulled down and open drain drive low is used when signal is externally pulled high. A common application for open drain drives low mode is driving I^{2}C 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 must 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 resistive 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 used when the bus is driven by other signals that may cause shorts.

7.3.2.2 Slew rate control

GPIO pins have fast and slow output slew rate options in strong drive mode; this is configured using PORT_SLOW bit of the Port Configuration register (GPIO_PRTx_PC[25]). Slew rate is individually configurable for each port. This bit is cleared by default 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; hence, the slow option is recommended for low-frequency signals or signals without strict timing constraints.

7.4 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. All ports connect directly to the HSIOM. The HSIOM_PORT_SELx register is provided to select the peripheral. It is a 32-bit wide register available for each port, with each pin occupying four bits. This register provides up to 16 different options for a pin as listed in **Table 7-3**.

Refer to the example of **Switching a GPIO pin connection between analog and digital sources**.

Table 7-3. PSoC™ 4000T MCU HSIOM port settings

HSIOM_PORT_SELx ('x' denotes port number and 'y' denotes pin number)			
Bits	**Name (SEL'y')**	**Value**	**Description (Selects pin 'y' source ($0 \leq y \leq 7$))**
4y+3 : 4y	GPIO	0	Pin is regular firmware-controlled I/O or connected to dedicated hardware block.
	GPIO_DSI	1	The output is firmware-controlled, but OE is controlled from DSI.
	DSI_DSI	2	Both output and OE are controlled from DSI.
	DSI_GPIO	3	The output is controlled from DSI, but OE is firmware-controlled.
	CSD_SENSE	4	Pin is a CSD sense pin (analog mode).
	CSD_SHIELD	5	Pin is a CSD shield pin (analog mode).
	AMUXA	6	Pin is connected to AMUXBUS-A.
	AMUXB	7	Pin is connected to AMUXBUS-B.
	ACTIVE_0	8	Pin-specific Active source #0 (TCPWM Output).
	ACTIVE_1	9	Pin-specific Active source #1 (SCB-UART).
	ACTIVE_2	10	Pin-specific Active source #2.
	ACTIVE_3	11	Pin-specific Active source #3.
	DEEP_SLEEP_0	12	Pin-specific Deep-Sleep source #0 (CAPSENSE™).
	DEEP_SLEEP_1	13	Pin-specific Deep-Sleep source #1 (SWD, CAPSENSE™).
	DEEP_SLEEP_2	14	Pin-specific Deep-Sleep source #2 (SCB-I^2C).
	DEEP_SLEEP_3	15	Pin-specific Deep-Sleep source #3 (SCB-SPI).

Note: The Active and Deep-Sleep sources are pin dependent. See the "Pinouts" section of the device datasheet for more details on the features supported by each pin. Refer to the example of ***Switching a GPIO pin connection between analog and digital sources****.*

7.5 Behavior in low-power modes

Table 7-4 shows the status of GPIOs in low-power modes.

Table 7-4. GPIO in low-power modes

Low-power mode	Status
Sleep	GPIOs are active and can be driven by peripherals such as CAPSENSE™, TCPWM and SCBs, which can operate in sleep mode. Input buffers are active; thus an interrupt on any I/O can be used to wake up the CPU. AMUXBUS connections are available.
Deep-Sleep	GPIO output pin states are latched and maintain the last output driver state, except the I^{2}C and SPI pins. SCB (I^{2}C and SPI) block can work in the deep-sleep mode and can wake up the CPU on address match or SPI slave select event. Input buffers are also active in this mode; pin interrupts are functional. AMUXBUS connections are available.

7.6 Interrupt

In the PSoC™ 4 device, all the port pins have the capability to generate interrupts. As shown in **Figure 7-2**, the pin signal is routed to the interrupt controller through the GPIO Edge Detect block.

Figure 7-4 shows the GPIO Edge Detect block architecture.

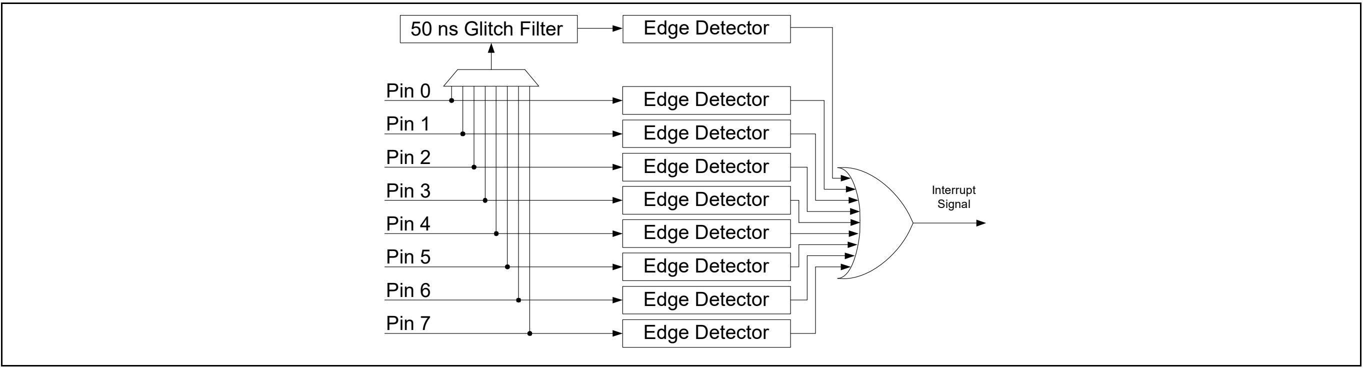


Figure 7-4. GPIO edge detect block architecture

An edge detector is present at each pin. It is capable of detecting rising edge, falling edge, and both edges without reconfiguration. The edge detector is configured by writing into the EDGE_SEL bits of the Port Interrupt Configuration register, GPIO_PRTx_INTR_CFG, as shown in **Table 7-5**.

Table 7-5. 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

Besides the pins, an edge detector is also present at the glitch filter output. This filter can be used on one of the pins of a port. The pin is selected by writing to the FLT_SEL field of the GPIO_PRTx_INTR_CFG register as shown in **Table 7-6**.

Table 7-6. 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

The edge detector outputs of a port are ORed together and then routed to the interrupt controller (NVIC in the CPU subsystem). Thus, there is only one interrupt vector per port. On a pin interrupt, it is required to know which pin caused an interrupt. This is done by reading the Port Interrupt Status register, GPIO_PRTx_INTR. This register not only includes the information on which pin triggered the interrupt, it also includes the pin status; it allows the CPU to read both information in a single read operation. This register has one more important use – to clear the interrupt. Writing '1' to the corresponding status bit clears the pin interrupt. It is important to clear the interrupt status bit; otherwise, the interrupt will occur repeatedly for a single trigger or respond only once for multiple triggers, which is explained later in this section. Also, note that when the Port Interrupt Control Status register is read when an interrupt is occurring on the corresponding port, it can result in the interrupt not being properly detected. Therefore, when using GPIO interrupts, it is recommended to read the status register only inside the corresponding interrupt service routine and not in any other part of the code. **Table 7-7** shows the Port Interrupt Status register bit fields.

Table 7-7. Port interrupt status register

GPIO_PRTx_INTR	Description
0000b to 0111b	Interrupt status on pin 0 to pin 7. Writing '1' to the corresponding bit clears the interrupt.
1000b	Interrupt status from the glitch filter
10000b to 10111b	Pin 0 to Pin 7 status
11000b	Glitch filter output status

The edge detector block output is routed to the Interrupt Source Multiplexer shown in **Figure 7-4**, which gives an option of Level and Rising Edge detect. 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, it is important to clear the interrupt status bit if the Edge Detect block is used.

Each IO port has an associated bit field in the GPIO_INTR_CAUSE register. The PORT_INT bit field reflects the IO port's interrupt line (bit field i reflects "gpio_interrupts[i]" for IO port i), as shown in Table 7-8. The register is used when the system uses a shared/combined interrupt line "gpio_interrupt". The SW ISR reads the register to determine which IO port(s) is responsible for the shared/combined interrupt line "gpio_interrupt" being set.

Once, the IO port(s) is determined, the IO port's INTR register is read to determine the IO pad(s) in the IO port that caused the interrupt.

Table 7-8. Interrupt cause

PORT_INT	Description
00001b	Interrupt caused by Port 0
00010b	Interrupt caused by Port 1
00100b	Interrupt caused by Port 2
01000b	Interrupt caused by Port 3
10000b	Interrupt caused by Port 4

7.7 Peripheral connections

7.7.1 Firmware controlled GPIO

For standard firmware-controlled GPIO using registers, the GPIO mode must be selected in the HSIOM_PORT_SELx register. GPIO_PRTx_DR is the data register used to read and write the output data for the GPIOs. A write operation to this register changes the GPIO output to the written value. Note that a read operation reflects the output data written to this register and not the current state of the GPIOs. Using this 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_DR_SET, GPIO_PRTx_DR_CLR, and GPIO_PRTx_INV – are provided to set, clear, and invert the output data respectively of a specific pin in a port without affecting other pins. Writing '1' into these registers will set, clear, or invert; writing '0' will have no affect on the pin status.

GPIO_PRTx_PS is the I/O pad register that provides the state of the GPIOs when read. Writes to this register have no effect.

7.7.2 CAPSENSE™

CAPSENSE™ (MSC) supports AMUXBUS interface. The pins that support MSC can be configured as CAPSENSE™ widgets such as buttons, slider elements, touchpad elements, or proximity sensors. CAPSENSE™ also requires shield lines. See the **PSoC™ 4 and PSoC™ 6 MCU CAPSENSE™ design guide** for more details. **Table 7-9** shows the GPIO and HSIOM settings required for CAPSENSE™.

Table 7-9. CAPSENSE™ settings

CAPSENSE™ pin[a)]	GPIO drive mode (GPIO_PRTx_PC)	Digital input buffer setting (GPIO_PRTx_PC2)	HSIOM setting
			AMUX mode
Sensor	High-Impedance Analog	Disable Buffer	CSD_SENSE
Shield	High-Impedance Analog	Disable Buffer	CSD_SHIELD
CMOD1	High-Impedance Analog	Disable Buffer	AMUXBUS A
CMOD2	High-Impedance Analog	Disable Buffer	AMUXBUS A

a) The IO pin connections can be directly controlled by the CAPSENSE™ block.

7.7.3 Serial communication block (SCB)

SCB blocks can be configured as UART, I^2C, and SPI have dedicated connections to the I/O pins. See the device datasheet for details on these dedicated pins. When UART and SPI mode are used, the SCB controls the digital output buffer drive mode for the input pin in order to keep the pin in the high-impedance state. The SCB block disables the output buffer at the UART Rx pin and MISO pin when configured as SPI master, and MOSI and select line when configured as SPI slave. This functionality overrides the drive mode settings, which is done using the GPIO_PRTx_PC register.

7.7.4 Timer, counter, and pulse width modulator (TCPWM) block

TCPWM has dedicated connections to the pin. See the device datasheet for details on these dedicated pins. Note that when the TCPWM block inputs such as start and stop are taken from the pins, the drive mode can be only high-z digital because the TCPWM block disables the output buffer at the input pins.

7.8 Registers

Name	Description
GPIO_PRTx_DR	Port Output Data Register
GPIO_PRTx_DR_SET	Port Output Data Set Register
GPIO_PRTx_DR_CLR	Port Output Data Clear Register
GPIO_PRTx_DR_INV	Port Output Data Inverting Register
GPIO_PRTx_PS	Port Pin State Register - Reads the logical pin state of I/O
GPIO_PRTx_PC	Port Configuration Register - Configures the output drive mode, input threshold, and slew rate
GPIO_PRTx_PC2	Port Secondary Configuration Register - Configures the input buffer of I/O pin
GPIO_PRTx_INTR_CFG	Port Interrupt Configuration Register
GPIO_PRTx_INTR	Port Interrupt Status Register
HSIOM_PORT_SELx	HSIOM Port Selection Register
GPIO_INTR_CAUSE	Port Interrupt Cause Register

Note: *The 'x' in the GPIO register name denotes the port number. For example, GPIO_PTR1_DR is the Port 1 output data register.*

8 Clocking system

The PSoC™ 4 clock system includes these clock resources:

- Two internal clock sources:
  - 24-48 MHz internal main oscillator (IMO)
  - 20-80 kHz (typically 40-kHz) internal low-speed oscillator (ILO)
- One external clock source (EXTCLK) generated using a signal from an I/O pin
- High-frequency clock (HFCLK) of up to 48 MHz, selected from IMO or external clock
- Low-frequency clock (LFCLK) sourced by ILO
- Dedicated prescaler for system clock (SYSCLK) of up to 48 MHz sourced by HFCLK
- Two 16-bit peripheral clock dividers
- Two fractional dividers (one 16.5 fractional dividers and one 24.5 fractional divider) for accurate clock generation
- Four peripheral clocks

8.1 Block diagram

Figure 8-1 gives a generic view of the clocking system in PSoC™ 4 devices.

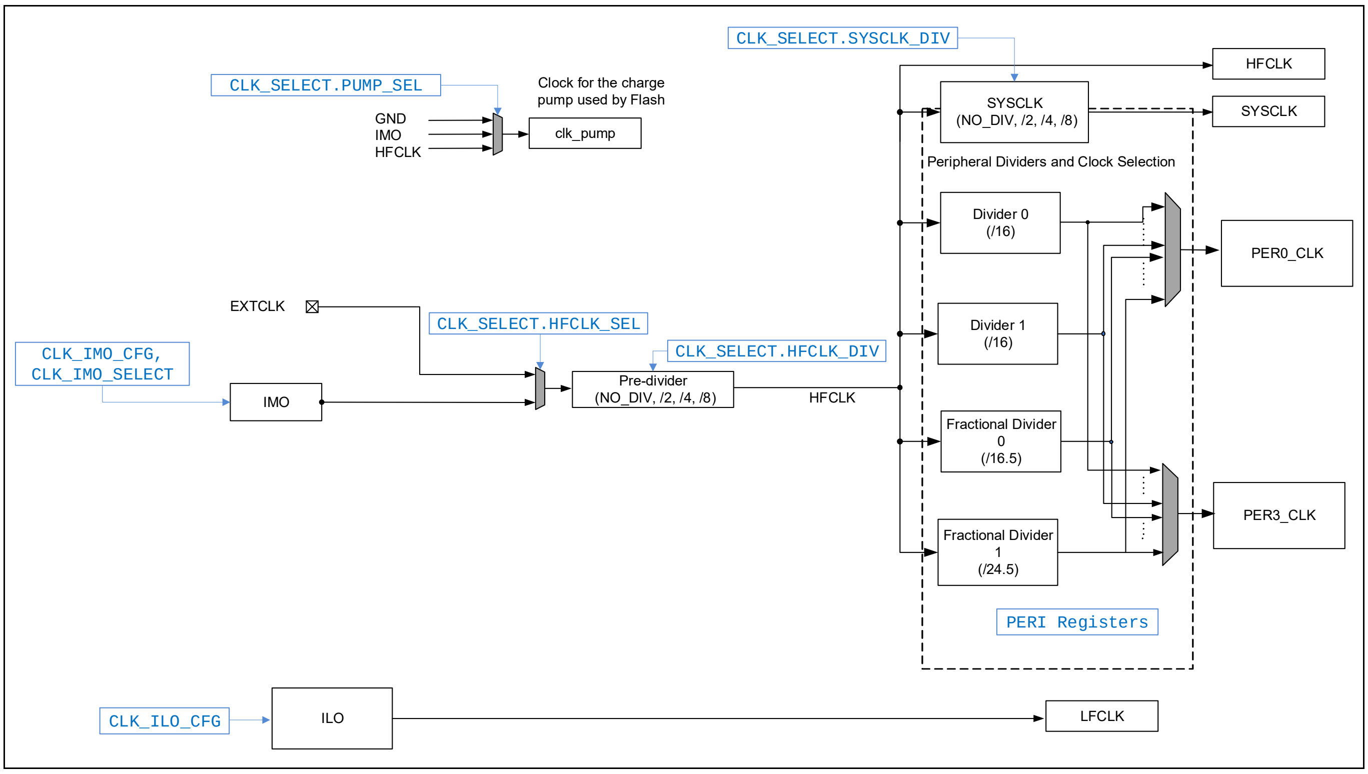

CLK_SELECT.SYSCLK_DIV
HFCLK
CLK_SELECT.PUMP_SEL
Clock for the charge pump used by Flash
GND
IMO
HFCLK
clk_pump
SYSCLK (NO_DIV, /2, /4, /8)
SYSCLK
Peripheral Dividers and Clock Selection
Divider 0 (/16)
PER0_CLK
EXTCLK
CLK_SELECT.HFCLK_SEL
CLK_SELECT.HFCLK_DIV
Divider 1 (/16)
CLK_IMO_CFG, CLK_IMO_SELECT
IMO
Pre-divider (NO_DIV, /2, /4, /8)
HFCLK
Fractional Divider 0 (/16.5)
Fractional Divider 1 (/24.5)
PER3_CLK
PERI Registers
CLK_ILO_CFG
ILO
LFCLK

Figure 8-1. Clocking system block diagram

Three clock sources in the device are IMO, EXTCLK, and ILO, as shown in **Figure 8-1**. The HFCLK mux selects the HFCLK source from the EXTCLK or the IMO. The HFCLK frequency can be a maximum of 48 MHz.

8.2 Clock sources

8.2.1 Internal main oscillator (IMO)

The IMO is an accurate, high-speed internal (crystal-less) oscillator that is available as the main clock source during Active and Sleep modes. It is the default clock source for the device. Its frequency can be changed in 4-MHz steps between 24 MHz and 48 MHz. Refer to the device datasheet for the exact specifications of clocks.

The IMO frequency is changed using the CLK_IMO_SELECT register, as shown in **Table 8-1**. The default frequency is 24 MHz.

Table 8-1. IMO frequency

CLK_IMO_SELECT[2:0]	Nominal IMO frequency	Corresponding SFLASH registers with TRIM values
0	24 MHz	SFLASH_IMO_TRIM_LT0, SFLASH_IMO_TCTRIM_LT0
1	28 MHz	SFLASH_IMO_TRIM_LT4, SFLASH_IMO_TCTRIM_LT4
2	32 MHz	SFLASH_IMO_TRIM_LT8, SFLASH_IMO_TCTRIM_LT8
3	36 MHz	SFLASH_IMO_TRIM_LT12, SFLASH_IMO_TCTRIM_LT12
4	40 MHz	SFLASH_IMO_TRIM_LT16, SFLASH_IMO_TCTRIM_LT16
5	44 MHz	SFLASH_IMO_TRIM_LT20, SFLASH_IMO_TCTRIM_LT20
6	48 MHz	SFLASH_IMO_TRIM_LT24, SFLASH_IMO_TCTRIM_LT24

To get the accurate IMO frequency, trim registers are provided – CLK_IMO_TRIM1 provides coarse trimming with a step size of 120 kHz, CLK_IMO_TRIM2 is for fine trimming with a step size of 15 kHz, and the TCTRIM field in CLK_IMO_TRIM3 is for temperature compensation. Trim settings are generated during manufacturing for every frequency that can be selected by CLK_IMO_SELECT. The trim settings CLK_IMO_TRIM1 and CLK_IMO_TRIM3 are stored in SFLASH.

The trim settings are loaded during device startup; however, firmware can load new trim values and change the frequency in run time. Follow the algorithm in **Figure 8-2** to change the IMO frequency.

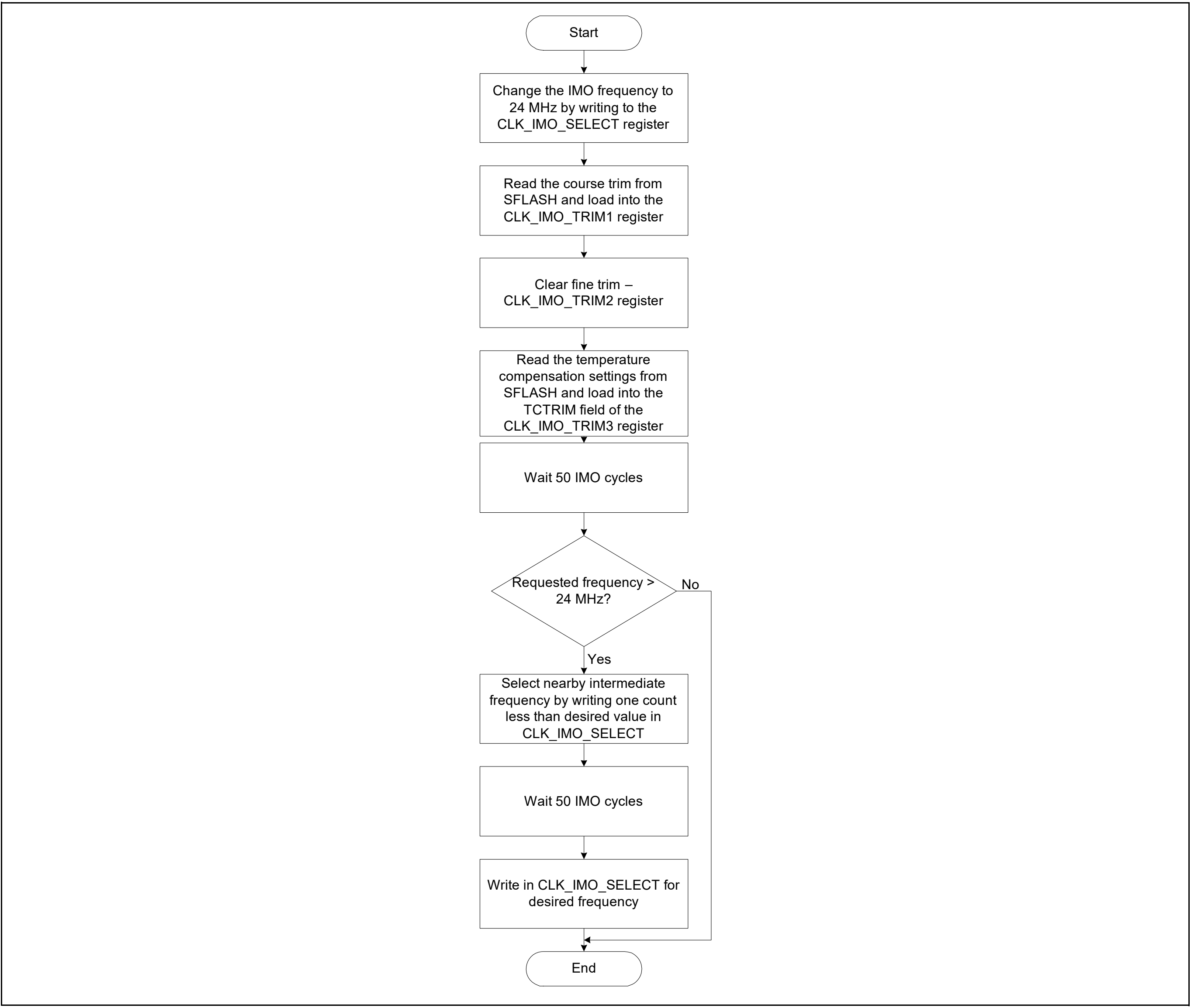

Start
Change the IMO frequency to 24 MHz by writing to the CLK_IMO_SELECT register
Read the course trim from SFLASH and load into the CLK_IMO_TRIM1 register
Clear fine trim – CLK_IMO_TRIM2 register
Read the temperature compensation settings from SFLASH and load into the TCTRIM field of the CLK_IMO_TRIM3 register
Wait 50 IMO cycles
Requested frequency > 24 MHz?
No
Yes
Select nearby intermediate frequency by writing one count less than desired value in CLK_IMO_SELECT
Wait 50 IMO cycles
Write in CLK_IMO_SELECT for desired frequency
End

Figure 8-2. Change IMO frequency

8.2.1.1 Startup behavior

After reset, the IMO is configured for 24-MHz operation. During the “boot” portion of startup, trim values are read from flash and the IMO is configured to achieve datasheet specified accuracy.

8.2.1.2 Programming clock (36-MHz)

IMO must be set to 48 MHz to program the flash. It is used to drive the charge pumps of flash and for program/erase timing purposes.

8.2.2 Internal low-speed oscillator (ILO)

The ILO operates with no external components and outputs a stable clock at 40-kHz nominal. The ILO is relatively low power and low accuracy. Refer to the device datasheet for ILO specifications. It can be calibrated periodically using a higher accuracy, HFCLK to improve accuracy. The ILO is available in all power modes. The ILO is used as the system low-frequency clock (LFCLK) in the device, which is used to generate low-frequency clocks. The ILO is enabled and disabled with ENABLE bit in CLK_ILO_CONFIG register.

8.2.3 External clock (EXTCLK)

The external clock (EXTCLK) is a MHz range clock that can be generated from a signal on a designated PSoC™ 4000T pin (P0.4). This clock may be used instead of the IMO as the source of the system high-frequency clock, HFCLK. Refer to the device datasheet for the required specifications of the external clock. The device always starts up using the IMO and the EXTCLK must be enabled in user mode; so the device cannot be started from a reset, which is clocked by the EXTCLK.

When manually configuring a pin as the input to the 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"** on page 44 for more details.

8.3 Clock distribution

PSoC™ 4 clocks are developed and distributed throughout the device, as shown in **Figure 8-1**. The distribution configuration options are as follows:

- HFCLK input selection
- LFCLK input selection
- Pump clock selection
- SYSCLK prescaler configuration
- Peripheral divider configuration

8.3.1 High-frequency clock (HFCLK) input selection

HFCLK in PSoC™ 4 has two input options: IMO and EXTCLK. Refer to the PSoC™ 4000T MCU registers TRM for the details of these registers and included bit fields.

Pre-divider is provided for HFCLK to limit the peak current of the device. The divider options are 2, 4, and 8 configured using HFCLK_DIV bits of the CLK_SELECT register. Default divider is 4.

8.3.2 Low-frequency (LFCLK) input selection

Only ILO can be the source for LFCLK in the PSoC™ 4000T MCU device.

8.3.3 System clock (SYSCLK) prescaler configuration

The SYSCLK Prescaler allows the device to divide the HFCLK before use as SYSCLK, which allows for non-integer relationships between peripheral clocks and the system clock. SYSCLK must be equal to or faster than all other clocks in the device that are derived from HFCLK. The SYSCLK prescaler is capable of dividing the HFCLK by powers of 2 between 2^0 = 1 and 2^3 = 8. The prescaler divide value is set using register CLK_SELECT bits SYSCLK_DIV, as described in **Table 8-3**. The prescaler is initially configured to divide by 1.

Table 8-3. SYSCLK prescaler divide value bits SYSCLK_DIV

Name	Description
SYSCLK_DIV[3:0]	SYSCLK prescaler divide value 0: SYSCLK = HFCLK 1: SYSCLK = HFCLK/2 2: SYSCLK = HFCLK/4 3: SYSCLK = HFCLK/8

8.3.4 Peripheral clock divider configuration

PSoC™ 4000T has four clock dividers, which include two 16-bit clock dividers, one 16.5 bit fractional clock divider, and one 24.5-bit clock divider. Fractional clock dividers allow the clock divisor to include a fraction of 0..31/32. The formula for the output frequency of a fractional divider is Fout = Fin / (INT16_DIV + 1 + (FRAC5_DIV/32)). For example, a 16.5-divider with an integer divide value of 2 (INT16_DIV=3, FRAC5_DIV=0), produces signals to generate a 16-MHz clock from a 48-MHz HFCLK. A 16.5-divider with an integer divide value of 3 (INT16_DIV=4, FRAC5_DIV=0), produces signals to generate a 12-MHz clock from a 48-MHz HFCLK. A 16.5-divider with an integer divide value of 2 (INT16_DIV=3) and a fractional divider of 16 (FRAC5_DIV=16) produces signals to generate a 13.7-MHz clock from a 48-MHz HFCLK. Not all 13.7-MHz clock periods are equal in size; half of them will be 3 HFCLK cycles and half of them will be 2 HFCLK cycles.

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 1 HFCLK cycle. The divide value for each of the two 16 integer clock dividers are configured with the PERI_DIV_16_CTLx registers and the one 16.5-bit fractional clock dividers are configured with the PERI_DIV_16_5_CTLx registers. **Table 8-4** and **Table 8-5** describe the configurations for these registers. The 24.5-bit fractional dividers is configured using the PERI_DIV_24_5_CTL register. **Table 8-6** describes the configuration for these registers.

Table 8-4. Non-fractional peripheral clock divider configuration register PERI_DIV_16_CTLx

Bits	Name	Description
0	EN_x	Divider enabled. HW sets this field to '1' as a result of an ENABLE command. HW sets this field to '0' as a result on a DISABLE command.
23:8	INT16_DIV_x	Integer division by (1+INT16_DIV). Allows for integer divisions in the range [1, 65536].

Table 8-5. Fractional peripheral clock divider configuration register PERI_DIV_16_5_CTLx

Bits	Name	Description
0	EN_x	Divider enabled. HW sets this field to '1' as a result of an ENABLE command. HW sets this field to '0' as a result on a DISABLE command.
7:3	FRAC5_DIV_x	Fractional division by (FRAC5_DIV/32). Allows for fractional divisions in the range [0, 31/32]. Note that fractional division results in clock jitter as some clock periods may be 1 “clk_hf” cycle longer than other clock periods.
23:8	INT16_DIV_x	Integer division by (1+INT16_DIV). Allows for integer divisions in the range [1, 65,536].

Table 8-6. Fractional peripheral clock divider configuration register PERI_DIV_24_5_CTL

Bits	Name	Description
0	EN_x	Divider enabled. Hardware sets this field to ‘1’ as a result of an ENABLE command and to ‘0’ as a result of a DISABLE command.
7:3	FRAC5_DIV_0	Fractional division by (FRAC5_DIV/32). Allows for fractional divisions in the range [0, 31/32]. Note that fractional division results in clock jitter as some clock periods may be 1 HFCLK cycle longer than other clock periods.
31:8	INT24_DIV_0	Integer division by (1+INT24_DIV). Allows for integer divisions in the range [1, 16,777,216].

This register acts as the command register for all 16-bit integer dividers and two fractional dividers. Each divider can be enabled using the PERI_DIV_CMD register format is as follows:

Bit	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
Description	Enable	Disable	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	PA_SEL_TYPE		PA_SEL_DIV						SEL_TYPE		SEL_DIV					

The SEL_TYPE field specifies the type of divider being configured. This field is '1' for the 16-bit integer divider, and ‘2’ for the 16.5-bit fractional divider, and ‘3’ for the 24.5-bit fractional divider.

The SEL_DIV field specifies the divider on which the command (ENABLE/DISABLE) is performed. For this integer divider, the number ranges from 0 to 63.When SEL_DIV = 63 and SEL_TYPE = 3, no divider is specified.

The (PA_SEL_TYPE, PA_SEL_DIV) field pair allows a divider to be phase-aligned with another divider. The PA_SEL_DIV specifies the divider which is phase aligned. Any enabled divider can be used as a reference. The PA_SEL_TYPE specifies the type of the divider being phase aligned. When PA_SEL_DIV = 63 and PA_SEL_TYPE = 3, HFCLK is used as a reference.

Consider a 48-MHz HFCLK and a need for a 12-MHz divided clock A and a 8-MHz divided clock B. Clock A uses a 16-bit integer divider 0 and is created by aligning it to HF_CLK ((PA_SEL_TYPE, PA_SEL_DIV) is (3, 63)) and DIV_16_CTL0.INT16_DIV is 3. Clock B uses the integer divider 1 and is created by aligning it to clock A ((PA_SEL_TYPE, PA_SEL_DIV) is (1, 0)) and DIV_16_CTL1.INT16_DIV is 5. This guarantees that clock B is phase-aligned with clock A as the smallest common multiple of the two clock periods is 12 HFCLK cycles, the clocks A and B will be aligned every 12 HFCLK cycles. Note that clock B is phase-aligned to clock A, but still uses HFCLK as a reference clock for its divider value.

Each peripheral block in PSoC™ has a unique peripheral clock (PERI#_CLK) associated with it. Each of the peripheral clocks have a multiplexed input, which can take the input clock from any of the existing clock dividers. **Table 8-7** shows the mapping of the MUX output to the corresponding peripheral blocks (shown in **Figure 8-1**). Any of the peripheral clock dividers can be mapped to a specific peripheral by using their respective PERI_PCLK_CTLx register.

Table 8-7. PSoC™ 4000T MCU device peripheral clock multiplexer output mapping

PERI#_CLK	Peripheral
0	SCB0
1	SCB1
2	TCPWM0
3	TCPWM1

Table 8-8. Programmable clock control register - PERI_PCLK_CTLx

Bits	Name	Description
0	SEL_DIV	Specifies one of the dividers of the divider type specified by SEL_TYPE. If SEL_DIV is ‘4’ and SEL_TYPE is “1”, then the fifth (zero being first) 16-bit clock divider will be routed to the mux output for peripheral clock_x. Similarly, if SEL_DIV is “0” and SEL_TYPE is “2”, then the first 16.5 clock divider will be routed to the mux output.
7:6	SEL_TYPE	0: Do not use 1: 16.0 (integer) clock dividers 2: 16.5 (fractional) clock dividers 3: 24.5 (fractional) clock dividers

8.4 Low-power mode operation

The high-frequency clocks including the IMO, EXTCLK, HFCLK, SYSCLK and peripheral clocks operate only in Active and Sleep modes. The ILO and LFCLK operate in all power modes.

8.5 Register list

Table 8-9. Clocking system register list

Register name	Description
CLK_IMO_TRIM1	IMO Trim Register - This register contains IMO trim for coarse correction.
CLK_IMO_TRIM2	IMO Trim Register - This register contains IMO trim for fine correction.
CLK_IMO_TRIM3	IMO Trim Register - This register contains the temperature compensation trim settings for IMO and trim settings to adjust the step size of the coarse and fine correction of IMO frequency.
PWR_BG_TRIM1	Bandgap Trim Registers - These registers control the trim of the bandgap reference, allowing manipulation of the voltage references in the device.
PWR_BG_TRIM2	
CLK_ILO_CONFIG ILO	Configuration Register - This register controls the ILO configuration.
CLK_IMO_CONFIG IMO	Configuration Register - This register controls the IMO configuration.
CLK_SELECT	Clock Select - This register controls clock tree configuration and selects different sources for the system clocks.
PERI_DIV_16_CTLx	Peripheral Clock Divider Control Registers - These registers configure the peripheral clock dividers, setting integer divide value, and enabling or disabling the divider.
PERI_DIV_16_5_CTLx	Peripheral Clock Fractional Divider Control Registers - These registers configure the peripheral clock dividers, setting fractional divide value, and enabling or disabling the divider.
PERI_PCLK_CTLx	Programmable Clock Control Registers - These registers are used to select the input clocks to peripherals.
PERI_DIV_24_5_CTL	Peripheral Clock Fractional Divider Control Registers - These registers configure the peripheral clock dividers, setting the fractional divide value and enabling or disabling the divider.

9 Power supply and monitoring

PSoC™ 4 is capable of operating from a 1.71 V to 5.5 V externally supplied voltage. This is supported through one of the two following operating ranges:

- 2.0 V to 5.50 V supply input to the internal regulators
- 1.71 V to 1.89 V[1)] direct supply

There are two internal regulators to support the various power modes - Active digital regulator and Deep-Sleep regulator.

9.1 Block diagram

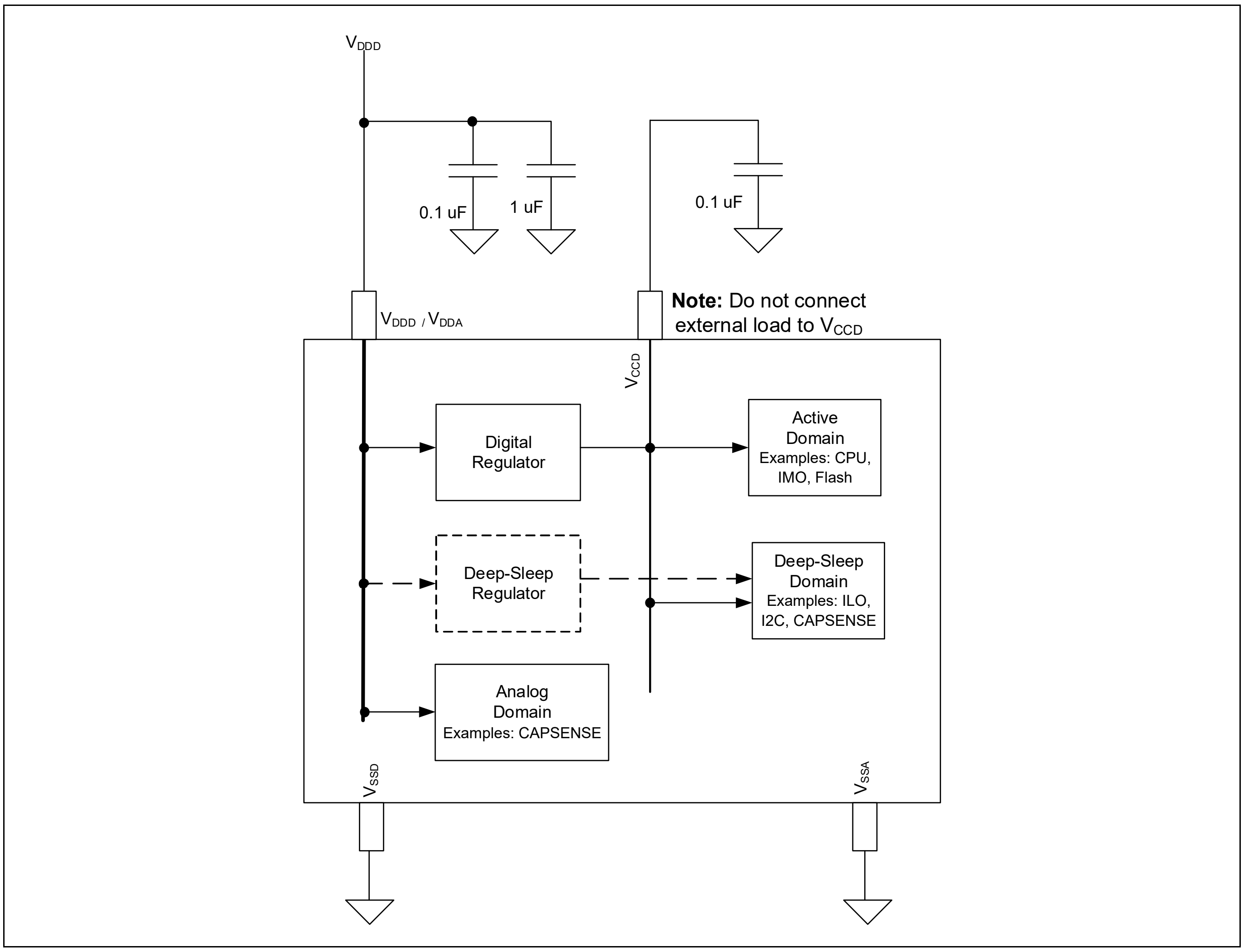

V_{DDD}
0.1 uF
1 uF
0.1 uF
V_{DDD} / V_{DDA}
Note: Do not connect external load to V_{CCD}
V_{CCD}
Digital Regulator
Active Domain Examples: CPU, IMO, Flash
Deep-Sleep Regulator
Deep-Sleep Domain Examples: ILO, I2C, CAPSENSE
Analog Domain Examples: CAPSENSE
V_{SSD}
V_{SSA}

Figure 9-1. Power system block diagram

Figure 9-1 shows the power system diagram and all the power supply pins. The system has one regulator in Active mode for the digital circuitry. There is no analog regulator; the analog circuits run directly from the V_{DDD} input. There is a separate regulator for Deep-Sleep mode.

1) When the system supply is in the range 1.71 V to 1.89 V, only direct supply (internal regulator bypass) option can be used. The selection can be made depending on the user's system capability. Note that the supply voltage cannot go above 1.89 V for the direct supply option because it will damage the device.

The supply voltage range is 1.71 V to 5.5 V with all functions and circuits operating in that range. The device allows two distinct modes of power supply operation: unregulated external supply and regulated external supply modes.

9.2 Power supply scenarios

The following diagrams illustrate the different ways in which the device is powered.

9.2.1 Single 2 V to 5.5 V regulated supply

If a 2-V to 5.5-V supply is to be used as the regulated power supply input, it should be connected as shown in **Figure 9-2**.

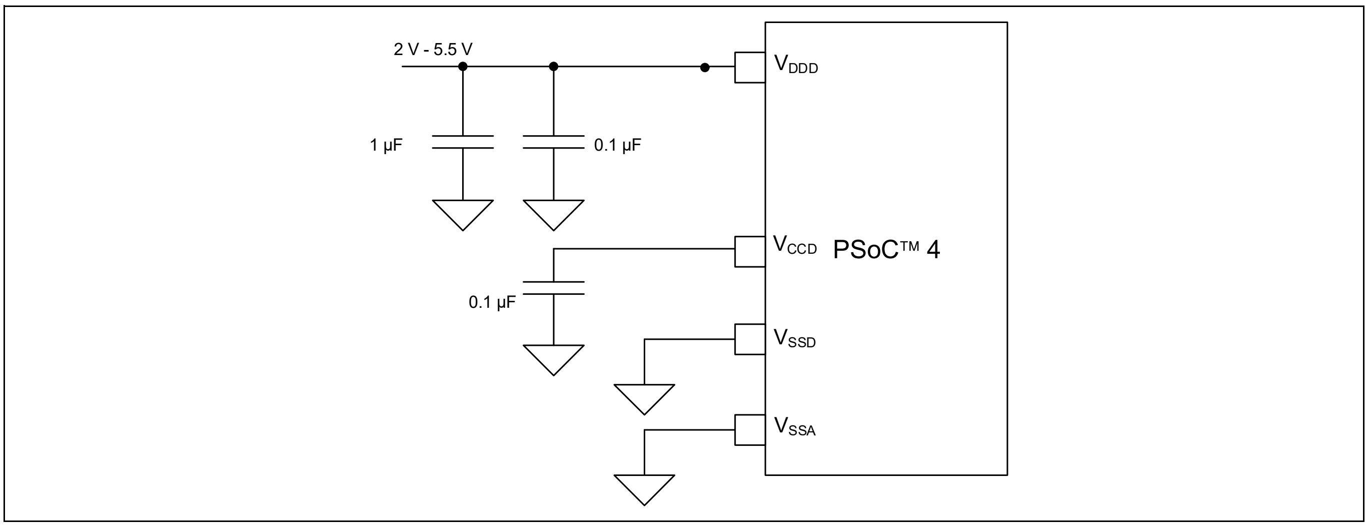


Figure 9-2. Single regulated V_{DDD} supply

In this mode, the device is powered by an external power supply that can be anywhere in the range of 2 V to 5.5 V. This range is also designed for battery-powered operation; for instance, the chip can be powered from a battery system that starts at 3.5 V and works down to 2 V. In this mode, the internal regulator supplies the internal logic. For supply voltage less than 2 V, the internal regulator need to bypass. The V_{CCD} pin need to short to the V_{DDD} pin as shown in the **Figure 9-3**. The V_{CCD} output must be bypassed to ground via a 0.1 µF external ceramic capacitor.

Bypass capacitors are also required from V_{DDD} to ground; typical practice for systems in this frequency range is to use a bulk capacitor in the 1 µF to 10 µF range in parallel with a smaller ceramic capacitor (0.1 µF, for example). Note that these are simply rules of thumb and that, for critical applications, the PCB layout, lead inductance, and the bypass capacitor parasitic should be simulated to design and obtain optimal bypassing.

9.2.2 Direct 1.71 V to 1.89 V supply

In direct supply configuration, V_{CCD} and V_{DDD} are shorted together and connected to a 1.71-V to 1.89-V supply. This regulated supply should be connected to the device, as shown in **Figure 9-3**.

In this mode, V_{CCD} and V_{DDD} pins are shorted together and bypassed.

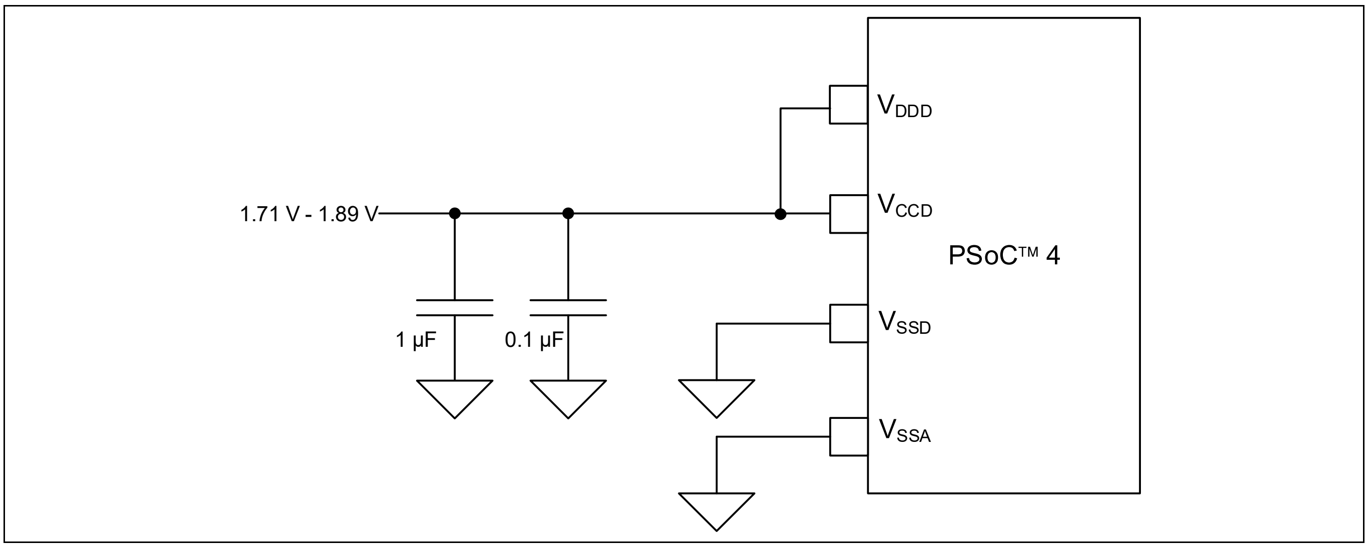


Figure 9-3. Single unregulated V_{DDD} supply

9.3 How it works

The regulators in **Figure** power the various domains of the device. All the core regulators and analog circuit draw their input power from the V_{DDD} pin supply.

9.3.1 Regulator summary

9.3.1.1 Active digital regulator

Table 9-1. Regulator status in different power modes

Mode	Active digital regulator	Deep-Sleep regulator
Deep-Sleep	Off	On
Sleep	On	On
Active	On	On

For external supplies from 2 V and 5.5 V, the Active digital regulator provides the main digital logic in Active and Sleep modes. This regulator has its output connected to a pin (V_{CCD}) and requires an external decoupling capacitor (0.1 µF X5R).

For supplies below 2 V, V_{CCD} must be supplied directly. In this case, V_{CCD} and V_{DDD} must be shorted together, as shown in **Figure 9-3**. The Active digital regulator is available only in Active and Sleep power modes.

9.3.1.2 Deep-Sleep regulator

This regulator supplies the circuits that remain powered in Deep-Sleep mode, such as the ILO and SCB (I^2C/SPI), and low-power comparator. The Deep-Sleep regulator is available in all power modes. In Active and Sleep power modes, the main output of this regulator is connected to the output of the Active digital regulator (V_{CCD}).

9.4 Voltage monitoring

The voltage monitoring system includes power-on-reset (POR) and brownout detection (BOD).

9.4.1 Power-on-reset (POR)

POR circuits provide a reset pulse during the initial power ramp. POR circuits monitor V_{CCD} voltage. Typically, the POR circuits are not very accurate with respect to trip-point. POR circuits are used during initial chip power-up and then disabled.

9.4.1.1 Brownout-detect (BOD)

The BOD circuit protects the operating or retaining logic from possibly unsafe supply conditions by applying reset to the device. BOD circuit monitors the V_{CCD} voltage. The BOD circuit generates a reset if a voltage excursion dips below the minimum V_{CCD} voltage required for safe operation (see the device datasheet for details). The system will not come out of RESET until the supply is detected to be valid again.

To ensure reliable operation of the device, the watchdog timer should be used in all designs. Watchdog timer provides protection against abnormal brownout conditions that may compromise the CPU functionality. See **"Watchdog timer"** on page 70 for more details.

9.5 Register list

Table 9-2. Power supply and monitoring register list

Register name	Description
PWR_CONTROL	Power mode control register – This register allows configuration of device power modes and regulator activity.

10 Power modes

The PSoC™ 4 provides three power modes, intended to minimize the average power consumption for a given application. The power modes, in the order of decreasing power consumption, are:

- Active
- Sleep
- Deep-Sleep

Active, Sleep, and Deep-Sleep are standard Arm®-defined power modes, supported by the Arm® CPUs.

The power consumption in different power modes is controlled by using the following methods:

- Enabling/disabling peripherals
- Powering on/off internal regulators
- Powering on/off clock sources
- Powering on/off other portions of the PSoC™ 4

Figure 10-1 illustrates the various power modes and the possible transitions between them.

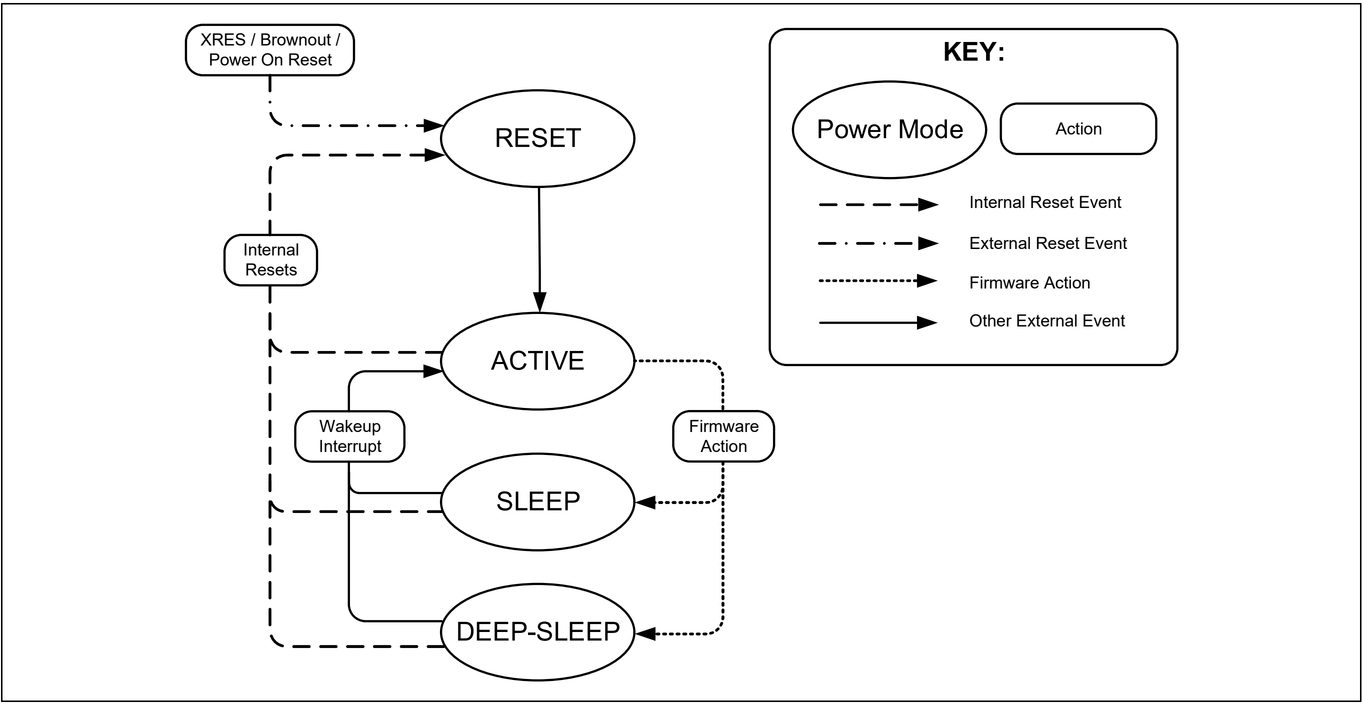


Figure 10-1. Power mode transitions State diagram

Note: *Arm® nomenclature for Deep-Sleep power mode is 'SLEEPDEEP'.*

Table 10-1 illustrates the power modes offered by PSoC™ 4.

Table 10-1. PSoC™ 4 power modes

Power mode	Description	Entry condition	Wakeup sources	Active clocks	Wakeup action	Available regulators
Active	Primary mode of operation; all peripherals are available (programmable).	Wakeup from other power modes, internal and external resets, brownout, power on reset	Not applicable	All (programmable)	N/A	All regulators are available.
Sleep	CPU enters Sleep mode and SRAM is in retention; all peripherals are available (programmable).	Manual register write	Any enabled interrupt	All (programmable) except CPU clock	Interrupt	All regulators are available.
Deep-Sleep	All internal supplies are driven from the Deep-Sleep regulator. IMO and high-speed peripherals are off. Only the low-frequency clock is available. Interrupts from low-speed, asynchronous peripherals, or CAPSENSE™ can cause a wakeup. Upon touch detection, MSCLP interrupts the CPU to change state to Active.	Manual register write	GPIO interrupt, watchdog timer, CAPSENSE™	ILO (40 kHz), CAPSENSE™ internal clock sources called MSC_ILO (32 kHz), MSC_IMO (48 MHz)	Interrupt	Deep-Sleep regulator

In addition to the wakeup sources mentioned in Table 10-1, external reset (XRES) and brownout reset bring the device to Active mode from any power mode.

10.1 Active mode

Active mode is the primary power mode of the PSoC™ device. This mode provides the option to use every possible subsystem/peripheral in the device. In this mode, the CPU is running and all the peripherals are powered. The firmware may be configured to disable specific peripherals that are not in use, to reduce power consumption.

10.2 Sleep mode

This is a CPU-centric power mode. In this mode, the Cortex®-M0+ CPU enters Sleep mode and its clock is disabled. It is a mode that the device should come to very often or as soon as the CPU is idle, to accomplish low power consumption. It is identical to Active mode from a peripheral point of view. Any enabled interrupt can cause wakeup from Sleep mode.

10.3 Deep Sleep mode

In Deep Sleep mode, the CPU, SRAM, and high-speed logic are in retention. The high-frequency clocks, including HFCLK and SYSCLK, are disabled. Optionally, the internal low-frequency oscillator (ILO, 40 kHz) remains on and low-frequency peripherals continue to operate. Digital peripherals that do not need a clock or receive a clock from their external interface (for example, I^2C slave) continue to operate. Interrupts from low-speed, asynchronous or low-power analog peripherals can cause a wakeup from Deep-Sleep mode.

This power mode is called wake-on-touch (WoT) mode. In this power mode, the MSCLP block operates using an internal clock source. The MSCLP block does the required initialization/setup, performs the frame scan and post-processes the scan results. When a touch is detected, the MSCLP block generates an interrupt to wake the CPU. This method saves power by not involving the CPU to enter the active or sleep state during scans. For details on power consumption, refer to the device datasheet. The available wakeup sources are listed in **Table 10-3**.

10.4 Power mode summary

Table 10-2 illustrates the peripherals available in each low-power mode; **Table 10-3** illustrates the wakeup sources available in each power mode.

Table 10-2. Available peripherals

Peripheral	Active	Sleep	Deep-Sleep
CPU	Available	Retention[a)]	Retention
SRAM	Available	Retention	Retention
High-speed peripherals	Available	Available	Retention
Low-speed peripherals	Available	Available	Available (optional)
Internal main oscillator (IMO)	Available	Available	Not Available
Internal low-speed oscillator (ILO, 40 kHz)	Available	Available	Available (optional)
Asynchronous peripheral (peripherals that do not run on internal clock)	Available	Available	Available
Power-on-reset, Brownout detection	Available	Available	Available
Analog mux bus connection	Available	Available	Available
GPIO output state	Available	Available	Available
CAPSENSE™	Available	Available	Available[b)]

a) The configuration and state of the peripheral is retained. Peripheral continues its operation when the device enters Active mode.

b) Only selected sensors can do scan operation in the Deep Sleep mode.

Table 10-3. Wakeup sources

Power mode	Wakeup source	Wakeup action
Sleep	Any enabled interrupt source	Interrupt
	Any reset source	Reset
Deep-Sleep	GPIO interrupt	Interrupt
	I2C address match	Interrupt
	Watchdog timer	Interrupt/Reset
	CAPSENSE™	Interrupt

Note: In addition to the wakeup sources mentioned in ***Table 10-3****, external reset (XRES) and brownout reset bring the device to Active mode from any power mode. XRES and brownout trigger a full system restart. All the states including frozen GPIOs are lost. In this case, the cause of wakeup is not readable after the device restarts.*

10.5 Low-power mode entry and exit

A Wait For Interrupt (WFI) instruction from the Cortex®-M0+ (CM0+) triggers the transitions into Sleep and Deep-Sleep mode. The Cortex®-M0+ can delay the transition into a low-power mode until the lowest priority ISR is exited (if the SLEEPONEXIT bit in the CM0 System Control Register is set).

The transition to Sleep and Deep-Sleep modes are controlled by the flags SLEEPDEEP in the CM0P System Control Register (CM0P_SCR).

- Sleep is entered when the WFI instruction is executed, SLEEPDEEP = 0.
- Deep-Sleep is entered when the WFI instruction is executed, SLEEPDEEP = 1.

The LPM READY bit in the PWR_CONTROL register shows the status of Deep-Sleep regulator. If the firmware tries to enter Deep-Sleep mode before the regulators are ready, then PSoC™ 4 goes to Sleep mode first, and when the regulators are ready, the device enters Deep-Sleep mode. This operation is automatically done in hardware.

In Sleep and Deep-Sleep modes, a selection of peripherals are available (see **Table 10-3**), and firmware can either enable or disable their associated interrupts. Enabled interrupts can cause wakeup from low-power mode to Active mode. Additionally, any RESET returns the system to Active mode. See the **"Interrupts"** on page 30 and the **"Reset system"** on page 76 for details.

10.6 Register list

Table 10-4. Power mode register list

Register name	Description
CM0P_SCR	System Control - Sets or returns system control data.
PWR_CONTROL	Power Mode Control - Controls the device power mode options and allows observation of current state.

11 Chip operational modes

PSoC™ 4 is capable of executing firmware in four different modes. These modes dictate execution from different locations in flash and ROM, with different levels of hardware privileges. Only three of these modes are used in end-applications; debug mode is used exclusively to debug designs during firmware development.

PSoC™ 4 operational modes are:

- Boot
- User
- Privileged
- Debug

11.1 Boot

Boot mode is an operational mode where the device is configured by instructions hard-coded in the device SROM. This mode is entered after the end of a reset, provided no debug-acquire sequence is received by the device. Boot mode is a privileged mode; interrupts are disabled in this mode so that the boot firmware can set up the device for operation without being interrupted. During boot mode, hardware trim settings are loaded from flash to guarantee proper operation during power-up. When boot concludes, the device enters user mode and code execution from flash begins. This code in flash may include automatically generated instructions from the ModusToolbox™ IDE that will further configure the device.

11.2 User

User mode is an operational mode where normal user firmware from flash is executed. User mode cannot execute code from SROM. Firmware execution in this mode includes the automatically generated firmware by the ModusToolbox™ and the firmware written by the user. The automatically generated firmware can govern both the firmware startup and portions of normal operation. The boot process transfers control to this mode after it has completed its tasks.

11.3 Privileged

Privileged mode is an operational mode, which allows execution of special subroutines that are stored in the device ROM. These subroutines cannot be modified by the user and are used to execute proprietary code that is not meant to be interrupted or observed. Debugging is not allowed in privileged mode.

The CPU can transition to privileged mode through the execution of a system call. For more information on how to perform a system call, see **“Performing a system call”** on page 194. Exit from this mode returns the device to user mode.

11.4 Debug

Debug mode is an operational mode that allows observation of the PSoC™ 4 operational parameters. This mode is used to debug the firmware during development. The debug mode is entered when an SWD debugger connects to the device during the acquire time window, which occurs during the device reset. Debug mode allows IDEs such as ModusToolbox™ to debug the firmware. Debug mode is only available on devices in open mode (one of the four protection modes). For more details on the debug interface, see the **“Program and debug interface”** on page 183. For more details on protection modes, see the **“Device security”** on page 41.

12 Watchdog timer

The watchdog timer (WDT) is used to automatically reset the device in the event of an unexpected firmware execution path or a brownout that compromises the CPU functionality. The WDT runs from ILO. The timer interrupt must be serviced periodically in firmware to avoid a reset. Otherwise, the timer will elapse and generate a device reset. The WDT can be used as an interrupt source or a wakeup source in low-power modes.

12.1 Features

The WDT has these features:

- System reset generation after a configurable interval
- Periodic interrupt/wake up generation in Active, Sleep, and Deep-Sleep power modes
- Features a 16-bit free-running counter

12.2 Block diagram

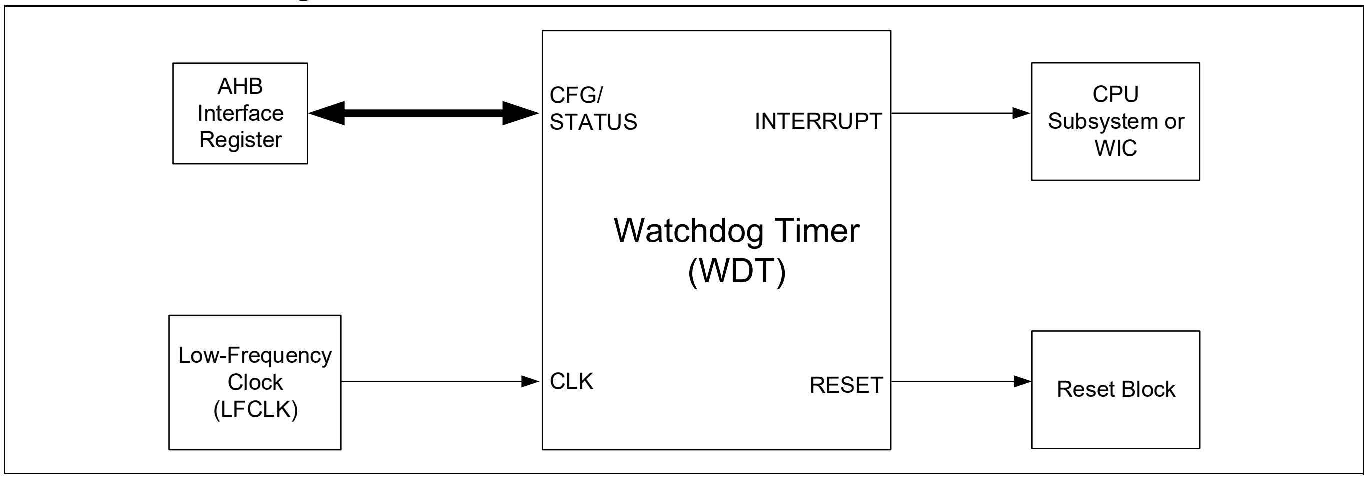


Figure 12-1. Watchdog timer block diagram

12.3 How it works

The WDT asserts a system reset to the device on the third WDT match event, unless it is periodically serviced in firmware. The WDT interrupt has a programmable period of up to 1638.4 ms. The WDT is a free-running wraparound up-counter with a maximum of 16-bit resolution. The resolution is configurable as explained later in this section.

The WDT_COUNTER register provides the count value of the WDT. The WDT generates an interrupt when the count value in WDT_COUNTER equals the match value stored in the WDT_MATCH register, but it does not reset the count to '0'. Instead, the WDT keeps counting until it overflows (after 0xFFFF when the resolution is set to 16 bits) and rolls back to 0. When the count value again reaches the match value, another interrupt is generated. Note that the match count can be changed when the counter is running.

A bit named WDT_MATCH in the SRSS_INTR register is set whenever the WDT interrupt occurs. This interrupt must be cleared by writing a '1' to the WDT_MATCH bit in SRSS_INTR to reset the watchdog. If the firmware does not reset the WDT for two consecutive interrupts, the third match event will generate a system reset.

The IGNORE_BITS in the WDT_MATCH register can be used to reduce the entire WDT counter period. The ignore bits can specify the number of MSBs that need to be discarded. For example, if the IGNORE_BITS value is 3, then the WDT counter becomes a 13-bit counter. For details, see the WDT_COUNTER, WDT_MATCH, and SRSS_INTR registers in the PSoC™ 4000T MCU registers TRM.

When the WDT is used to protect against system crashes, clearing the WDT interrupt bit to reset the watchdog must be done from 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.

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.
- Reset 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 SRSS_INTR register.
- It is not recommended to reset watchdog in the WDT interrupt service routine (ISR), if WDT is being used as a reset source to protect the system against crashes. Hence, it is not recommended to use WDT reset feature and ISR together.

Follow these steps to use WDT as a periodic interrupt generator:

1. Write the desired IGNORE_BITS in the WDT_MATCH register to set the counter resolution.
2. Write the desired match value to the WDT_MATCH register.
3. Clear the WDT_MATCH bit in SRSS_INTR to clear any pending WDT interrupt.
4. Enable the WDT interrupt by setting the WDT_MATCH bit in SRSS_INTR_MASK
5. Enable global WDT interrupt in the CM0_ISER register (see the **"Interrupts"** on page 30 for details).
6. In the ISR, clear the WDT interrupt and add the desired match value to the existing match value. By doing so, another periodic interrupt will be generated when the counter reaches the new match value.

For more details on interrupts, see the **"Interrupts"** on page 30.

12.3.1 Enabling and disabling WDT

The watchdog counter is a free-running counter that cannot be disabled. However, it is possible to disable the watchdog reset by writing a key '0xACED8865' to the WDT_DISABLE_KEY register. Writing any other value to this register will enable the watchdog reset. If the watchdog system reset is disabled, the firmware does not have to periodically reset the watchdog to avoid a system reset. The watchdog counter can still be used as an interrupt source or wakeup source. The only way to stop the counter is to disable the ILO by clearing the ENABLE bit in the CLK_ILO_CONFIG register. The watchdog reset must be disabled before disabling the ILO. Otherwise, any register write to disable the ILO will be ignored. Enabling the watchdog reset will automatically enable the ILO.

Note Disabling the WDT reset is not recommended if:

- Protection is required against firmware crashes
- The power supply can produce sudden brownout events that may compromise the CPU functionality

12.3.2 WDT interrupts and low-power modes

The watchdog counter can send interrupt requests to the CPU in Active power mode and to the WakeUp Interrupt Controller (WIC) in Sleep and Deep-Sleep power modes. It works as follows:

- **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. The interrupt must be cleared after entering the ISR in firmware.
- **Sleep or Deep-Sleep mode:** In this mode, the CPU subsystem is powered down. Therefore, the interrupt request from the WDT is directly sent to the WIC, which will then wake up the CPU. The CPU acknowledges the interrupt request and executes the ISR. The interrupt must be cleared after entering the ISR in firmware.

For more details on device power modes, see the **"Power modes"** on page 65.

12.3.3 WDT reset mode

The RESET_WDT bit in the RES_CAUSE register indicates the reset generated by the WDT. This bit remains set until cleared or until a power-on reset (POR), brownout reset (BOD), or external reset (XRES) occurs. All other resets leave this bit untouched. For more details, see the **"Reset system"** on page 76.

12.4 Register list

Table 12-1 provides the register control details.

Table 12-1. WDT registers

Register name	Description
WDT_DISABLE_KEY	Disables the WDT when 0XACED8865 is written; for any other value WDT works normally.
WDT_COUNTER	Provides the count value of the WDT.
WDT_MATCH	Holds the match value of the WDT.
SRSS_INTR	Services the WDT to avoid reset.
SRSS_INTR_MASK	Controls forwarding of the interrupt to CPU.

13 Trigger multiplexer block

Select peripherals in the PSoC™ 4 MCU are 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.

13.1 Features

The Trigger Multiplex block has these features:

- Ability to connect trigger signals from one peripheral to another
- Supports a software trigger, which can trigger signals in the block
- Supports multiplexing of triggers between peripherals

13.2 Architecture

The trigger signals in the PSoC™ 4 MCU are digital signals generated by peripheral blocks to denote a state such as TCPWM overflow, 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 chaining two TCPWMs together in order to make a 32-bit counter instead of a 16-bit counter. This can be done by using a counter overflow trigger to then trigger a second TCPWM's count input. This can be seen in **Figure 13-1**.

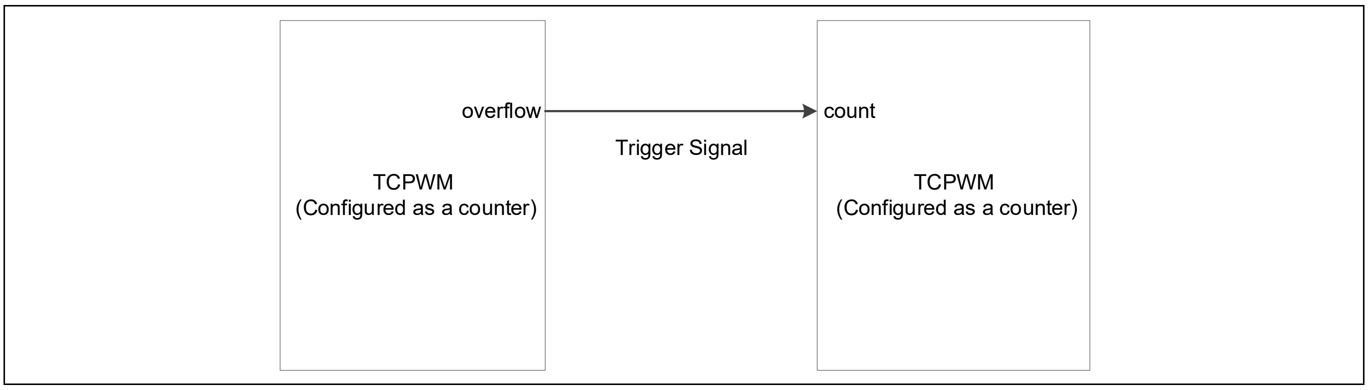


Figure 13-1. Trigger signal example

To support trigger routing the PSoC™ 4 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 a trigger signal emanating out of a peripheral block in the PSoC™ 4 MCU and route it to a different peripheral to initiate or affect an operation at the destination peripheral block. There are two types of triggers, level sensitive triggers and rising edge triggers. Rising edge triggers should remain '1' for at least 2 "clk_sys" cycles.

13.2.1 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. This can be seen in **Figure 13-3**. The multiplexers are grouped into a trigger group. 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 13-2**.

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.

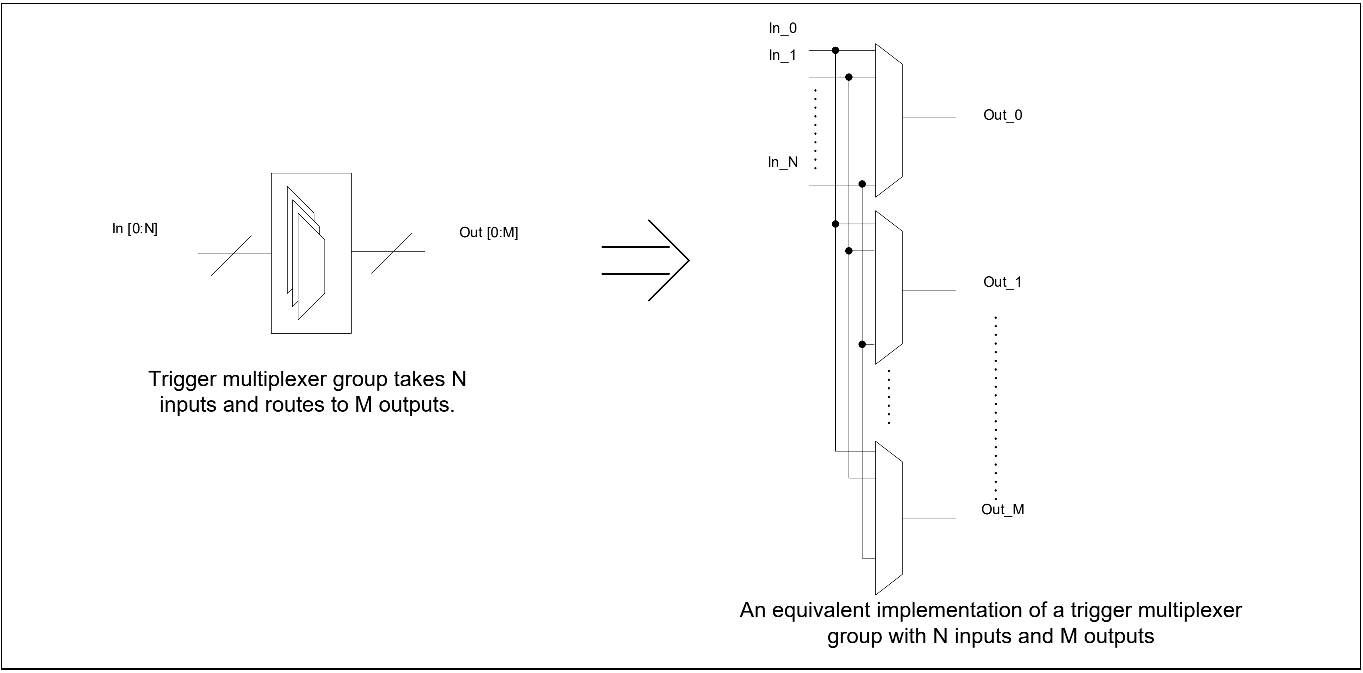

In [0:N]
Out [0:M]
Trigger multiplexer group takes N inputs and routes to M outputs.
In_0
In_1
In_N
Out_0
Out_1
Out_M
An equivalent implementation of a trigger multiplexer group with N inputs and M outputs

Figure 13-2. Trigger multiplexer groups

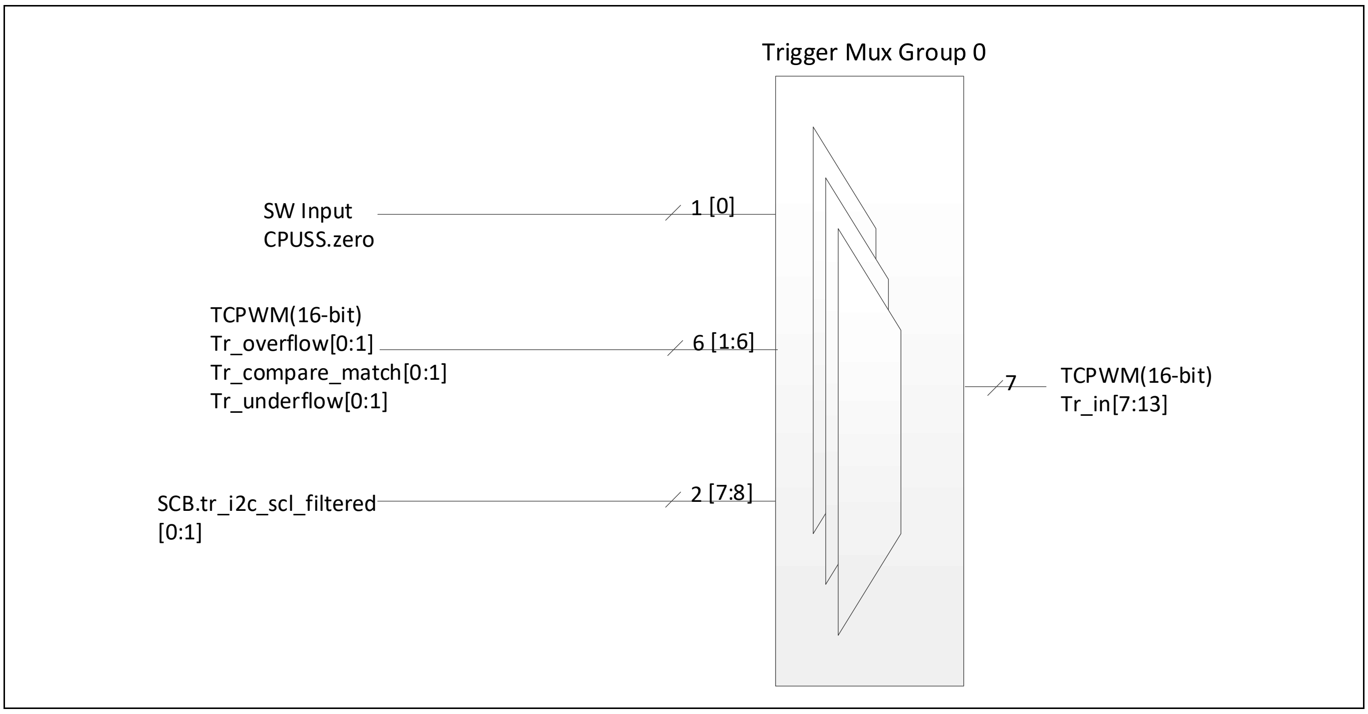


Figure 13-3. PSoC™ 4000T MCU trigger multiplexer block architecture

13.2.2 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_CTL register. This register allows you to trigger the corresponding signal for a number of peripheral clock cycles. The PERI_TR_CTL[TR_GROUP] bitfield selects the trigger group of the signal being activated. The PERI_TR_CTL[TR_OUT] bitfield determines whether the trigger signal is in output or input of the multiplexer. PERI_TR_CTL[TR_SEL] selects the specific line in the trigger group. The PERI_TR_CTL[TR_COUNT] bitfield sets up the number of peripheral clocks the trigger will be activated. The PERI_TR_CTL[TR_ACT] 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_CTL[TR_COUNT].

13.3 Register list

Table 13-1. Register list

Register name	Description
PERI_TR_CTL	Trigger control register. The control enables software activation of a specific input trigger or output trigger of the trigger multiplexer structure.
PERI_TR_GROUP[X]_TR_OUT_CTL[Y]	This register specifies the input trigger for a specific output trigger in a trigger group. Every trigger multiplexer group has a group of registers, the number of registers being equal to the output bus size from that multiplexer group. In the register format, X is the trigger group and Y is the output trigger line number from the multiplexer.

14 Reset system

PSoC™ 4 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. PSoC™ 4 also contains hardware to enable the detection of certain resets.

The reset system has these sources:

- Power-on reset (POR) to hold the device in reset while the power supply ramps up
- Brownout reset (BOD) to reset the device if the power supply falls below specifications during operation
- Watchdog reset (WRES) to reset the device if firmware execution fails to service the watchdog timer
- Software initiated reset (SRES) to reset the device on demand using firmware
- External reset (XRES) to reset the device using an external electrical signal
- Protection fault reset (PROT_FAULT) to reset the device if unauthorized operating conditions occur

14.1 Reset sources

The following sections provide a description of the reset sources available in PSoC™ 4.

14.1.1 Power-on reset

Power-on reset is provided for system reset at power-up. POR holds the device in reset until the supply voltage, V_{DDD}, is according to the datasheet specification. The POR activates automatically at power-up.

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.

14.1.2 Brownout reset

Brownout reset monitors the chip digital voltage supply V_{CCD} and generates a reset if V_{CCD} is below the minimum logic operating voltage specified in the device datasheet. BOD is available in all power modes.

14.1.3 Watchdog reset

Watchdog reset (WRES) detects errant code by causing a reset if the watchdog timer is not cleared within the user-specified time limit. This feature is enabled by default. It can be disabled by writing '0xACED8865' to the WDT_DISABLE_KEY register.

The RESET_WDT status bit of the RES_CAUSE register is set when a watchdog reset occurs. This bit remains set until cleared or until a POR, XRES, or BOD reset; for example, in the case of a device power cycle. All other resets leave this bit untouched. For more details, see the **"Watchdog timer"** on page 70.

14.1.4 Software initiated reset

Software initiated reset (SRES) is a mechanism that allows a software-driven reset. The Cortex®-M0+ application interrupt and reset control register (CM0P_AIRCR) forces a device reset when a '1' is written into the SYSRESETREQ bit. CM0P_AIRCR requires a value of 05FA written to the top two bytes for writes. Therefore, write 05FA0004 for the reset.

The RESET_SOFT status bit of the RES_CAUSE register is set when a software reset occurs. This bit remains set until cleared or until a POR, XRES, or BOD reset; for example, in the case of a device power cycle. All other resets leave this bit untouched.

14.1.5 External reset

External reset (XRES) is a user-supplied reset that causes immediate system reset when asserted. The XRES pin is **active low** – a high voltage on the pin has no effect and a low voltage causes a reset. The pin is pulled high inside the device. XRES is available as a dedicated pin in most of the devices. For detailed pinout, refer to the Pinout section of the device datasheet.

The XRES pin holds the device in reset while held active. When the pin is released, 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 device datasheet.

XRES 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.

14.1.6 Protection fault reset

Protection fault reset (PROT_FAULT) detects unauthorized protection violations and causes a device reset if they occur. One example of a protection fault is if a debug breakpoint is reached while executing privileged code. For details about privilege code, see **“Privileged”** on page 69.

The RESET_PROT_FAULT bit of the RES_CAUSE register is set when a protection fault occurs. This bit remains set until cleared or until a POR, XRES, or BOD reset; for example, in the case of a device power cycle. All other resets leave this bit untouched.

14.2 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 in the device primarily through the RES_CAUSE register. This register has specific status bits allocated for some of the reset sources. The RES_CAUSE register supports detection of watchdog reset, software reset, and protection fault reset. It does not record the occurrences of POR, BOD, or XRES. The bits are set on the occurrence of the corresponding reset and remain set after the reset, until cleared or a loss of retention, such as a POR reset, external reset, or brownout detect.

If the RES_CAUSE register cannot detect the cause of the reset, then it can be one of the non-recorded and non-retention resets: BOD, POR, XRES. These resets cannot be distinguished using on-chip resources.

14.3 Register list

Register name	Description
WDT_DISABLE_KEY	Disables the WDT when 0XACED8865 is written, for any other value WDT works normally
CM0P_AIRCR	Cortex®-M0+ Application Interrupt and Reset Control Register - This register allows initiation of software resets, among other Cortex®-M0+ functions.
RES_CAUSE	Reset Cause Register - This register captures the cause of recent resets.

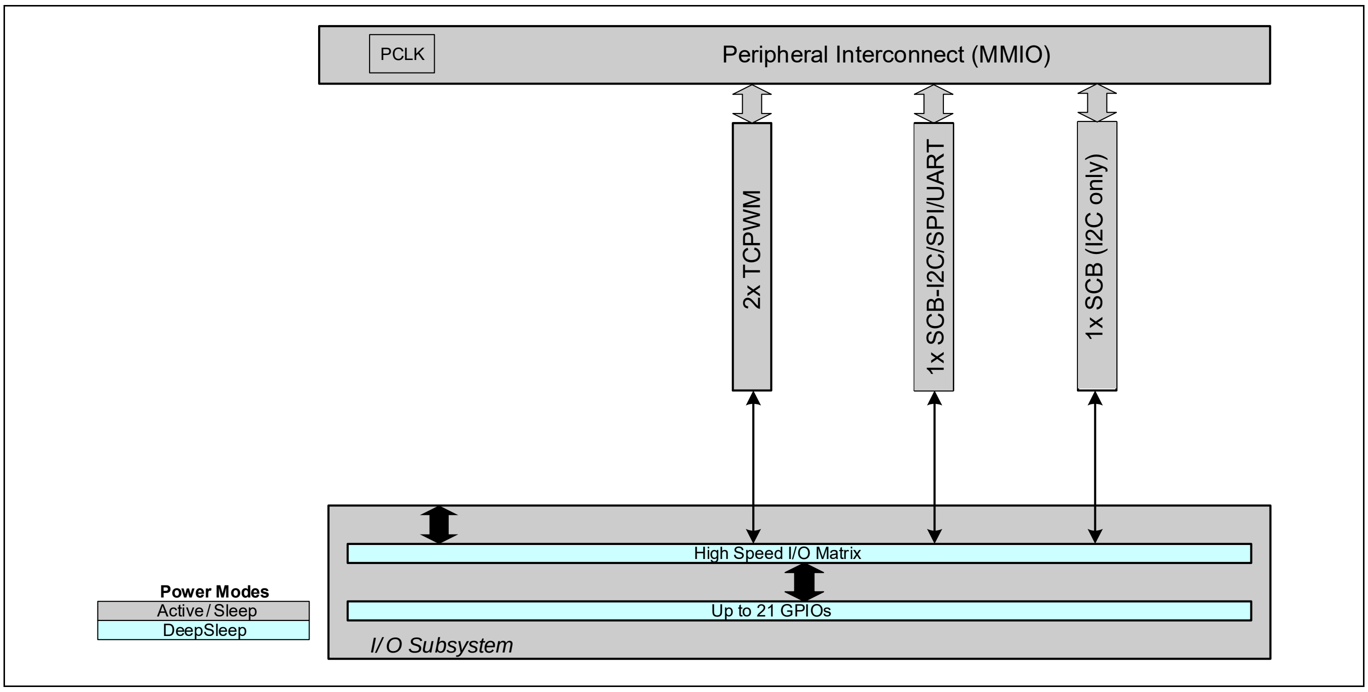
Section D: Digital system

This section encompasses the following chapters:

- **"Serial communications block (SCB)"** on page 79
- **"Timer, counter, and PWM (TCPWM)"** on page 140

Top level architecture

Digital system block diagram



15 Serial communications block (SCB)

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 maximum number of SCBs in the PSoC™ 4 MCU devices varies by part number. Refer to the **device datasheet** to determine the number of SCBs and the SCB pin locations.

15.1 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
- Multiple interrupt sources to indicate status of FIFOs and transfers
- SCBs are Deep Sleep capable and allow for operation without CPU intervention.
  - Deep Sleep only works for EZ mode for SPI and I2C configured as a slave.
  - During Deep Sleep the SCB clock is not available so an external clock must be used.
  - Deep Sleep wakeup when I2C or SPI are configured in FIFO mode.
- Two serial communication blocks (SCBs). SCB[0] programmable to be UART/SPI/I2C and SCB[1] is programmable to be I2C master/slave (for Host MCU interface).

15.2 Architecture

The operation modes supported by SCB are described in the following sections.

15.2.1 Buffer modes

Each SCB has 32 bytes of dedicated RAM for transmit and receive operation. The RAM can be configured in two different modes (FIFO and EZ). 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 both FIFO and EZ modes
- UART only uses FIFO mode

15.2.1.1 FIFO mode

In this mode the RAM is split into two 16-byte FIFOs, one for transmit (TX) and one for receive (RX). The FIFOs can be configured to be 8 bits × 16 elements or 16 bits × 8 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.

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 130.

15.2.1.2 EZ mode

In easy (EZ) mode the RAM is used as a single 32-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^{2}C slave.

EZ mode is available in Active, Sleep, and Deep Sleep power modes.

Note: This document discusses hardware implementation of the EZ mode.

15.2.2 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^{2}C master modes must use clk_scb.
- Only SPI slave and I^{2}C slave can use the clock from and external master.

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^{2}C 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^{2}C address matching) is internally (‘0’) or externally (‘1’) clocked.

Note:

- 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 mode is supported with externally clocked operation (EC_OP_MODE is ‘1’).

Table 15-1 provides an overview of the clocking and buffer modes supported for each communication mode.

Table 15-1. Clock mode compatibility

	Internally clocked (IC)		**Externally clocked (EC) (Deep Sleep SCB only)**	
	FIFO	**EZ**	**FIFO**	**EZ**
I^{2}C master	Yes	No	No	No
I^{2}C slave	Yes	Yes	No[a)]	Yes
I^{2}C master-slave	Yes	No	No	No
SPI master	Yes	No	No	No
SPI slave	Yes	Yes	No[b)]	Yes
UART transmitter	Yes	No	No	No
UART receiver	Yes	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.

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.

Table 15-2. 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

15.3 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.

15.3.1 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
- Supports FIFO Mode
- Supports EZ Mode (slave only)

15.3.2 General description

Figure 15-1 illustrates an example of SPI master with four slaves.

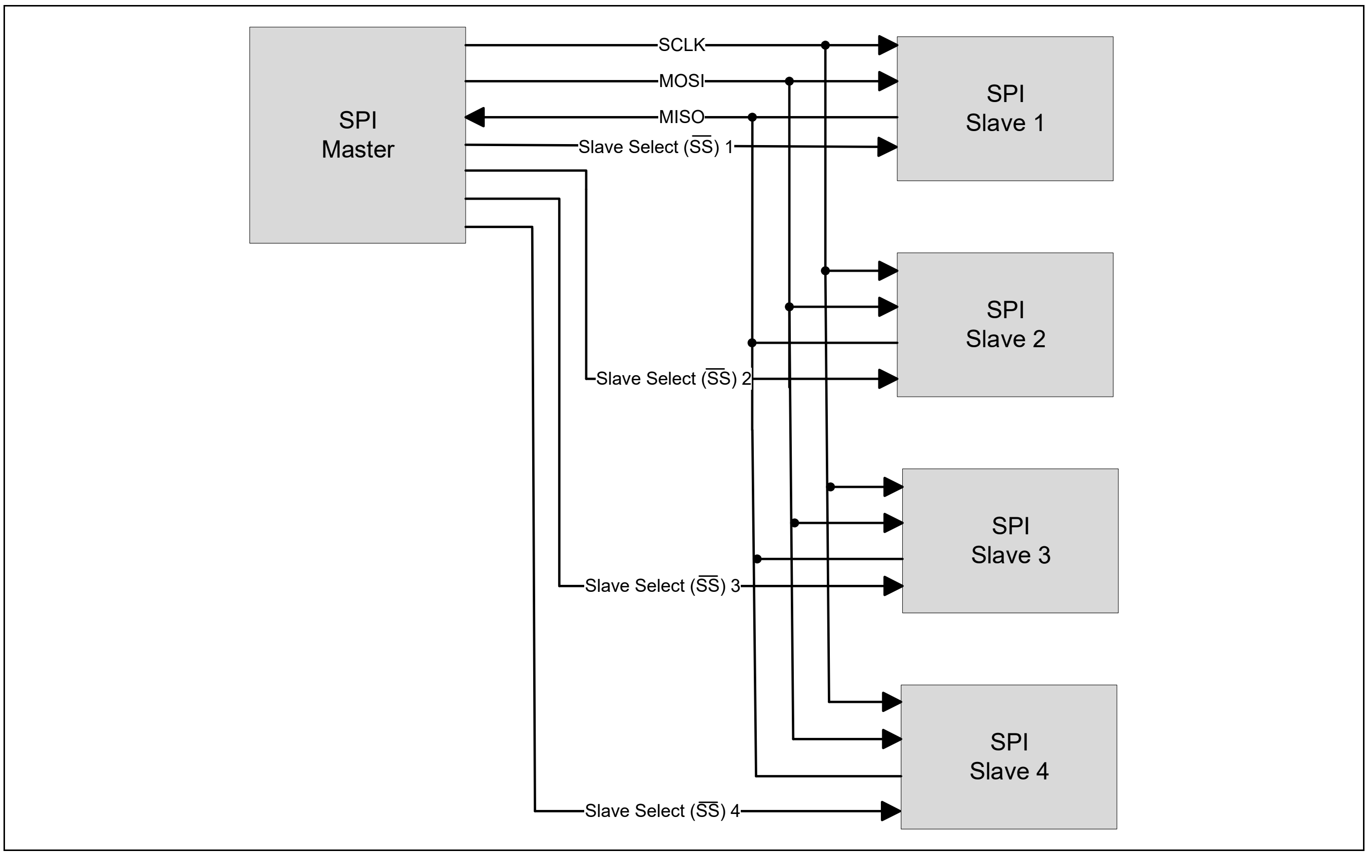

SPI
Master
SCLK
MOSI
MISO
Slave Select ($\overline{\text{SS}}$) 1
SPI
Slave 1
SPI
Slave 2
Slave Select ($\overline{\text{SS}}$) 2
SPI
Slave 3
Slave Select ($\overline{\text{SS}}$) 3
SPI
Slave 4
Slave Select ($\overline{\text{SS}}$) 4

Figure 15-1. 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 ($\overline{\text{SS}}$): 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 $\overline{\text{SS}}$ 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.

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 $\overline{\text{SS}}$ line.
- National Semiconductors SPI: A half-duplex variation of the original SPI protocol.

15.3.3 SPI modes of operation

15.3.3.1 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 15-2 illustrates driving and capturing of MOSI/MISO data as a function of CPOL and CPHA.

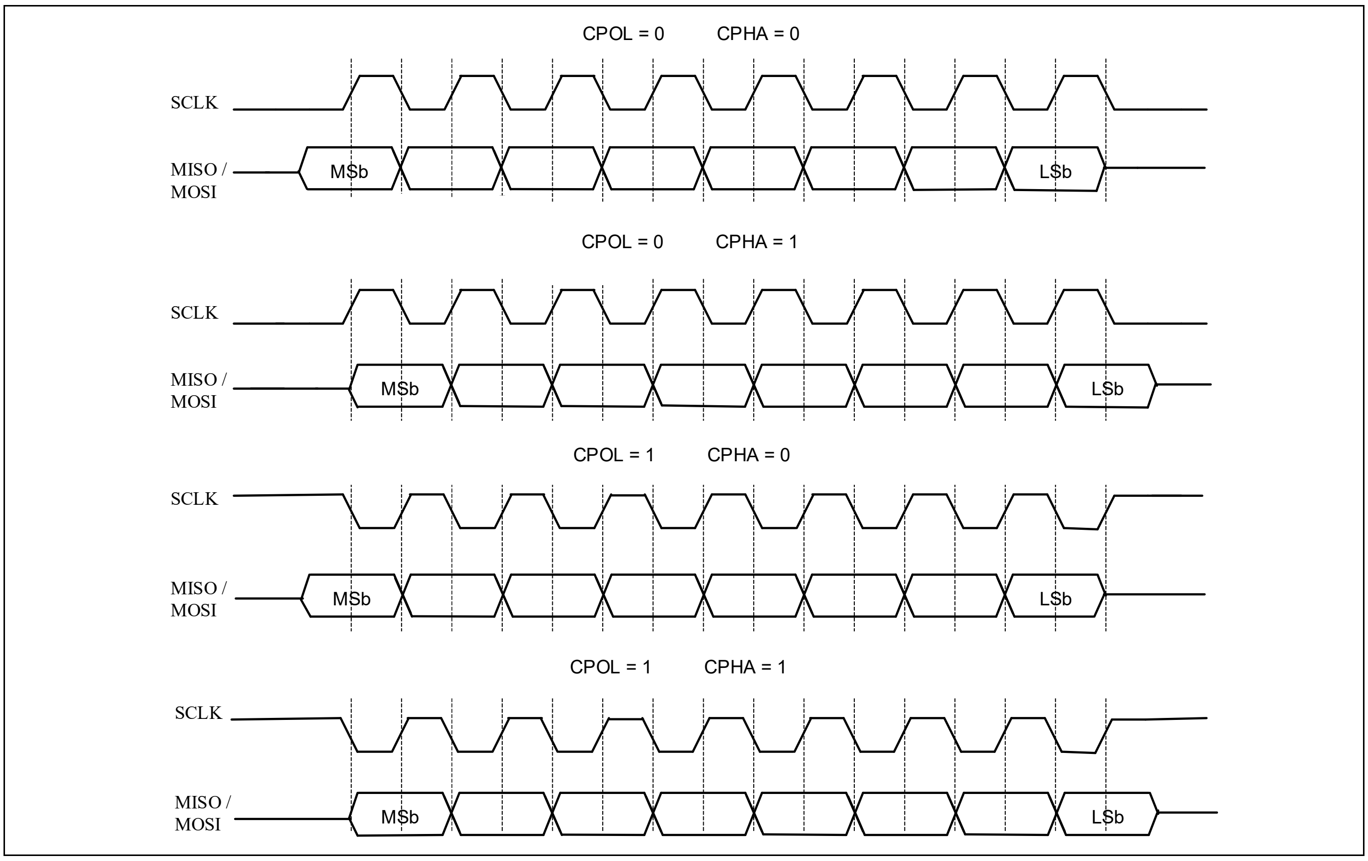


Figure 15-2. SPI Motorola, 4 modes

Figure 15-3 illustrates a single 8-bit data transfer and two successive 8-bit data transfers in mode 0 (CPOL is '0', CPHA is '0').

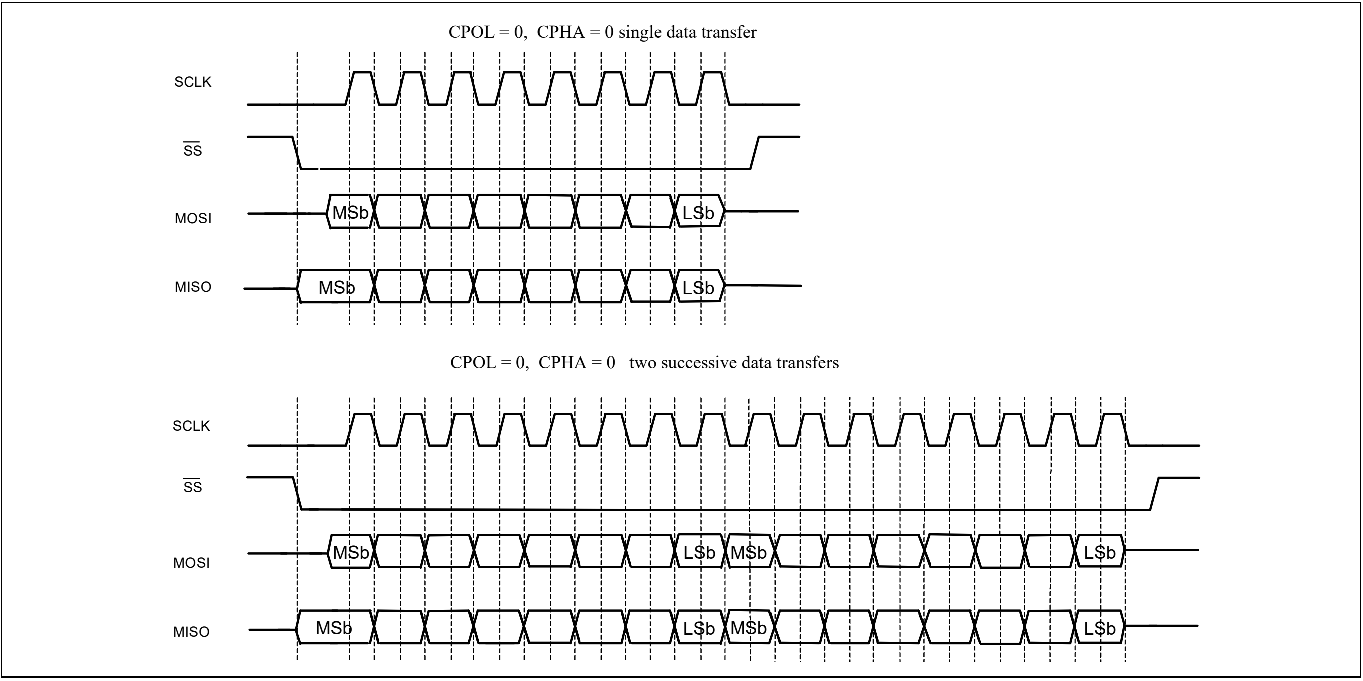
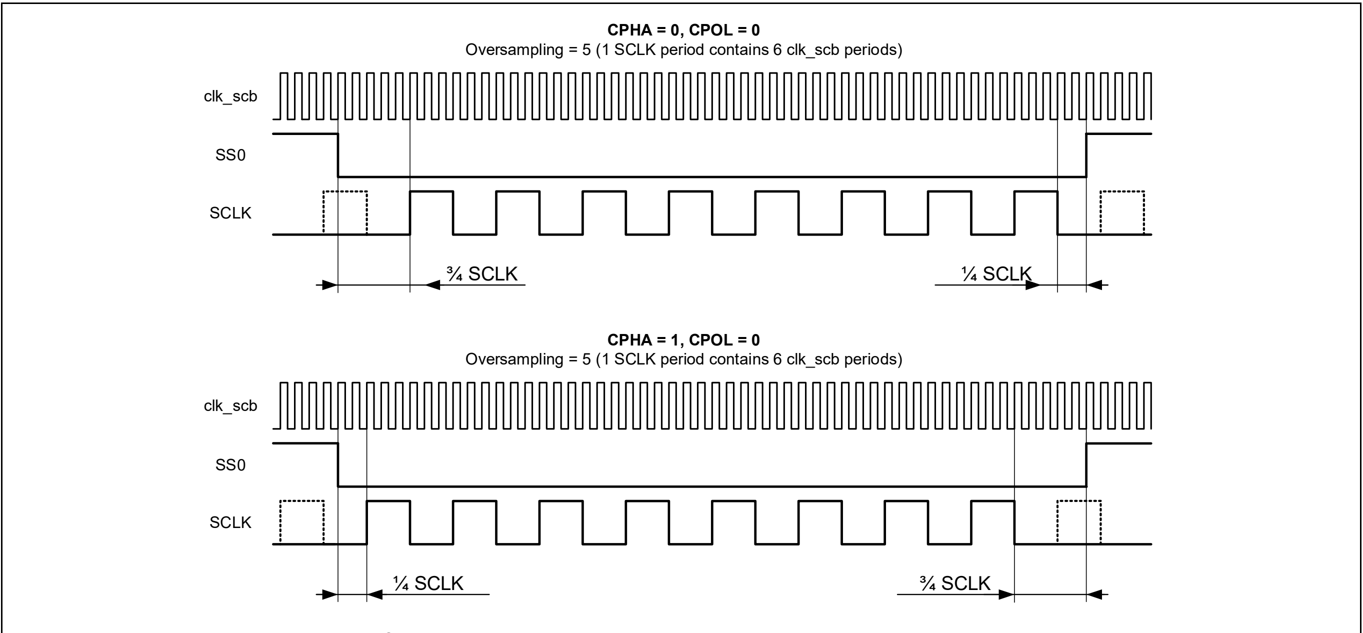

CPOL = 0, CPHA = 0 single data transfer
SCLK
SS
MOSI
MISO
MSb
LSb
CPOL = 0, CPHA = 0 two successive data transfers

Figure 15-3. SPI Motorola data transfer example


CPHA = 0, CPOL = 0
Oversampling = 5 (1 SCLK period contains 6 clk_scb periods)
clk_scb
SS0
SCLK
¾ SCLK
¼ SCLK
CPHA = 1, CPOL = 0
Oversampling = 5 (1 SCLK period contains 6 clk_scb periods)

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.

Figure 15-4. SELECT and SCLK timing correlation

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 95.

For more information on these registers, see the PSoC™ 4000T MCU registers TRM.

15.3.3.2 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 15-5 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.

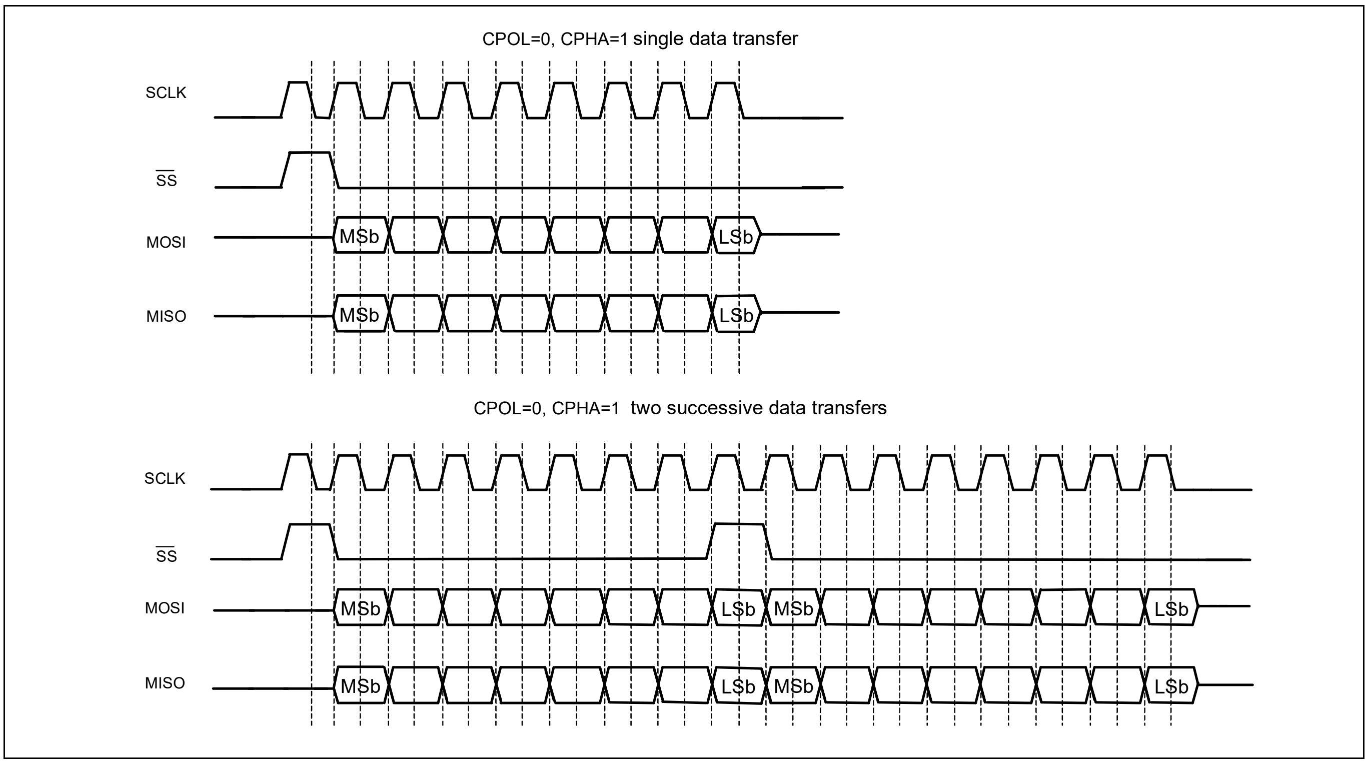

CPOL=0, CPHA=1 single data transfer
SCLK
SS
MOSI
MISO
MSb
LSb
CPOL=0, CPHA=1 two successive data transfers
SCLK
SS
MOSI
MISO
MSb
LSb

Figure 15-5. SPI TI data transfer example

Figure 15-6 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.

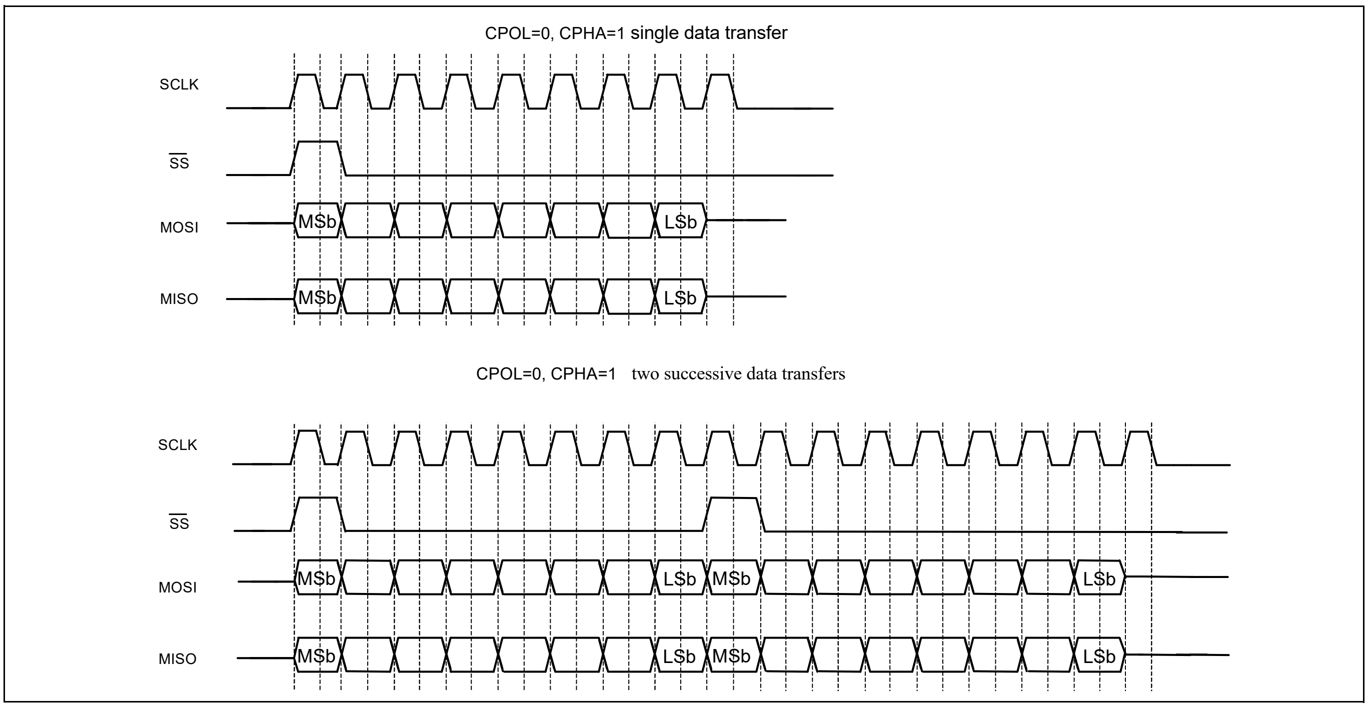

CPOL=0, CPHA=1 single data transfer
SCLK
SS
MOSI
MISO
MSb
LSb
CPOL=0, CPHA=1 two successive data transfers
SCLK
SS
MOSI
MISO
MSb
LSb

Figure 15-6. 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 2) of the SCB_SPI_CONTROL register to '1', and the CPOL (bit 3) of the same register to '0'.
5. Follow steps 2 to 4 mentioned in **"Enabling and initializing SPI"** on page 95.

For more information on these registers, see the PSoC™ 4000T MCU registers TRM.

15.3.3.3 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 only supports CPOL/CPHA mode 0.

Figure 15-7 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.

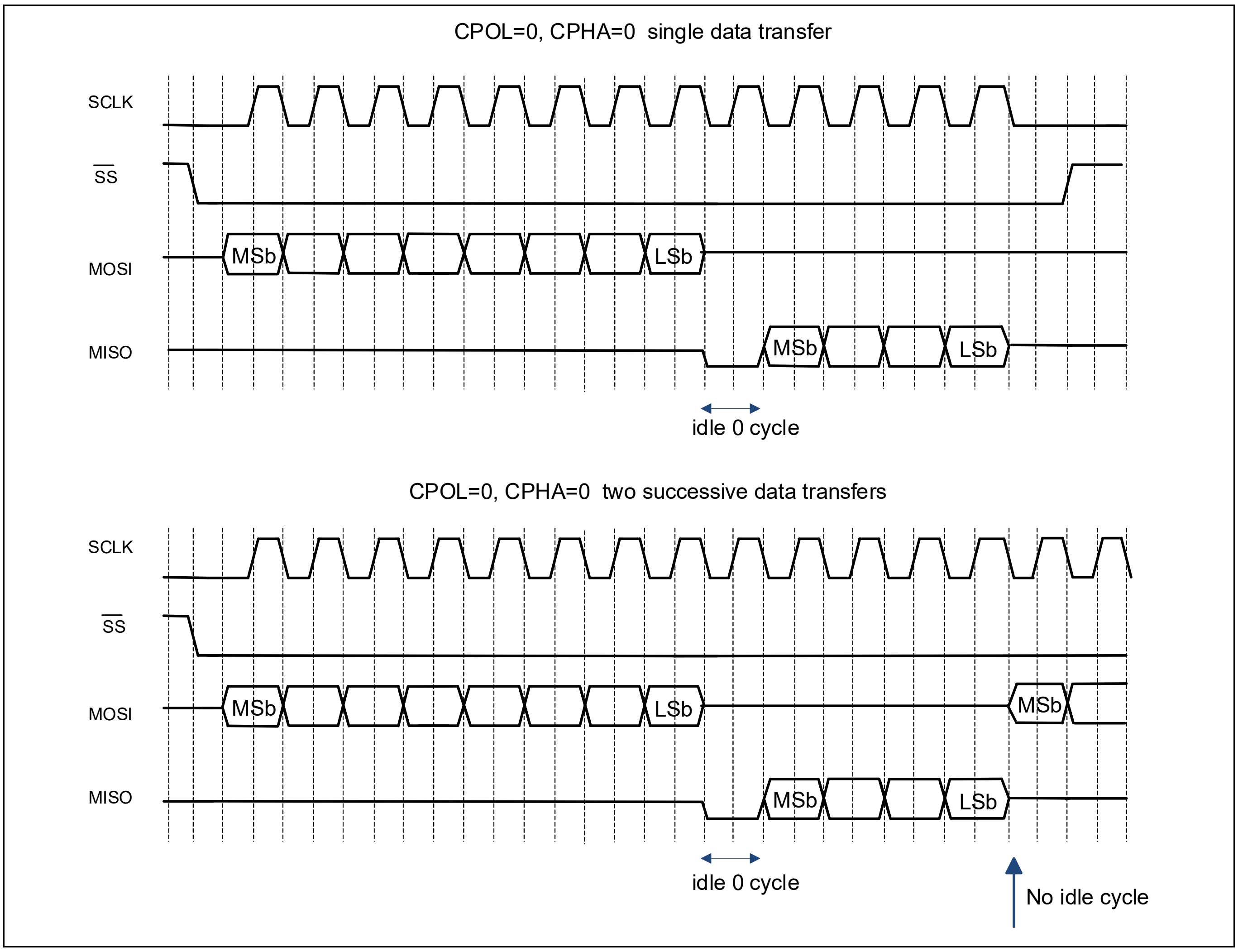


Figure 15-7. SPI NS data transfer example

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 95.

For more information on these registers, see the PSoC™ 4000T MCU registers TRM.

15.3.4 SPI buffer modes

SPI can operate in two different buffer modes – FIFO and EZ modes. The buffer is used in different ways in each of these modes. The following subsections explain each of these buffer modes in detail.

15.3.5 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 8 elements deep with 16-bit data elements or 16 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 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.

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 is done automatically by ModusToolbox™ Software with the provided API. This can be done manually by disabling the clock divider for the SCB clock. For more information on clock dividers, consult the **"Clocking system"** on page 54.

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"** on page 44 for more information on pin drive modes.

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, enable the clock to the SCB, this is done by enabling the clock divider. See **"Clocking system"** on page 54 for more information. At this point, the master can read valid data from the slave. Before that any data read by the master will be invalid.

15.3.5.1 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.

The EZSPI protocol defines a single memory buffer with an 5-bit EZ address that indexes the buffer (32-entry array of eight bit per entry) located on the slave device. The EZ address is used to address these 32 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 (32), 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 (32), 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 15-8 illustrates the write of EZ address, write to a memory array and read from a memory array operations in the EZSPI protocol.

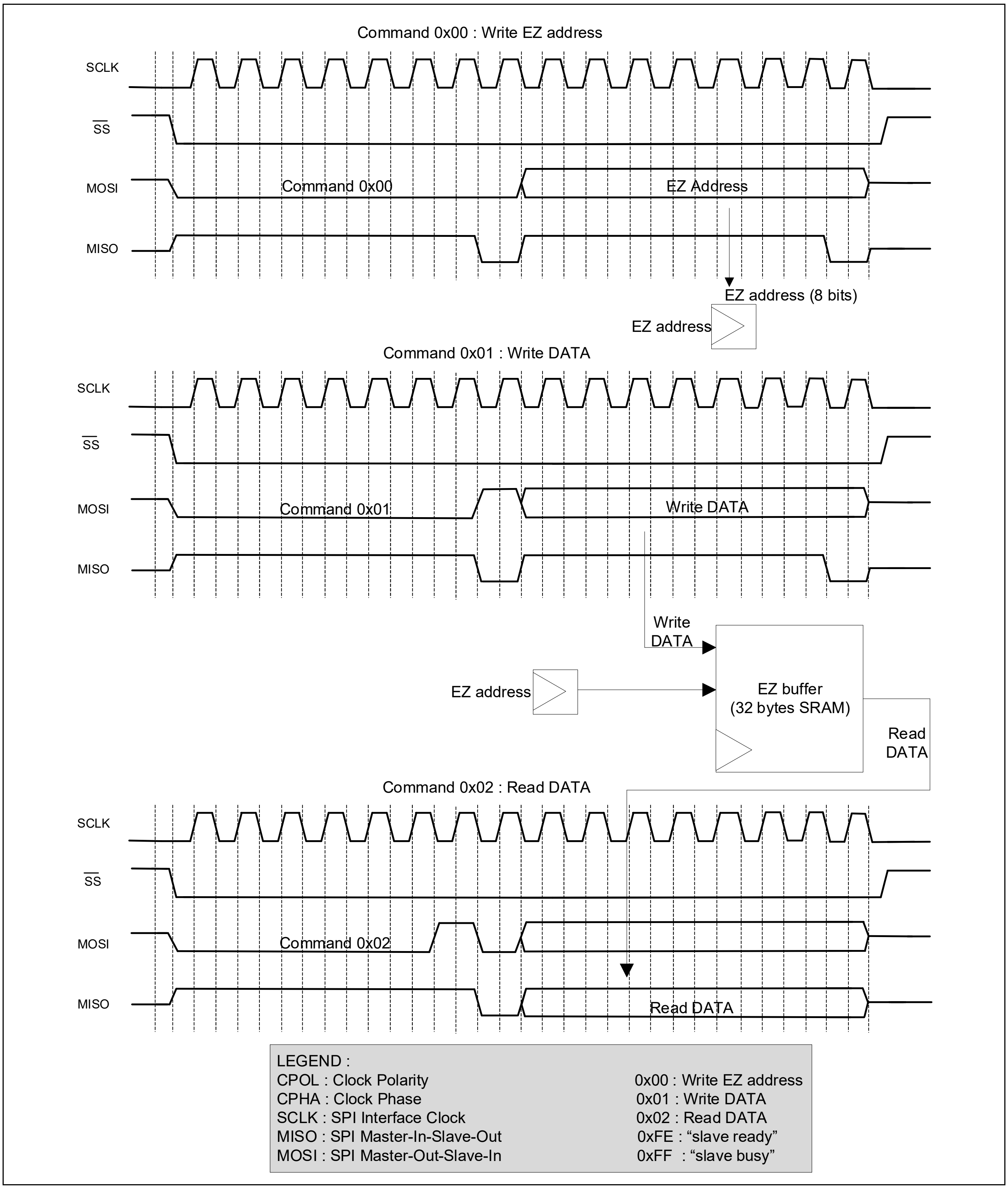


Figure 15-8. EZSPI example

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 85.
5. Follow steps 2 to 4 mentioned in **"Enabling and initializing SPI"** on page 95.

For more information on these registers, see the PSoC™ 4000T MCU 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 disabling the clock divider for the clock going to the SCB, see the **"Clocking system"** on page 54.

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, enable the clock to the SCB, this is done by enabling the clock divider. See the **"Clocking system"** on page 54 for more information. 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, enable the clock to the SCB, this is done by enabling the clock divider. See the **"Clocking system"** on page 54 for more information.

15.3.6 Clocking and oversampling

15.3.6.1 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).

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"** on page 54.

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

- 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 88 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.
- 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.

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 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™ 4 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 15-3. SPI modes compatibility

	Internally clocked (IC)		Externally clocked (EC)	
	FIFO	**EZ**	**FIFO**	**EZ**
SPI Master	Yes	No	No	No
SPI Slave	Yes	Yes	Yes[a)]	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.

15.3.6.2 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.

15.3.6.3 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).

In SPI master mode, the valid range for oversampling is 4 to 16. Hence, with a clock speed of 48 MHz, the maximum bit rate is 12 Mbps. However, if you consider the I/O cell and routing delays, the oversampling must be set between 6 and 16 for proper operation. Therefore, the maximum bit rate is 8 Mbps.

Note: *To achieve maximum possible bit rate, LATE_MISO_SAMPLE must be set to '1' in SPI master mode. This has a default value of '0'.*

$$\text{Bit Rate} = \text{clk\_scb/OVS} \tag{15.1}$$

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 **Equation (15.2)**.

$$\frac{1}{2}t_{SCLK} \geq t_{SCLK\_PCB\_D} + t_{DSO} + t_{SCLK\_PCB\_D} + t_{DSI} \tag{15.2}$$

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 t_{DSO}, It may have a different name; look for MISO output valid after SCLK edge. Most master datasheets will also list t_{DSI}, or master setup time. $t_{SCLK\_PCB\_D}$ and $t_{SCLK\_PCB\_D}$ 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. 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 15-9**.

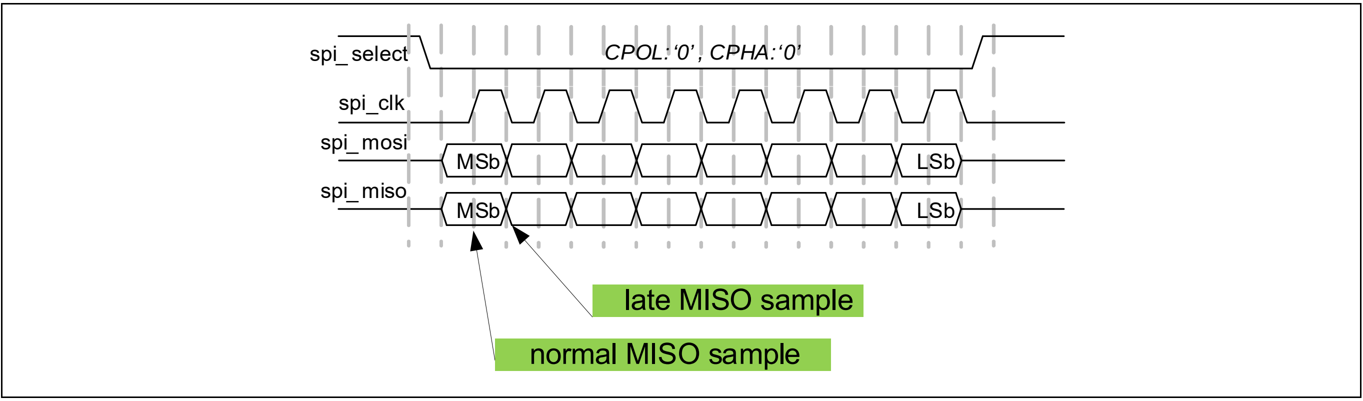


Figure 15-9. MISO sampling timing

This changes the equation to:

$$t_{SCLK} \geq t_{SCLK\_PCB\_D} + t_{DSO} + t_{SCLK\_PCB\_D} + t_{DSI} \quad (15.3)$$

Because late sample allows for better timing, leave it enabled all the time. The t_{DSI} specification in the PSoC™ 4 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 (15.2)** and **Equation (15.3)**. Late MISO sample is determined by the external master in this case, not by SPI_CTRL.LATE_MISO_SAMPLE.

For PSoC™ 4 MCUs, t_{DSO} is given in the **device 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 (15.1)** can be re-arranged to $t_{CLK\_SCB} \leq ((t_{SCLK}) - 40 \text{ ns})/6$.

15.3.7 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 can be used with slave mode only.
2. Program the OVS field and configure clk_scb as appropriate. See the **"Clocking system"** on page 54 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.
   b) Specify whether MSb or LSb is the first bit to be transmitted/received. This should always be MSb first for EZSPI.
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).*

15.3.8 I/O pad connection

15.3.8.1 SPI master

Figure 15-10 and Table 15-4 list the use of the I/O pads for SPI master.

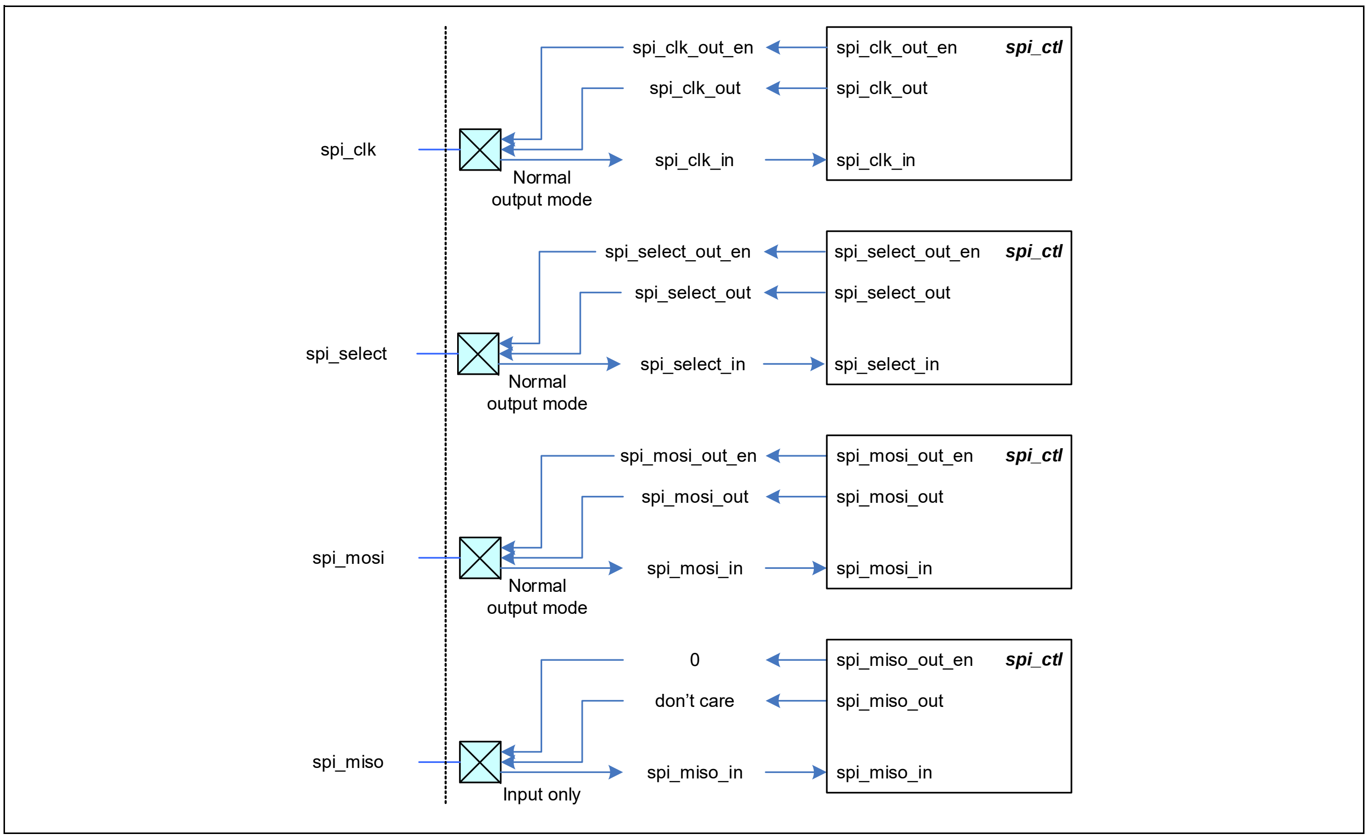


Figure 15-10. SPI master I/O pad connections

Table 15-4. 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

15.3.8.2 SPI slave

Figure 15-11 and **Table 15-5** list the use of I/O pads for SPI Slave.

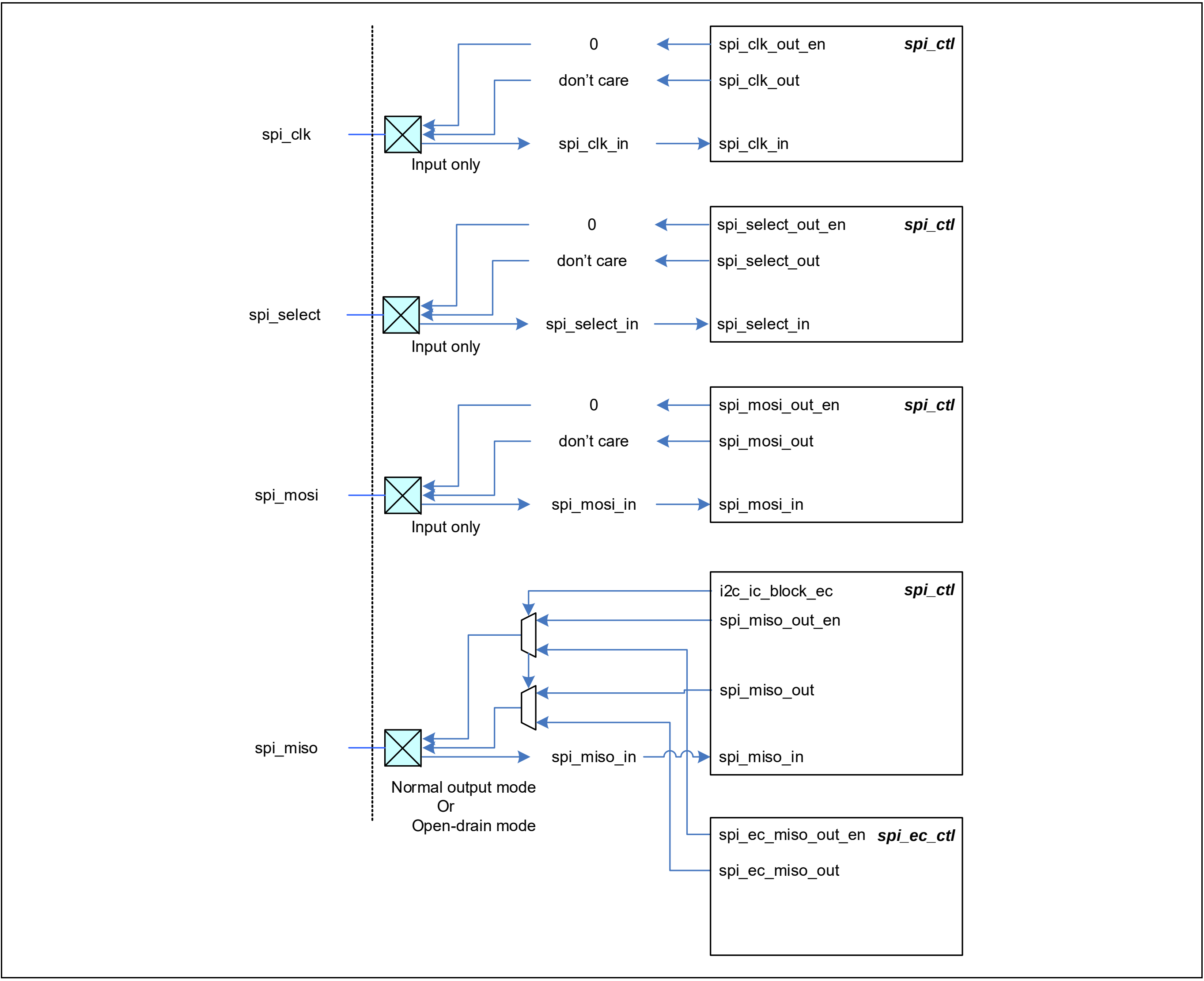


Figure 15-11. SPI slave I/O pad connections

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 15-5. 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

15.3.8.3 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.

15.3.9 SPI registers

The SPI interface is controlled using a set of 32-bit control and status registers listed in **Table 15-6**. For more information on these registers, see the PSoC™ 4000T MCU registers TRM.

Table 15-6. SPI registers

Register name	Operation
SCB_CTRL	Enables the SCB, selects the type of serial interface (SPI, UART, I^{2}C), and selects internally and externally clocked operation, and EZ and non-EZ modes of operation.
SCB_STATUS	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 $\overline{\text{SS}}$ 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_SILEN T	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	Holds the data in EZ memory location.

15.4 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.

15.4.1 Features

- Supports UART protocol
  - Standard UART
  - SmartCard (ISO7816) reader
  - IrDA
- Multi-processor mode
- 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 7
- Parity support (odd and even parity)
- Median filter on RX input
- Programmable oversampling
- Start skipping
- Hardware flow control

15.4.2 General description

Figure 15-12 illustrates a standard UART TX and RX.

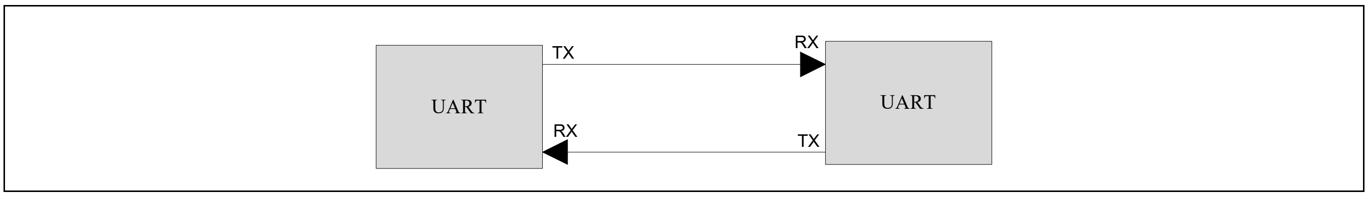


Figure 15-12. UART example

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 7. 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.*

15.4.3 UART modes of operation

15.4.3.1 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 15-13 illustrates the UART protocol.

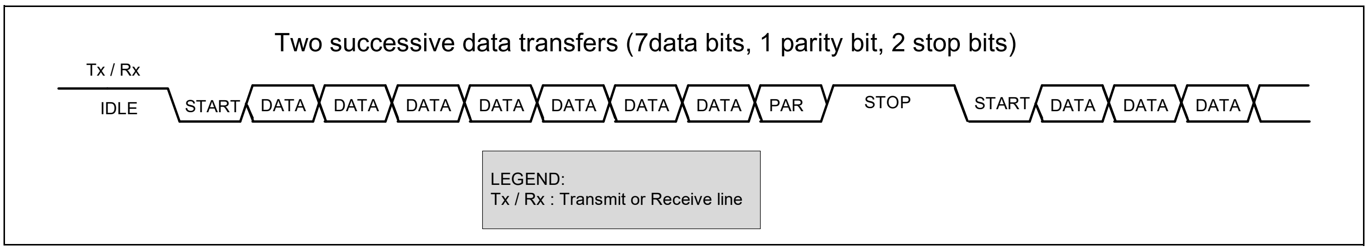

Two successive data transfers (7data bits, 1 parity bit, 2 stop bits)
Tx / Rx
IDLE
START
DATA
PAR
STOP
LEGEND:
Tx / Rx : Transmit or Receive line

Figure 15-13. 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 15-14** illustrates this.

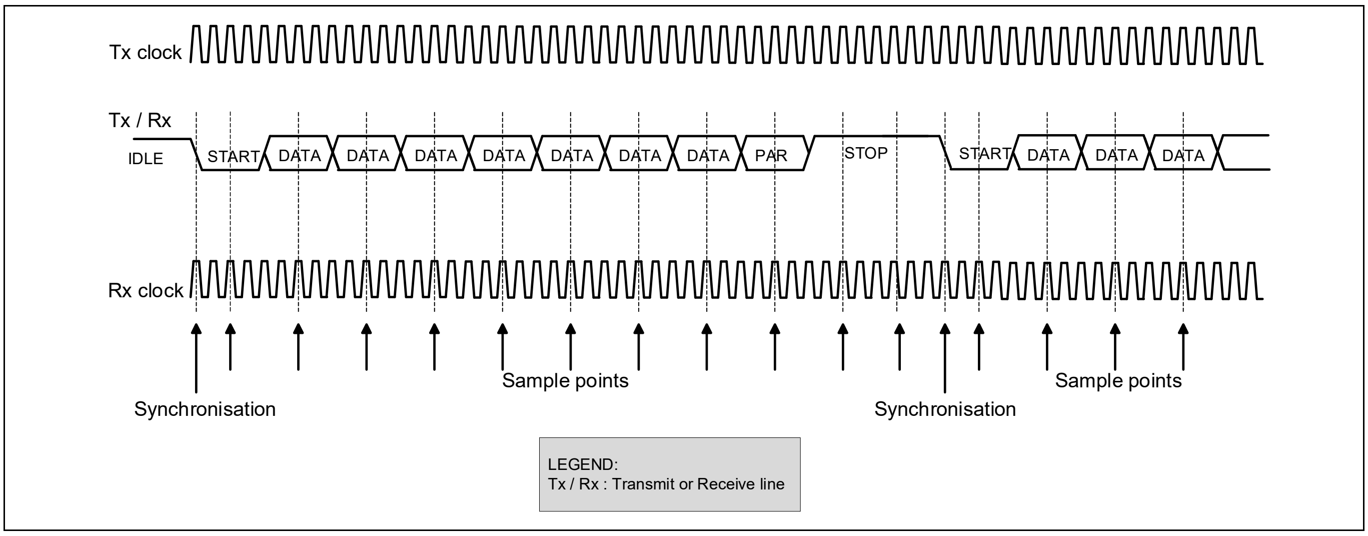


Figure 15-14. UART, standard protocol example (Single sample)

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 15-15** illustrates this.

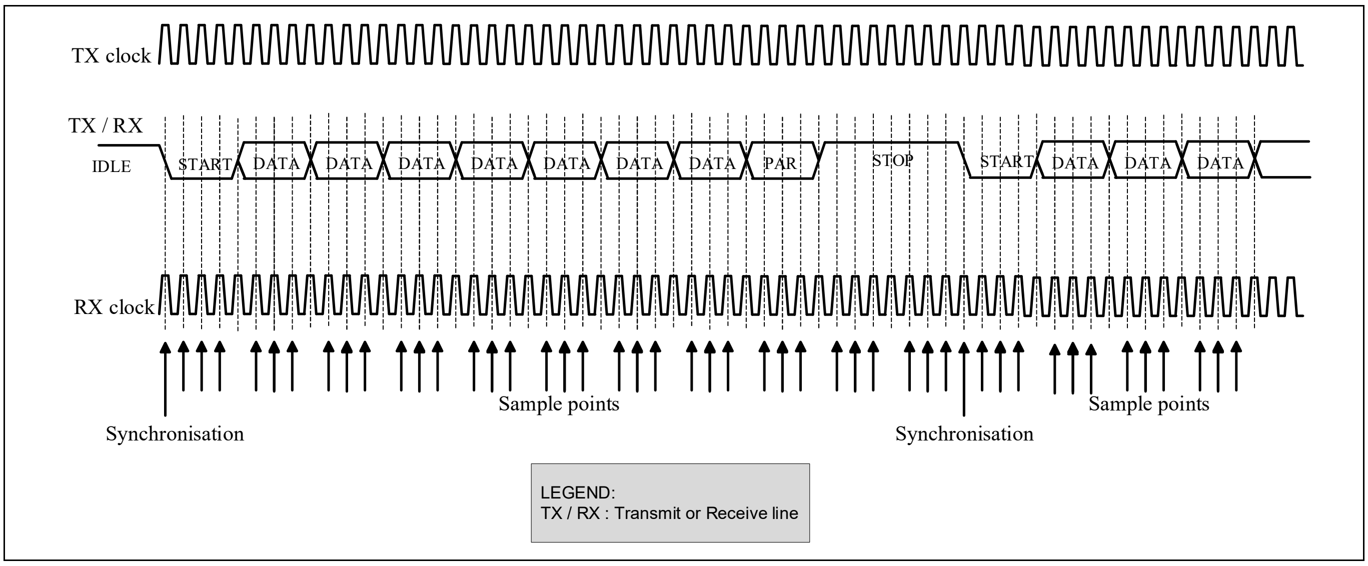


Figure 15-15. UART, standard protocol (Multiple samples)

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).

Figure 15-6 illustrates the parity functionality (8-bit data frame).

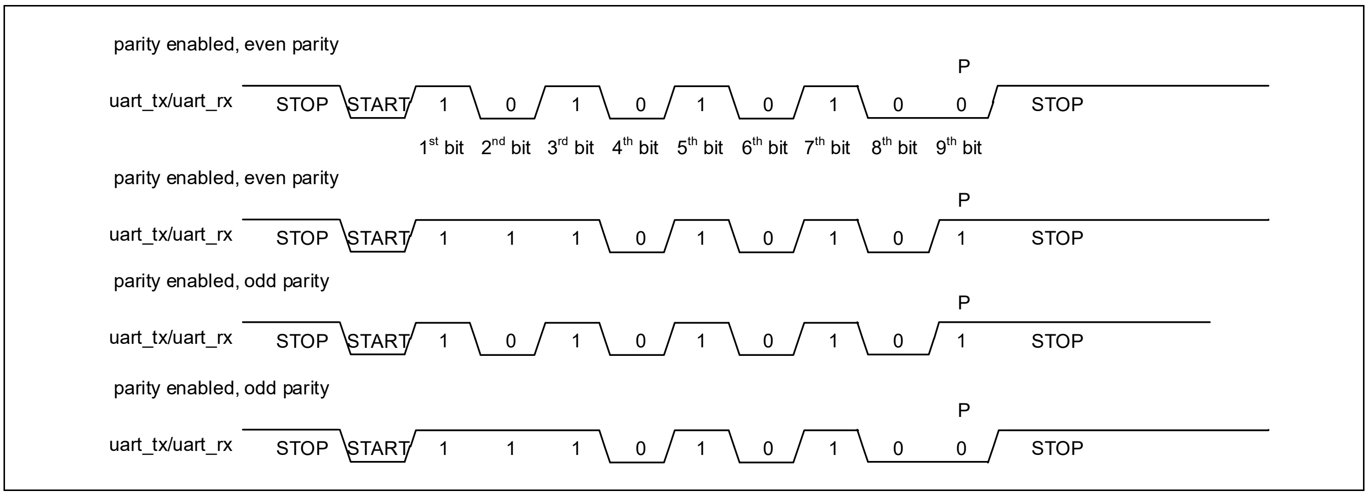


Figure 15-16. UART parity examples

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 15-17 illustrates the process.

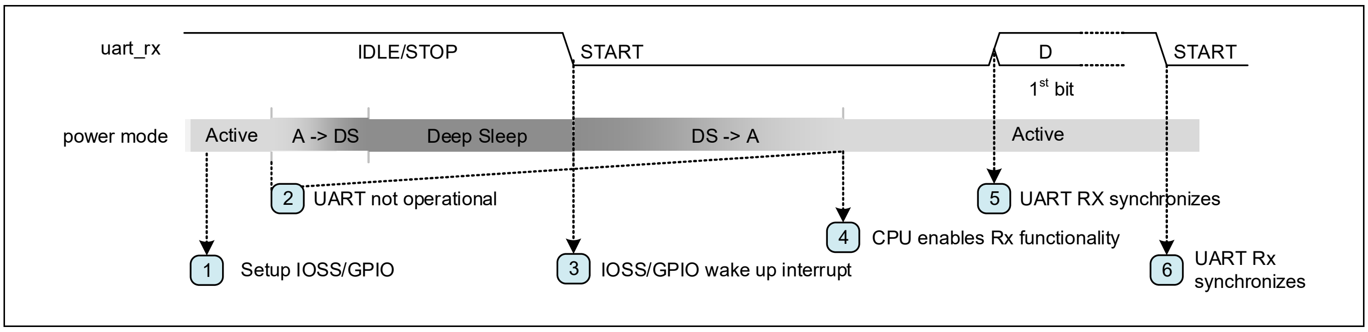


Figure 15-17. UART start skip and wakeup from Deep Sleep

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.

Note: *If skip start is used and a wakeup occurs from another source then a dummy byte will be received in the RX_FIFO when the RX line is in idle.*

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 15-18 illustrates a regular data frame and break frame (8-bit data frame, parity support, and a break width of 12-bit periods).

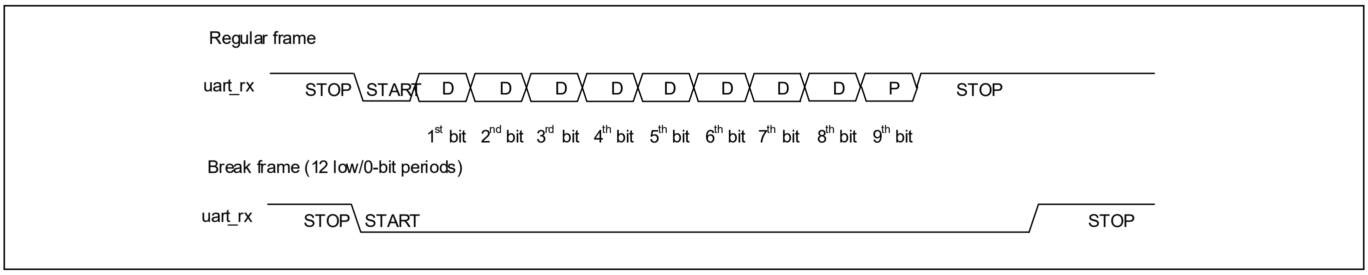

Regular frame
uart_rx
STOP
START
D
D
D
D
D
D
D
D
P
STOP
1st bit 2nd bit 3rd bit 4th bit 5th bit 6th bit 7th bit 8th bit 9th bit
Break frame (12 low/0-bit periods)
uart_rx
STOP
START
STOP

Figure 15-18. 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 trans-fer 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 de-rived 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 15-19** gives an overview of the flow control functionality.

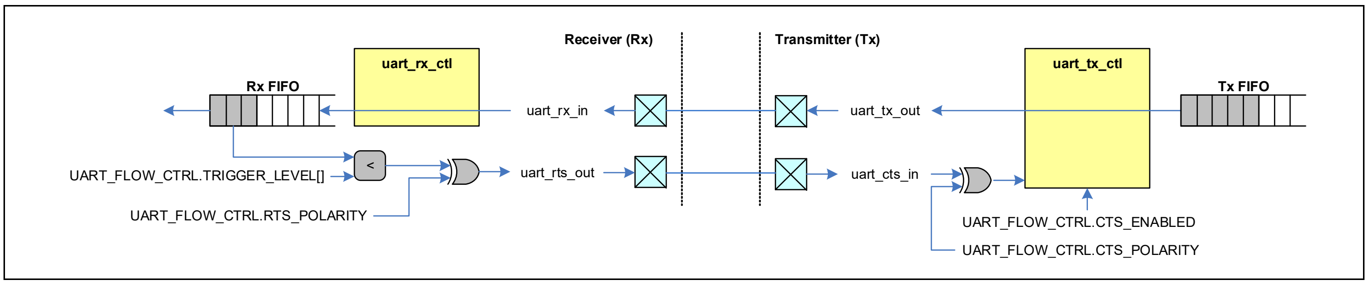

Receiver (Rx)
Transmitter (Tx)
uart_rx_ctl
Rx FIFO
uart_rx_in
uart_rts_out
<
UART_FLOW_CTRL.TRIGGER_LEVEL[]
UART_FLOW_CTRL.RTS_POLARITY
uart_tx_ctl
Tx FIFO
uart_tx_out
uart_cts_in
UART_FLOW_CTRL.CTS_ENABLED
UART_FLOW_CTRL.CTS_POLARITY

Figure 15-19. UART flow control connection

UART Multi-processor mode

The UART_MP (multi-processor) mode is defined with single-master-multi-slave topology, as **Figure 15-20** 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.

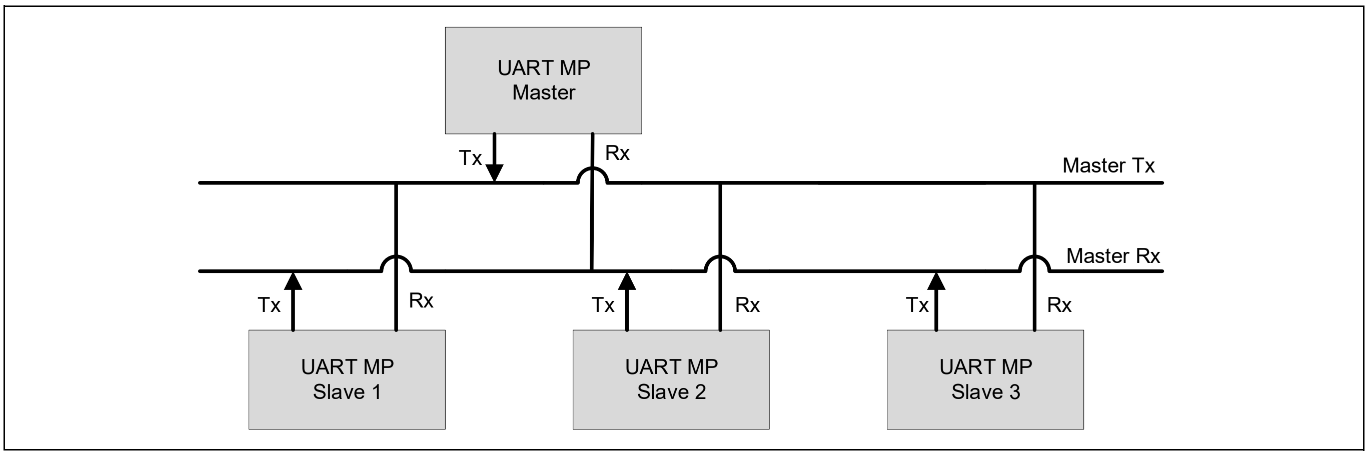


Figure 15-20. UART MP mode bus connections

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 15-21**.
- Parity bit is disabled.

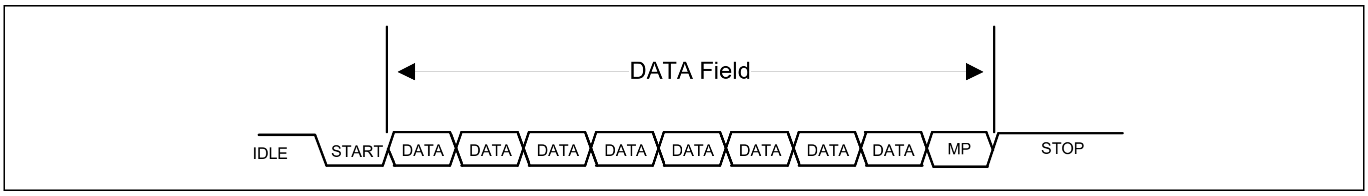


Figure 15-21. UART MP address and data frame

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 112.

For more information on these registers, see the PSoC™ 4000T MCU registers TRM.

15.4.3.2 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 15-22** illustrates the UART_LIN and LIN transceiver.

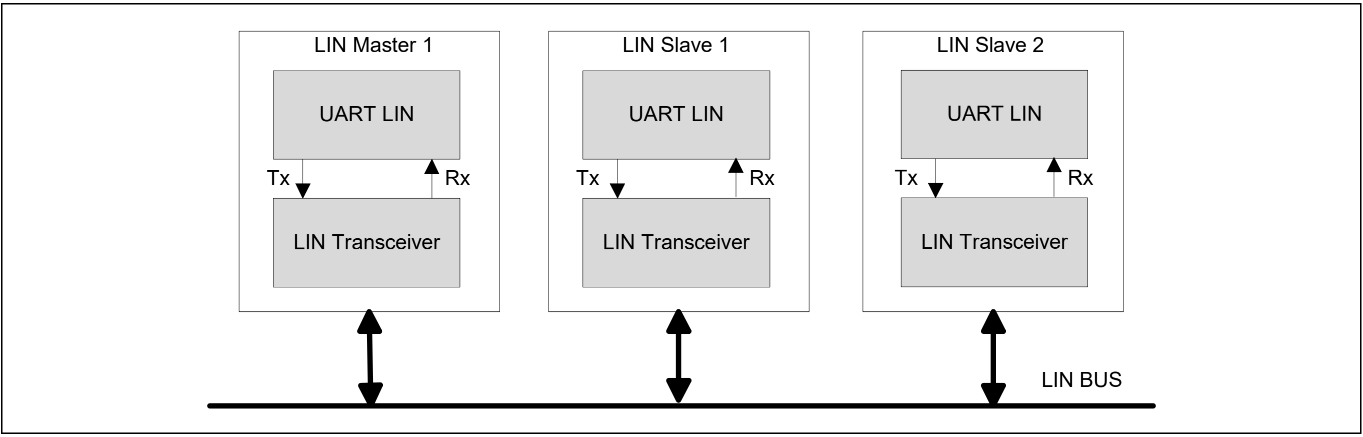


Figure 15-22. UART_LIN and LIN transceiver

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 15-23**.

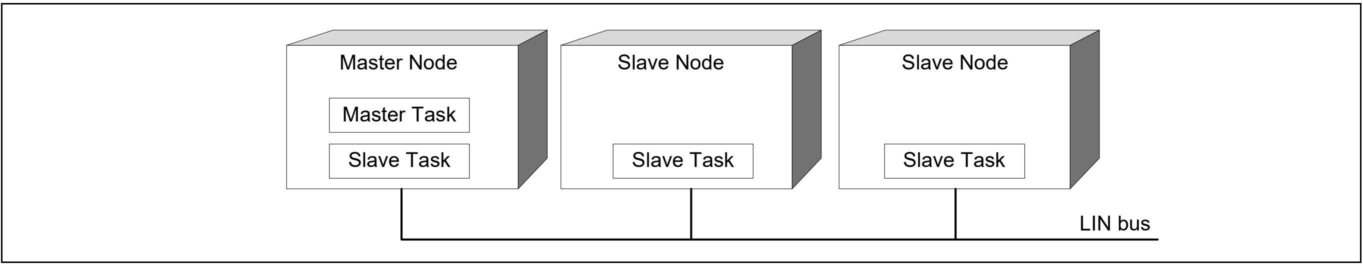


Figure 15-23. LIN bus nodes and tasks

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 15-24**.

- 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.

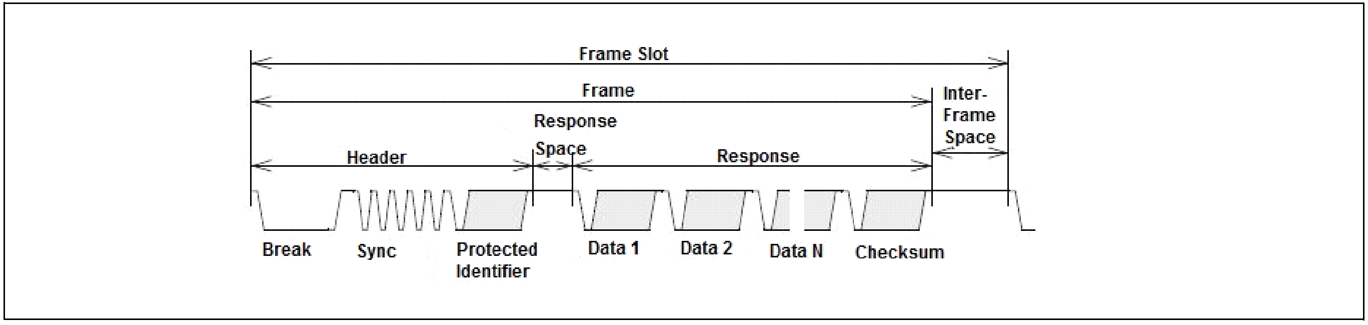


Figure 15-24. LIN frame structure

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 15-25**.

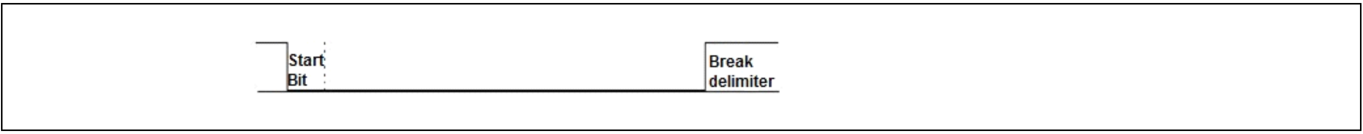


Figure 15-25. LIN break field

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 15-26** shows the LIN sync field structure.

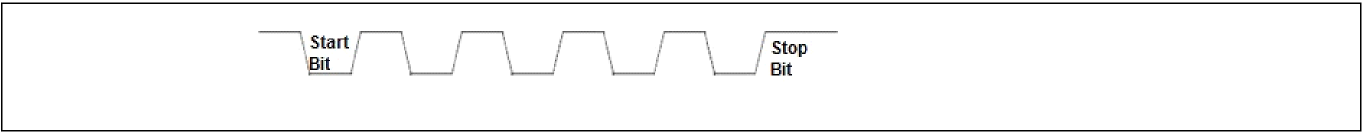


Figure 15-26. LIN sync field

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 15-27**.

- 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 15-27 shows the PID field structure.

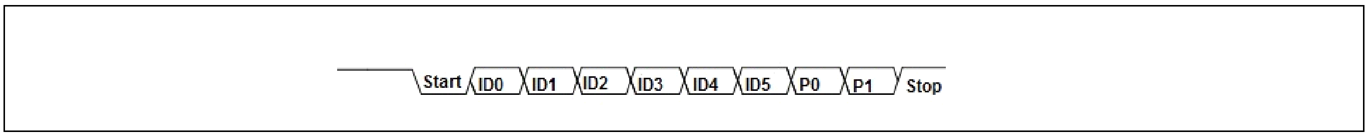


Figure 15-27. PID field

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).

Note: *LIN go-to-sleep and wake-up should be taken care in user firmware.*

Configuring the SCB in UART LIN mode

1. Configure the SCB as UART interface by writing '10' 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 LIN Mode, write '1' to the LIN_MODE (bit 12) of the SCB_UART_RX_CTRL register.
4. Follow steps 2 to 4 described in Enabling and Initializing UART.

15.4.3.3 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 acknowledgment (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 half bit transfer period into the stop period, the receiver may drive a NACK on the line (a value of ‘0’) for one to two 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 at least two bits of 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 15-28 illustrates the SmartCard protocol.

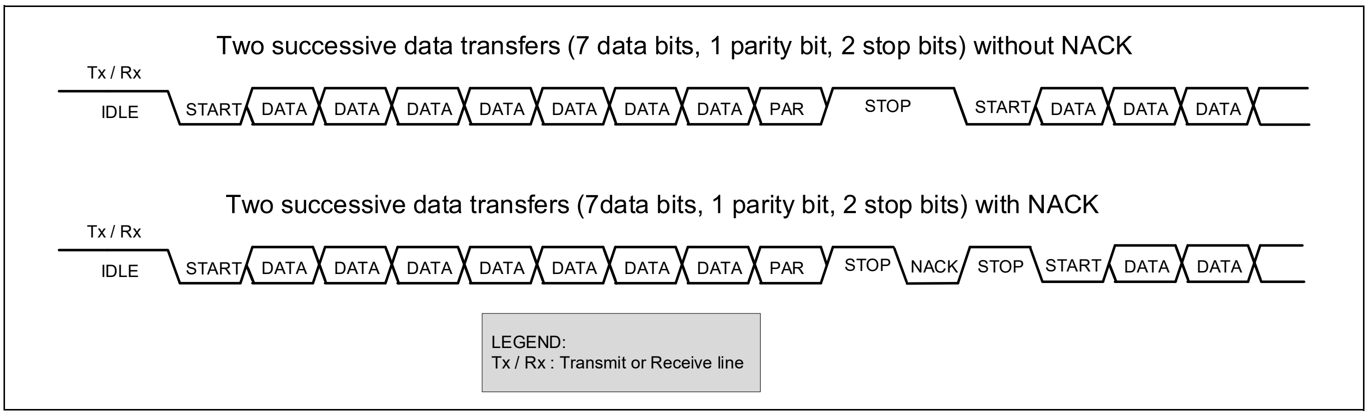

Two successive data transfers (7 data bits, 1 parity bit, 2 stop bits) without NACK
Tx / Rx
IDLE
START
DATA
PAR
STOP
Two successive data transfers (7data bits, 1 parity bit, 2 stop bits) with NACK
NACK
LEGEND:
Tx / Rx : Transmit or Receive line

Figure 15-28. 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™ software does all this automatically with the help of GUIs. For more information on these registers, see the PSoC™ 4000T MCU 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 112.

15.4.3.4 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™ 4 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 15-29 shows how a UART transfer is IrDA modulated.

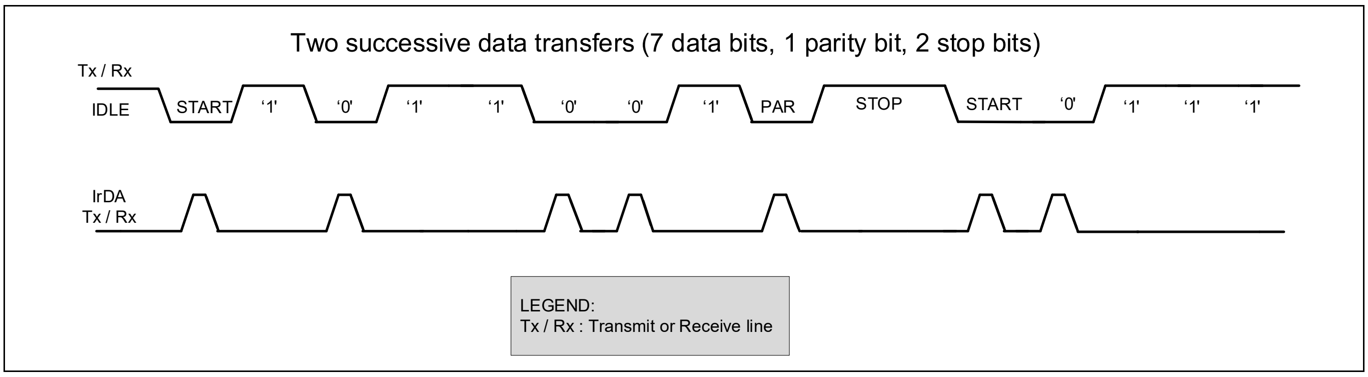

Two successive data transfers (7 data bits, 1 parity bit, 2 stop bits)
Tx / Rx
IDLE
START
'1'
'0'
'1'
'1'
'0'
'0'
'1'
PAR
STOP
START
'0'
'1'
'1'
'1'
IrDA Tx / Rx
LEGEND:
Tx / Rx : Transmit or Receive line

Figure 15-29. 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™ Software does all this automatically with the help of GUIs. For more information on these registers, see the PSoC™ 4000T MCU 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 112.

15.4.4 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 PSoC™ 4000T MCU registers TRM for more details on the OVS values for different baud rates.

15.4.5 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.
   c) Set the trigger level.
   d) 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).

15.4.6 I/O pad connection

15.4.6.1 Standard UART mode

Figure 15-30 and Table 15-7 list the use of the I/O pads for the Standard UART mode.

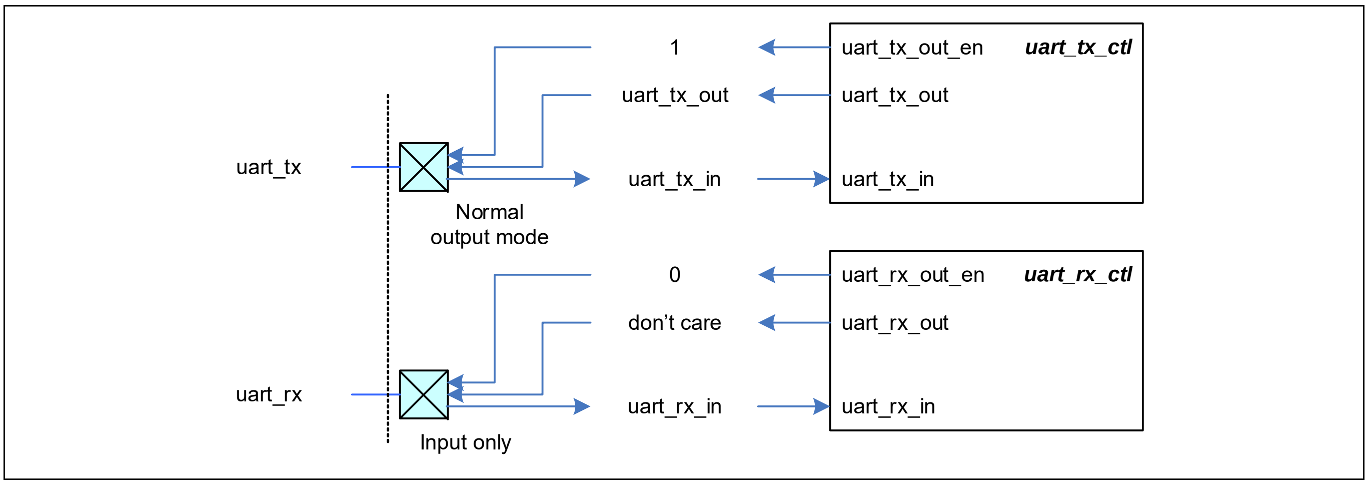


Figure 15-30. Standard UART mode I/O pad connections

Table 15-7. 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

15.4.6.2 SmartCard mode

Figure 15-31 and **Table 15-8** list the use of the I/O pads for the SmartCard mode.

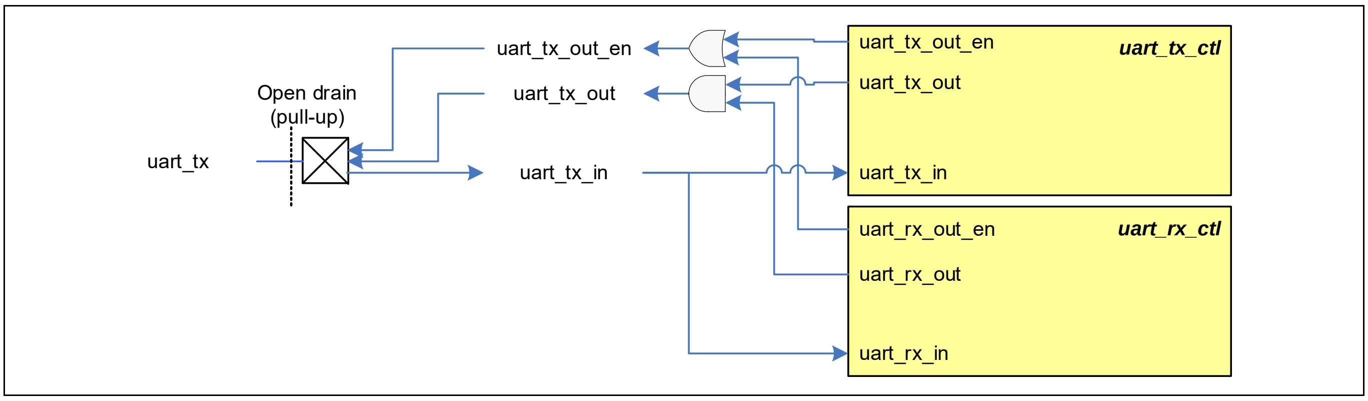


Figure 15-31. SmartCard mode I/O pad connections

Table 15-8. 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 acknowledgement 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.

15.4.6.3 LIN mode

Figure 15-32 and **Table 15-9** list the use of the I/O pads for LIN mode.

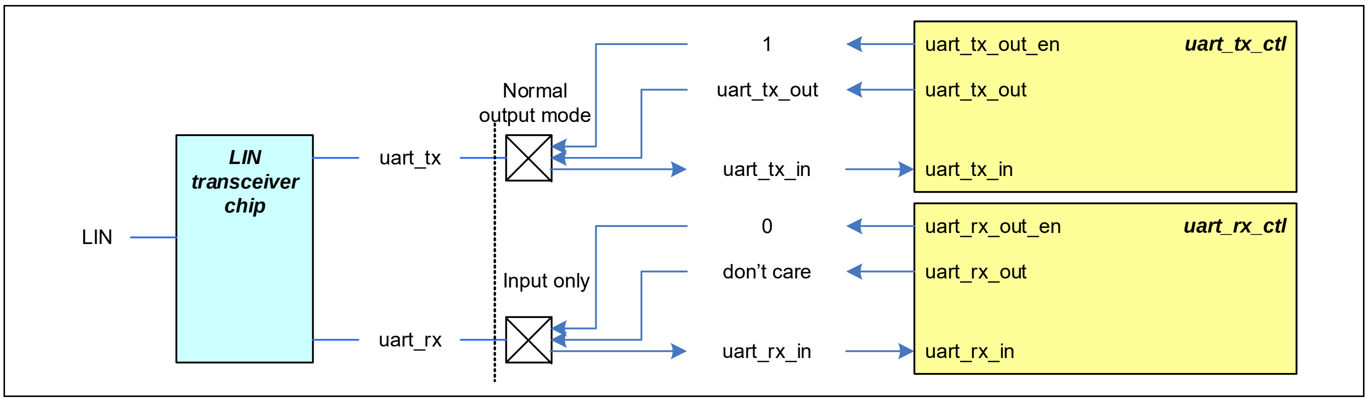


Figure 15-32. LIN mode I/O pad connections

Table 15-9. 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.

15.4.6.4 IrDA mode

Figure 15-33 and **Table 15-10** list the use of the I/O pads for IrDA mode.

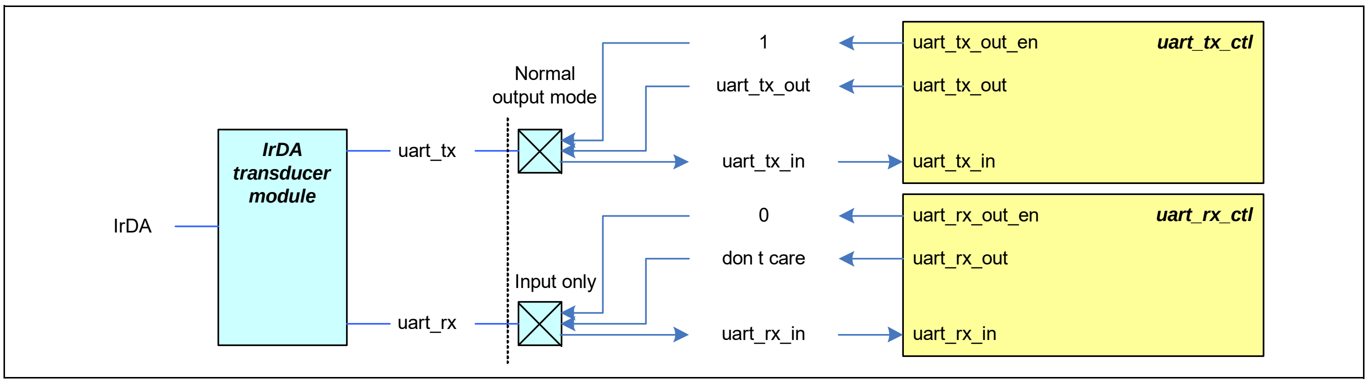


Figure 15-33. IrDA mode I/O pad connections

Table 15-10. 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.

15.4.7 UART registers

The UART interface is controlled using a set of 32-bit registers listed in **Table 15-11**. For more information on these registers, see the PSoC™ 4000T MCU registers TRM.

Table 15-11. 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.

15.5 Inter integrated circuit (I^{2}C)

This section explains the I^{2}C implementation in the PSoC™ 4 MCU. For more information on the I^{2}C protocol specification, refer to the I^{2}C-bus specification available on the **NXP website**.

15.5.1 Features

This block supports the following features:

- Master, slave, and master/slave mode
- Standard-mode (100 kbps), fast-mode (400 kbps), fast-mode plus (1000 kbps)
- 7-bit slave addressing
- Clock stretching
- Collision detection
- Programmable oversampling of I^{2}C clock signal (SCL)
- Auto ACK when RX FIFO not full, including address
- General address detection
- FIFO and EZ modes

15.5.2 General description

Figure 15-34 illustrates an example of an I^{2}C communication network.

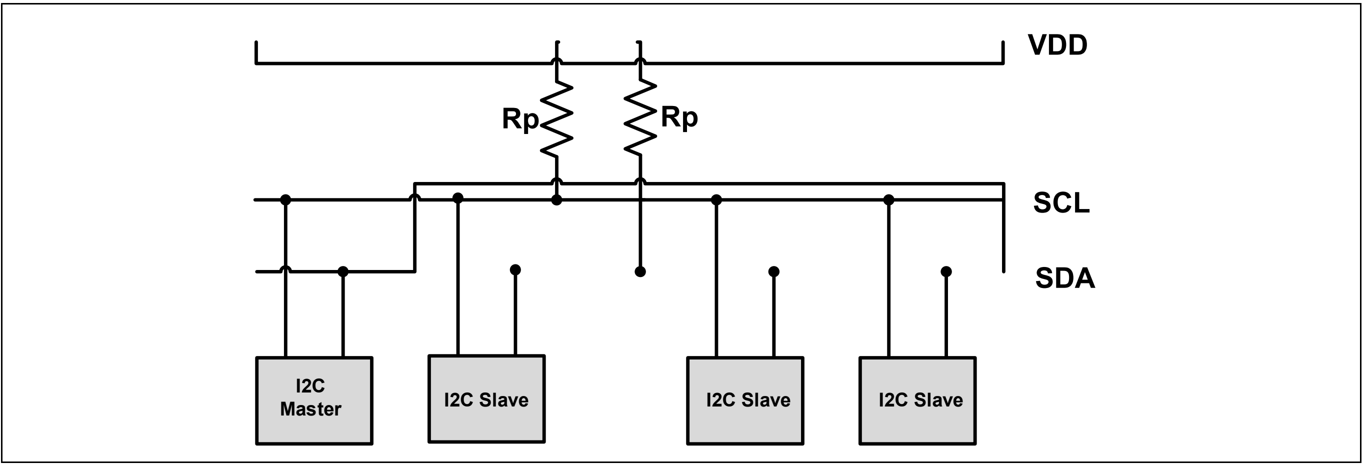


Figure 15-34. I^{2}C interface block diagram

The standard I^{2}C bus is a two wire interface with the following lines:

- Serial Data (SDA)
- Serial Clock (SCL)

I^{2}C 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.

15.5.3 External electrical connections

As shown in **Figure 15-35**, the I^{2}C 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 Infineon® recommends using the UM10204 I^{2}C-bus specification and user manual Rev. 6, available from the NXP website at **www.nxp.com**.

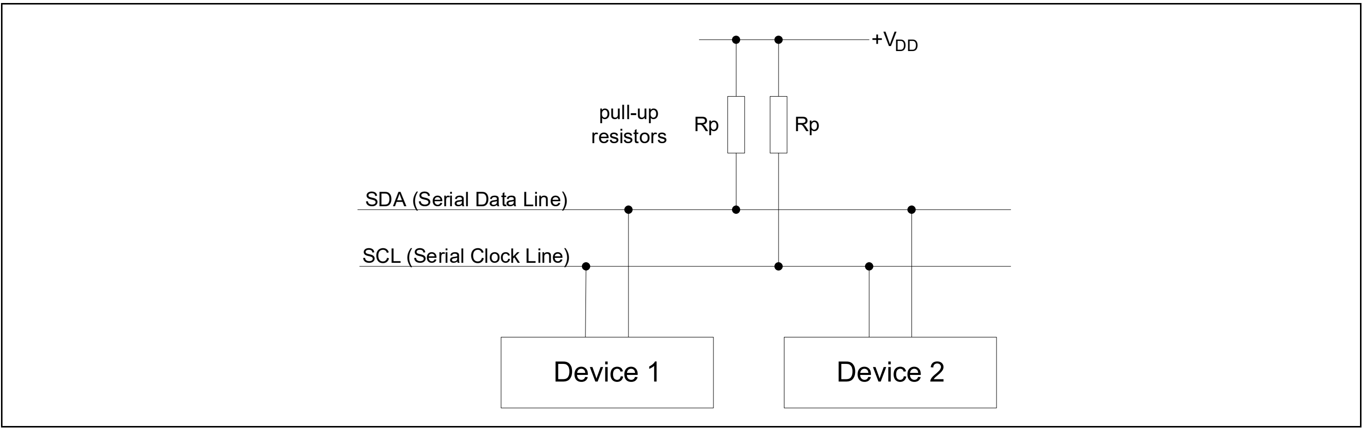


Figure 15-35. Connection of devices to the I2C bus

For most designs, the default values shown in **Table 15-12** provide excellent performance without any calculations. The default values were chosen to use standard resistor values between the minimum and maximum limits.

Table 15-12. Recommended default pull-up resistor values

Standard mode (0 – 100 kbps)	Fast mode (0 – 400 kbps)	Fast mode plus (0 – 1000 kbps)	Unit
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 I^{2}C specification. These equations are:

$$R_{PMIN} = (V_{DD}(max) - V_{OL}(max)) / I_{OL}(min) \tag{15.4}$$

$$R_{PMAX} = T_R(max) / 0.8473 \times C_B(max) \tag{15.5}$$

$$R_{PMAX} = V_{DD}(min) - (V_{IH}(min) + V_{NH}(min)) / I_{IH}(max) \tag{15.6}$$

Equation parameters:

- V_{DD} = Nominal supply voltage for I^{2}C bus
- V_{OL} = Maximum output low voltage of bus devices
- I_{OL}= Low-level output current from I^{2}C specification
- T_R = Rise time of bus from I^{2}C 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 I^2C 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 15.4 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 15.5 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 15.6. 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.

15.5.4 Terms and definitions

Table 15-13 explains the commonly used terms in an I^2C communication network.

Table 15-13. 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

15.5.4.1 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™ 4 MCU can and will stretch the clock.

15.5.4.2 Bus arbitration

The I^{2}C 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.

15.5.5 I^{2}C modes of operation

I^{2}C 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 15-14** illustrates the I^{2}C modes of operation.

Table 15-14. I^{2}C modes

Mode	Description
Slave	Slave only operation (default)
Master	Master only operation
Multi-master	Supports more than one master on the bus
Master-slave	The SCB can change between master and slave modes.

Table 15-15 lists some common bus events that are part of an I^{2}C data transfer. The **"Write transfer"** on page 119 and **"Read transfer"** on page 119 sections explain the I^{2}C bus bit format during data transfer.

Table 15-15. I^{2}C 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.

Note: High speed mode requires HW detection of the following sequence: START, ADDR, NACK.

15.5.5.1 Write transfer

- A typical write 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 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.

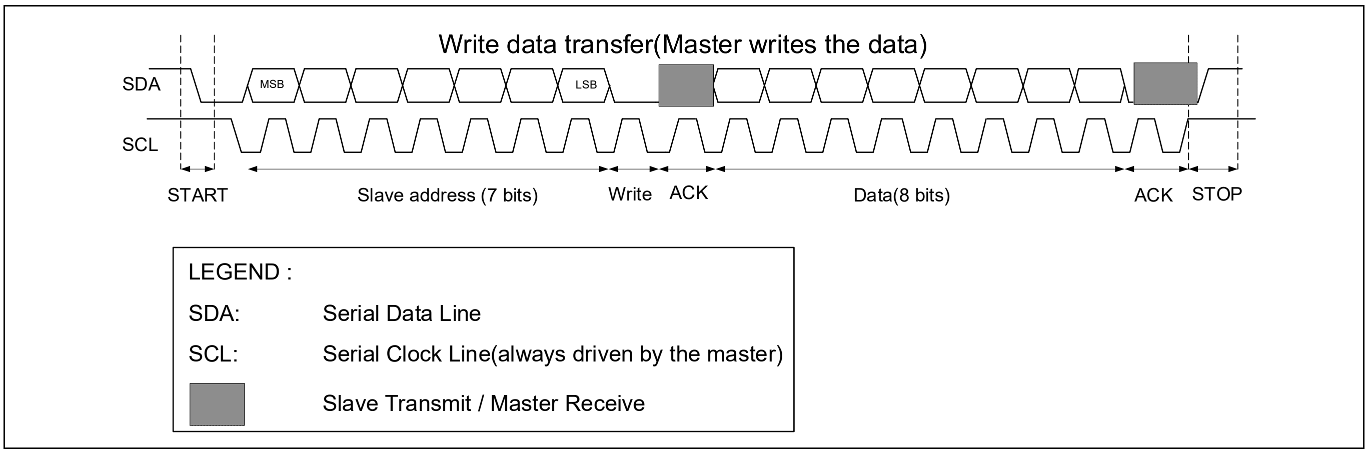

Write data transfer(Master writes the data)
SDA
SCL
MSB
LSB
START
Slave address (7 bits)
Write
ACK
Data(8 bits)
ACK
STOP
LEGEND :
SDA: Serial Data Line
SCL: Serial Clock Line(always driven by the master)
Slave Transmit / Master Receive

Figure 15-36. Master write data transfer

15.5.5.2 Read transfer

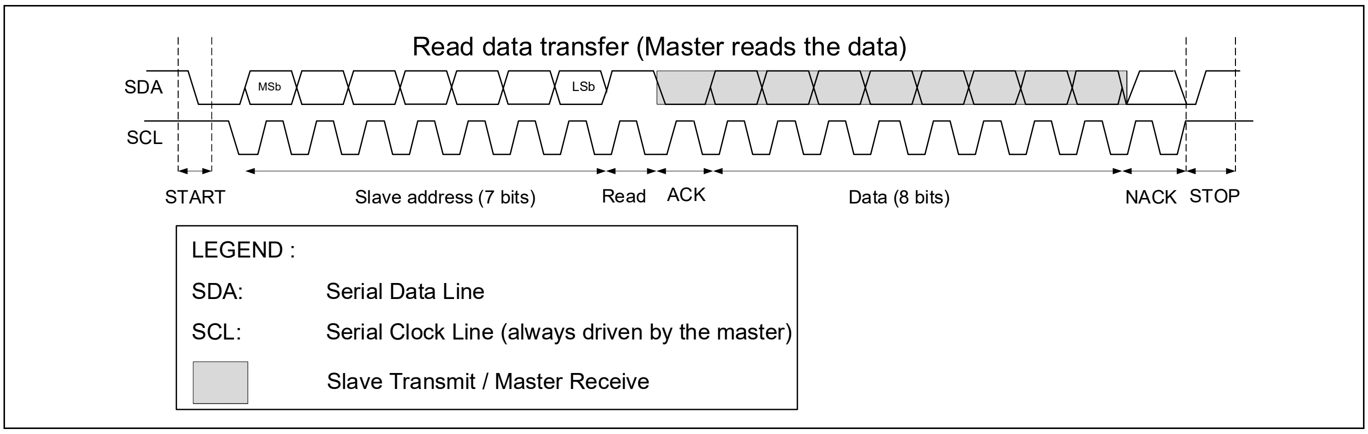

Read data transfer (Master reads the data)
SDA
SCL
MSb
LSb
START
Slave address (7 bits)
Read
ACK
Data (8 bits)
NACK
STOP
LEGEND :
SDA: Serial Data Line
SCL: Serial Clock Line (always driven by the master)
Slave Transmit / Master Receive

Figure 15-37. Master read data transfer

- A typical read transfer begins with the master generating a START condition on the I^2C bus. The master then writes a 7-bit I^2C 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.

15.5.6 I^2C buffer modes

I^2C can operate in two different buffered modes – FIFO and EZ modes. The buffer is used in different ways in each of the modes. The following subsections explain each of these buffered modes in detail.

15.5.6.1 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 8 elements deep with 16-bit data elements or 16 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^2C, 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, 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 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.

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 disabling the clock divider that supplies the clock to the SCB, see the **"Clocking system"** on page 54 for more information.

Deep Sleep to Active transition

EC_AM = 1, EC_OP = 0, FIFO Mode.

The following descriptions only apply to slave mode.

The system wakes up from Sleep or Deep-Sleep system power modes when an I2C address match occurs. The fixed-function I2C block performs either of two actions after address match: Address ACK or Address NACK.

- Address ACK: The I^{2}C slave executes clock stretching and waits until the device wakes up and ACKs the address.
- Address NACK: The I^{2}C slave NACKs the address immediately. The master must poll the slave again after the device wakeup time is passed. This option is only valid in the slave mode.

Note:

- The interrupt bit WAKE_UP (bit 0) of the SCB_INTR_I2C_EC register must be enabled for the I^{2}C to wake up the device on slave address match while switching to the Sleep mode.
- If the device is configured in I^{2}C slave mode, the clock to the SCB should be disabled when entering Deep-Sleep power mode; enable the clock when waking up from Deep-Sleep mode.

15.5.6.2 EZI2C mode

The Easy I^{2}C (EZI2C) protocol is a unique communication scheme built on top of the I^{2}C protocol by Infineon®. It uses a meta protocol around the standard I^{2}C protocol to communicate to an I^{2}C 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 (32-entry array of 8-bit per entry is supported) located on the slave device. The EZ address is used to address these 32 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^{2}C 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 (31), 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^{2}C 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 SCB can implement a hardware based EZI2C that can operate in deep sleep. This document focuses on hardware-implemented EZI2C.

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.

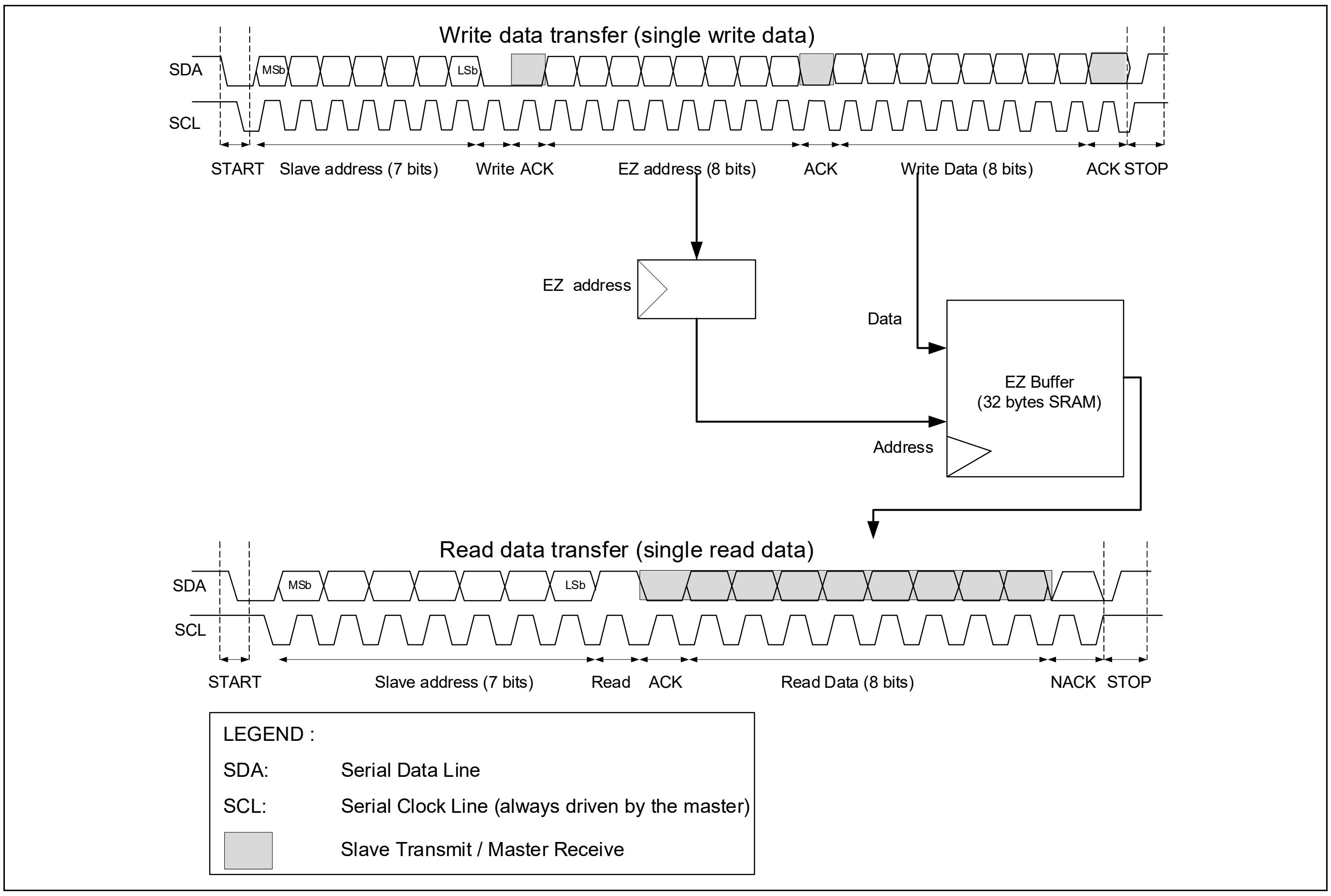


Figure 15-38. EZI2C write and read data transfer

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 disabling the clock divider that supplies the clock to the SCB, see the **"Clocking system"** on page 54 for more information.

Deep Sleep to Active transition

EC_AM = 1, EC_OP = 0, EZ Mode.

The system wakes up from Sleep or Deep-Sleep system power modes when an I^{2}C address match occurs. The fixed-function I^{2}C block performs either of two actions after address match: Address ACK or Address NACK.

- **Address ACK**: The I^{2}C slave executes clock stretching and waits until the device wakes up and ACKs the address.
- **Address NACK**: The I^{2}C slave NACKs the address immediately. The master must poll the slave again after the device wakeup time is passed. This option is only valid in the slave mode.

Note:

- The interrupt bit WAKE_UP (bit 0) of the SCB_INTR_I2C_EC register must be enabled for the I^{2}C to wake up the device on slave address match while switching to the Sleep mode
- If the device is configured in I^{2}C slave mode, the clock to the SCB should be disabled when entering Deep-Sleep power mode; enable the clock when waking up from Deep-Sleep mode.

15.5.7 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"** on page 54.

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
- 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 matsching 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 120.
  - EZ mode: See **"Deep Sleep to Active transition"** on page 120.
- 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.

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 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™ 4 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).

15.5.7.1 Glitch filtering

The PSoC™ 4 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 (three input median 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.

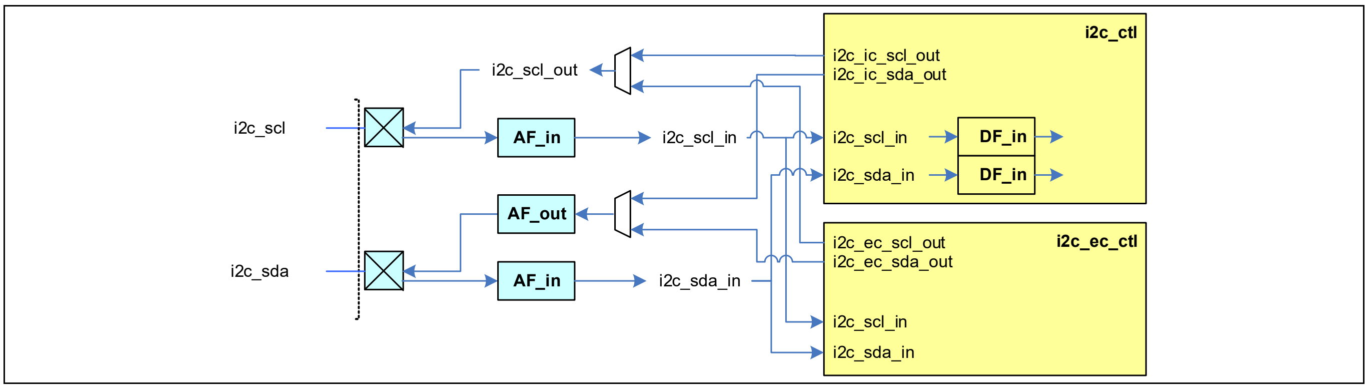


Figure 15-39. I^{2}C glitch filtering connection

Table 15-16 lists the useful combinations of glitch filters.

Table 15-16. 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, 1000-kHz, and high speed 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

15.5.7.2 Oversampling and bit rate

Internally-clocked master

The PSoC™ 4 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 15-17. I^{2}C frequency and oversampling requirements in I^{2}C 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	[495, 1000] kHz	[6, 15]	[9, 15]	[16.15-25.29] MHz

Table 15-17 assumes worst-case conditions on the I^{2}C bus. The following equations can be used to determine the settings for your own system. These are general guidelines and recommendations for I^{2}C timing compliance. 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 I^{2}C specification will be violated. t_F needs to be measured in your system.

$t_{CLK\_SCB(Max)} = (t_{VD} - t_{RF} - 100\text{ ns})/3$ (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)

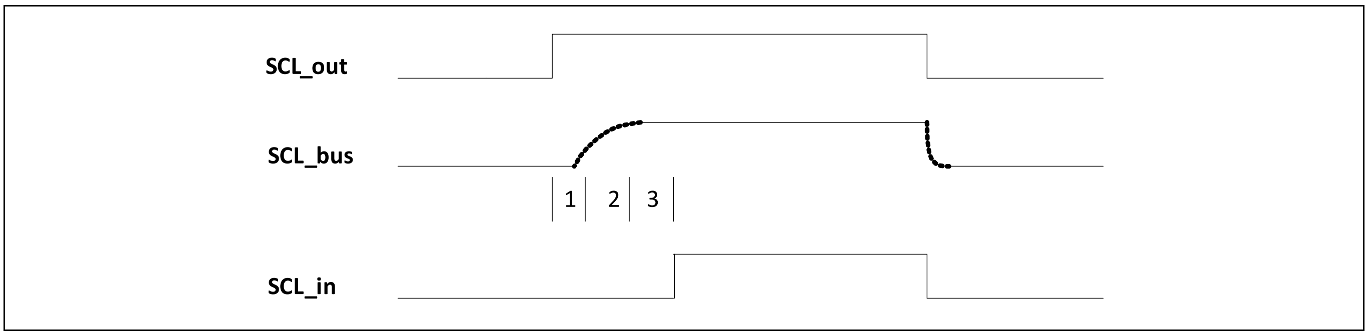


Figure 15-40. I²C SCL turnaround path

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 I²C frequency is determined by the incoming I²C SCL signal. To ensure proper operation, clk_scb must be significantly higher than the I²C bus frequency. Unlike master mode, this mode does not use programmable oversampling factors.

Table 15-18. SCB input clock requirements in I²C 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

$t_{CLK\_SCB(Max)} = (t_{VD} - t_{RF} - 100\text{ ns}) / 3$ (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

$t_{CLK\_SCB(Min)} = (0.6 * t_F - 50\text{ ns})/2$ (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

To configure the I^{2}C for master-slave mode, write ‘1’ to the MASTER_MODE(bit-31) and SLAVE_MODE(bit-30) of the I2C_CTRL register. When the I^{2}C initializes a transfer, it is a master and when it is addressed by anther master it is a slave. In this mode, when the SCB is acting as a master device, the block determines the I^{2}C frequency. When the SCB is acting as a slave device, the block does not determine the I^{2}C frequency. Instead, the incoming I^{2}C 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.

Note: *The I^{2}C cannot support wakeup from deepsleep power modes in fast-plus mode (1 MHz), when configured in master-slave mode.*

15.5.8 Enabling and initializing the I^{2}C

The following section describes the method to configure the I^{2}C block for standard (non-EZ) mode and EZI2C mode.

15.5.8.1 Configuring for I^{2}C FIFO mode

The I^{2}C 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^{2}C block and select the I^{2}C mode. For a complete description of the I^{2}C registers, see **“I2C registers”** on page 129.

15.5.8.2 Configuring for EZ mode

To configure the I^{2}C block for EZ mode, set the following I^{2}C register bits:

1. Select the EZI2C mode by writing ‘1’ to the EZ_MODE bit (bit 10) 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”*** *on page 54.*

The GPIO must also be connected to the SCB; see the following section for more details.

15.5.9 I/O pad connections

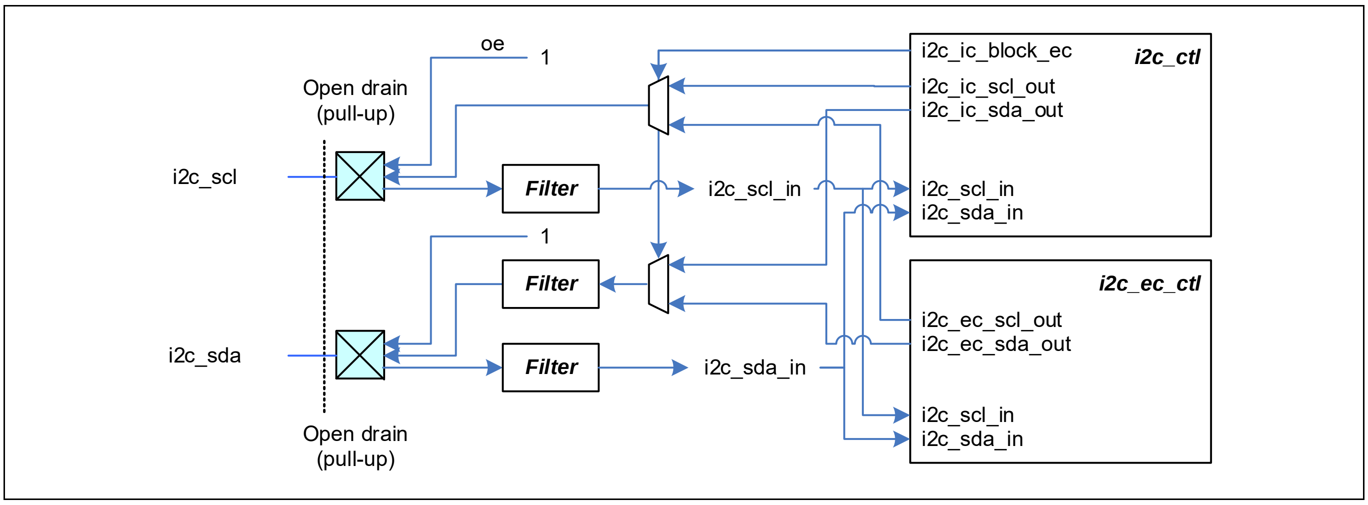


Figure 15-41. I^{2}C I/O pad connections

Table 15-19. I^{2}C I/O pad description

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 I^{2}C SDA/SCL lines, the following sequence must be followed. If this sequence is not followed, the I^{2}C 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 I^{2}C pins for high-impedance drive mode.
5. Configure SCB for I^{2}C
6. Enable SCB
7. Configure I^{2}C pins for Open Drain Drives Low.

15.6 I^{2}C registers

The I^{2}C interface is controlled by reading and writing a set of configuration, control, and status registers, as listed in **Table 15-20**.

Table 15-20. I^{2}C registers

Register	Function
SCB_CTRL	Enables the SCB block and selects the type of serial interface (SPI, UART, I^{2}C). **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_I2C_STRETCH_CTRL	Specifies the stretch threshold, which is SCL turnaround delay in number of clk_scb cycles.
SCB_I2C_STRETCH_STATUS	Indicates the turnaround count, stretch detection, and synchronization status.
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.

Table 15-20. I²C registers *(continued)*

Register	Function
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 *PSoC™ 4000T MCU registers TRM.*

15.7 SCB interrupts

SCB supports interrupt generation on various events. The interrupts generated by the SCB block vary depending on the mode of operation.

Table 15-21. SCB interrupts

Interrupt	Functionality	Active/Deep Sleep	Registers
interrupt_master	I²C master and SPI master functionality	Active	INTR_M, INTR_M_SET, INTR_M_MASK, INTR_M_MASKED
interrupt_slave	I²C 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 I²C 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.

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 15-42 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.

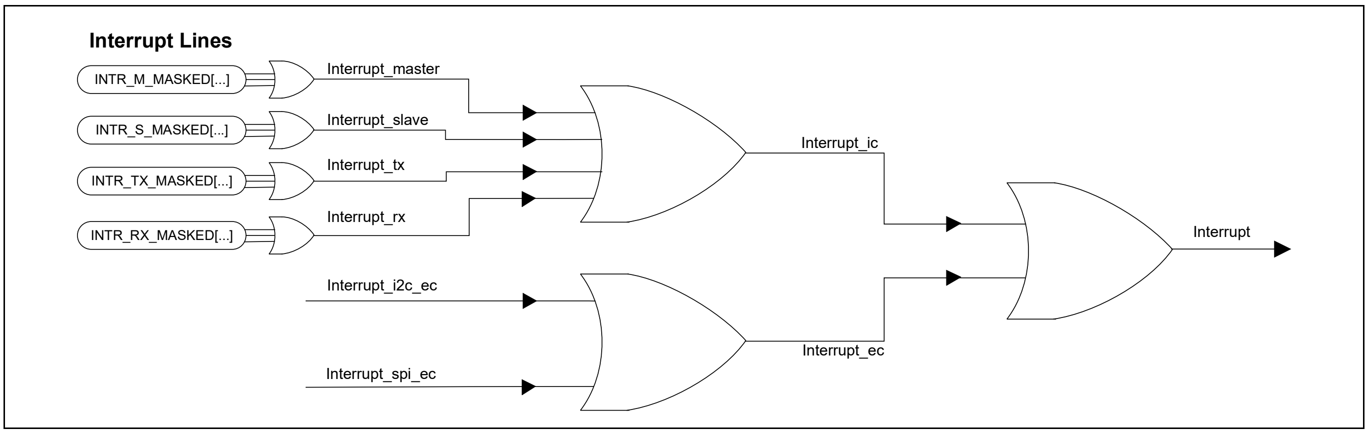


Figure 15-42. Interrupt lines

15.7.1 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 PSoC™ 4000T MCU 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 mode, 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 mode, 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 mode when EC_OP is '1'.

Figure 15-43 and Figure 15-44 show how each of the interrupts are triggered. Figure 15-43 shows the TX buffer and the corresponding interrupts while Figure 15-44 shows all the corresponding interrupts for the RX buffer. The FIFO has 32 bytes split into 16 bytes for TX and 16 bytes for RX instead of the 8 bytes shown below.

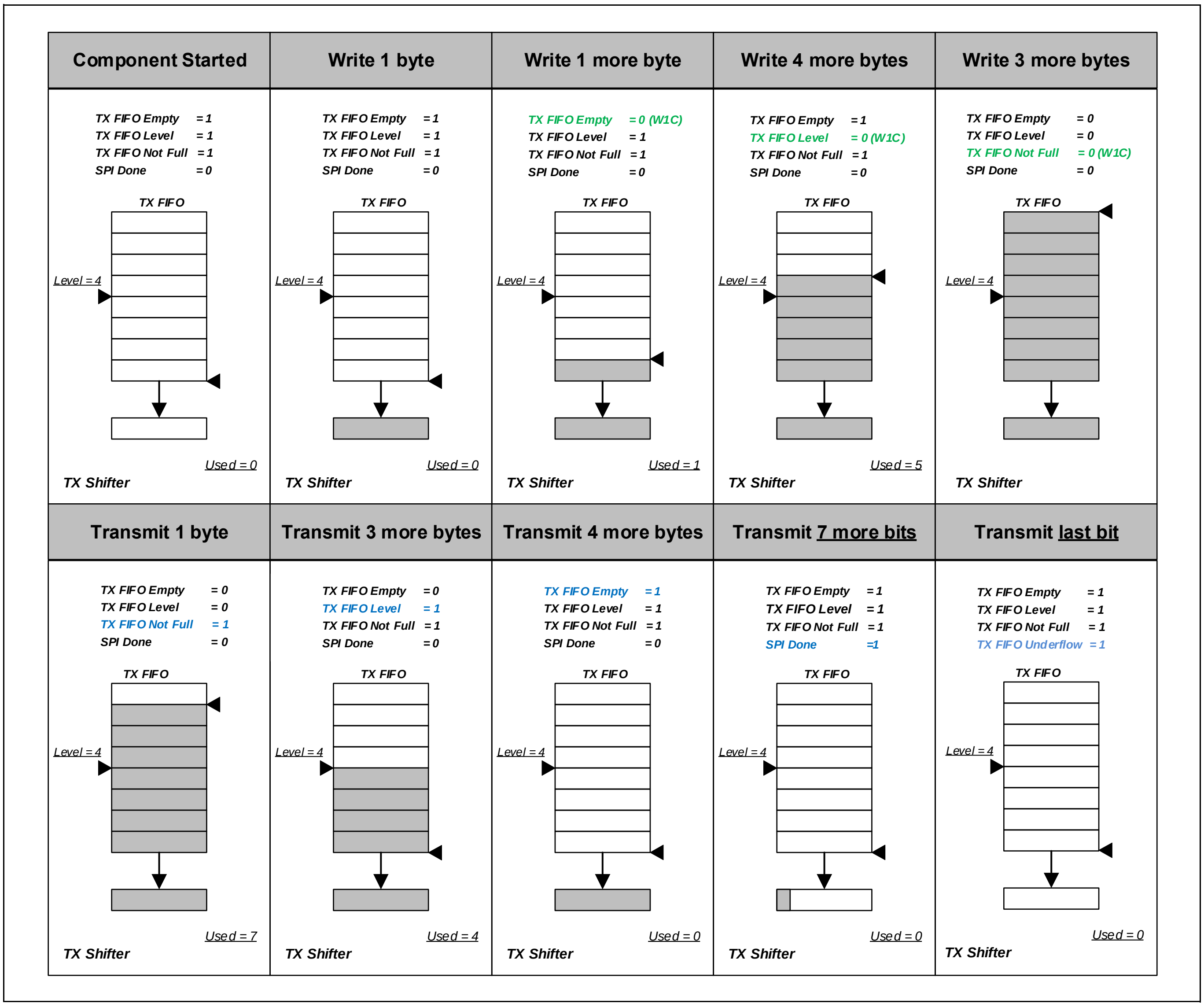


Figure 15-43. TX interrupt source operation

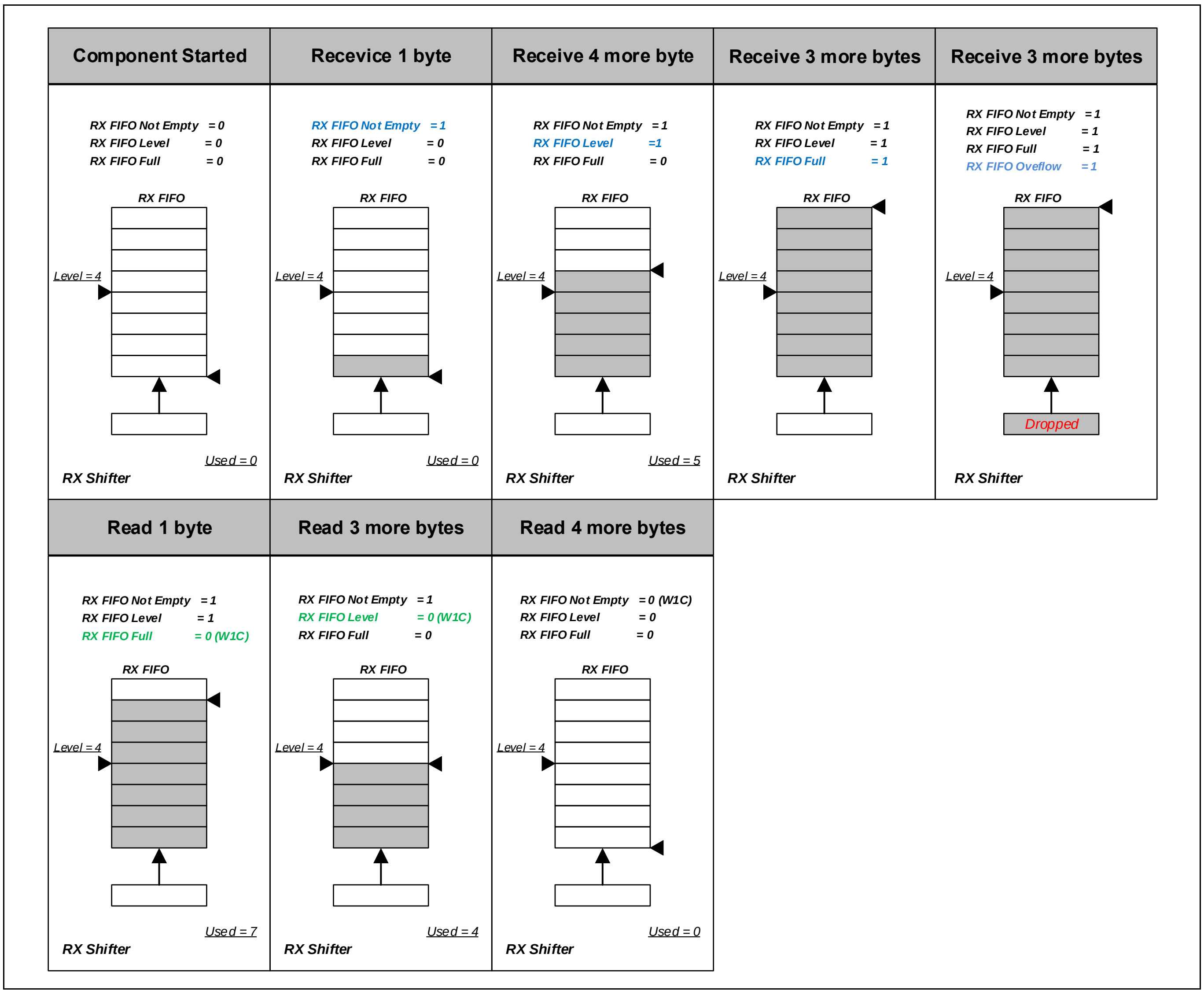


Figure 15-44. RX interrupt source operation

15.7.2 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 PSoC™ 4000T MCU 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 15-45 and **Figure 15-46** show how each of the interrupts are triggered. **Figure 15-45** shows the TX buffer and the corresponding interrupts while **Figure 15-46** shows all the corresponding interrupts for the RX buffer. The FIFO has 32 bytes split into 16 bytes for TX and 16 bytes for RX instead of the 8 bytes shown below.

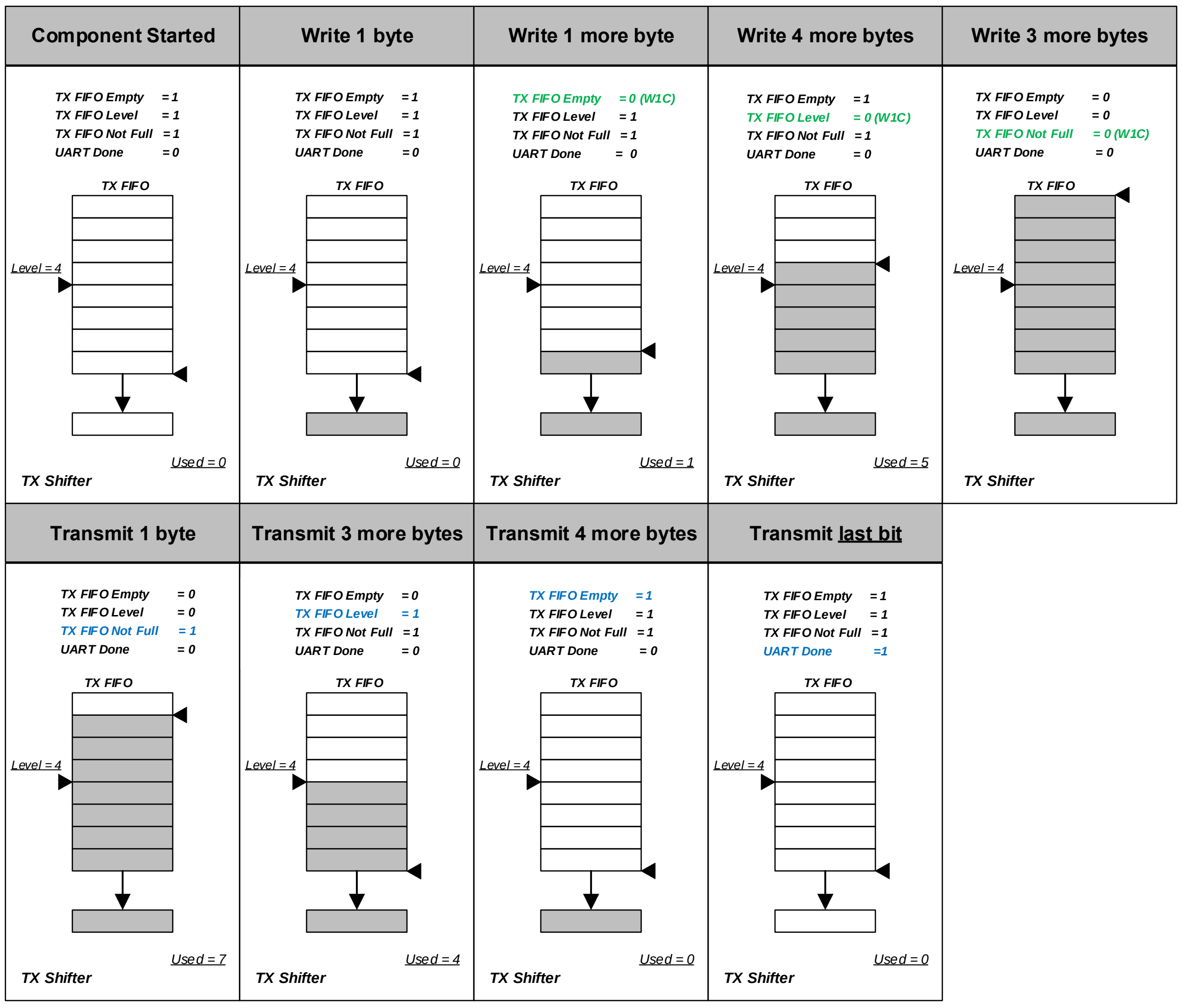


Figure 15-45. TX interrupt source operation

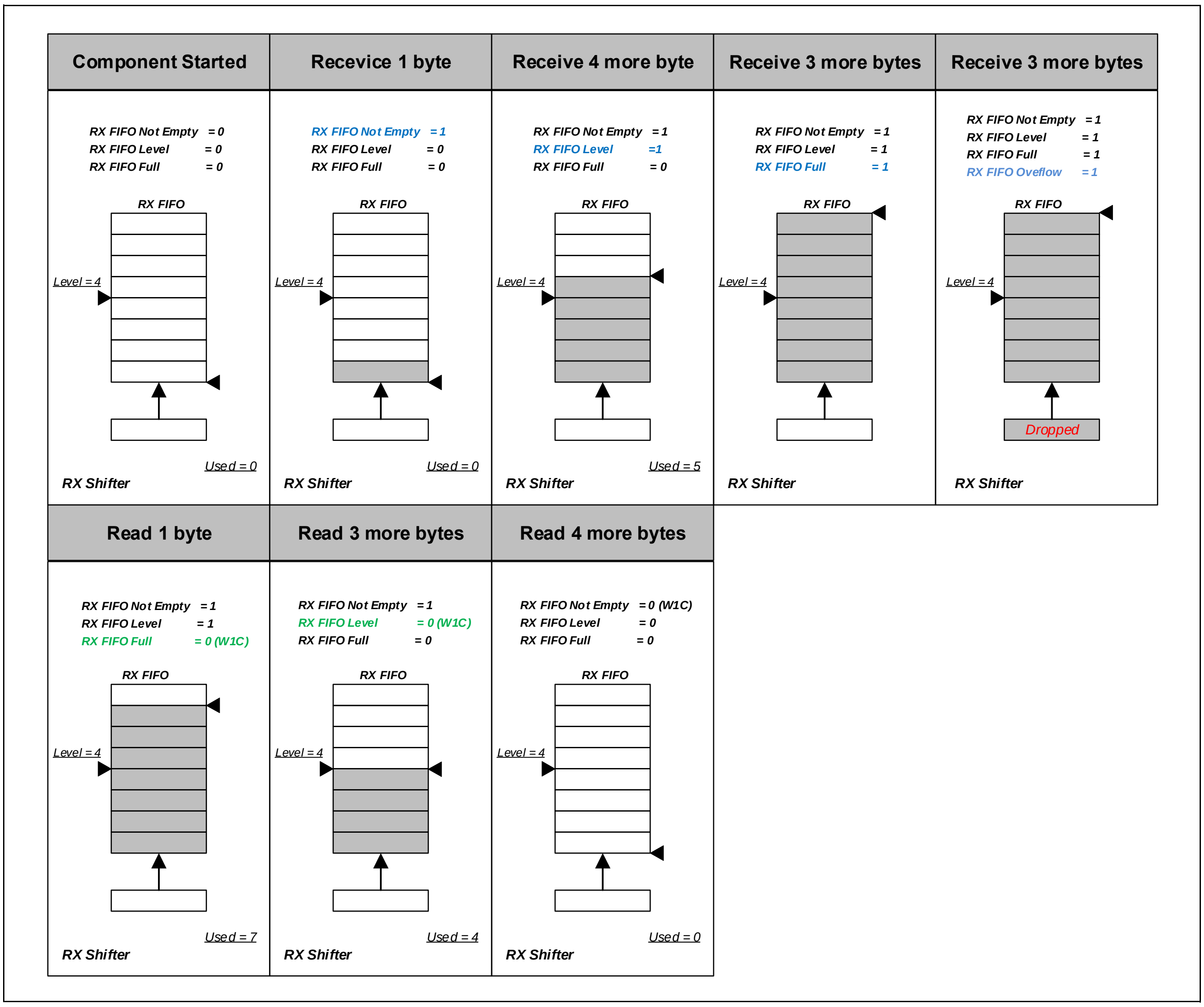


Figure 15-46. RX interrupt source operation

15.7.3 I^{2}C interrupts

I^{2}C 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 PSoC™ 4000T MCU registers TRM. The I^{2}C block generates interrupts for the following conditions.

- **I^{2}C Master**
  - I^{2}C 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.
  - I^{2}C master received NACK – When the master receives a NACK (typically after the master transmitted the slave address or TX data).
  - I^{2}C master received ACK – When the master receives an ACK (typically after the master transmitted the slave address or TX data).
  - I^{2}C master sent STOP – When the master has transmitted a STOP.
  - I^{2}C bus error – Unexpected stop/start condition is detected.
- **I^{2}C Slave**
  - I^{2}C 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 I^{2}C bus behavior. In case of lost arbitration, the I^{2}C slave state machine aborts the ongoing transfer. Software may decide to clear the TX and RX FIFOs in case of this error.
  - I^{2}C slave received NACK – When the slave receives a NACK (typically after the slave transmitted TX data).
  - I^{2}C slave received ACK – When the slave receives an ACK (typically after the slave transmitted TX data).
  - I^{2}C slave received STOP – I^{2}C STOP event for I^{2}C (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 I^{2}C transfers separated by a REPEATED START can be distinguished and potentially treated separately by the firmware. Note that the second I^{2}C transfer (after a REPEATED START) may be to a different slave address.
    The event is detected on any I^{2}C transfer intended for this slave. Note that an I^{2}C address intended for the slave (address matches) will result in an I2C_STOP event independent of whether the I^{2}C address is ACK'd or NACK'd.
  - I^{2}C 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.
  - I^{2}C slave address matched – I^{2}C 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.
  - I^{2}C bus error – Unexpected STOP/START condition is detected
  - I^{2}C restart – When repeated start event is detected

- **I^{2}C 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.
- **I^{2}C 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^{2}C Externally Clocked**
  - Wake up request on address match – Active on incoming slave request (with address match). Only set when EC_AM is ‘1’.
  - I^{2}C STOP detection at the end of each transfer – Only set for a slave request with an address match, in EZ mode, when EC_OP is ‘1’.
  - I^{2}C STOP detection at the end of a write transfer – Activated at the end of a write transfer (I^{2}C 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 mode, when EC_OP is ‘1’.
  - I^{2}C STOP detection at the end of a read transfer – Activated at the end of a read transfer (I^{2}C 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 mode, when EC_OP is ‘1’.

16 Timer, counter, and PWM (TCPWM)

The Timer, Counter, Pulse Width Modulator (TCPWM) block in the PSoC™ 4 MCU uses a 16-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 (times), 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.

16.1 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)
- 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.

16.2 Architecture

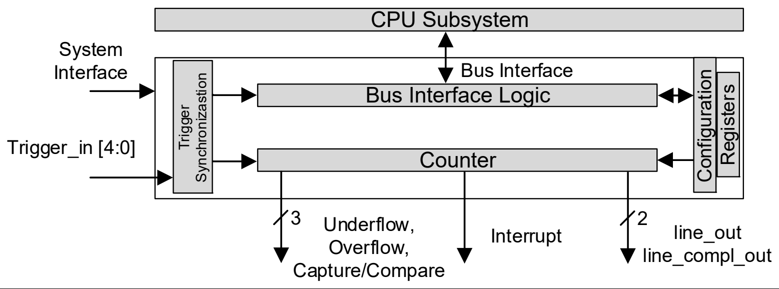


Figure 16-1. TCPWM block diagram

The TCPWM block can contain up to two counters. Each counter can be 16-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.

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 line_out, line_compl_out, 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 179 for more information on all registers required for this block.

16.2.1 Enabling and disabling counters in a TCPWM block

A counter can be enabled by setting the corresponding bit to 1 in the COUNTER_ENABLED field of the control register TCPWM_CTRL. It can be disabled by setting the same bit back to 0.

Note: *The counter must be configured before enabling it. Disabling the counter retains the values in the configuration registers.*

16.2.2 Clocking

The TCPWM receives the HFCLK through the system interface to synchronize all events in the block. The counter enable signal (counter_en), which is generated when the counter is enabled, gates the HFCLK to provide a counter-specific clock (counter_clock). Output triggers (explained later in this chapter) are also synchronized with the HFCLK.

16.2.2.1 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.*

16.2.2.2 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.*

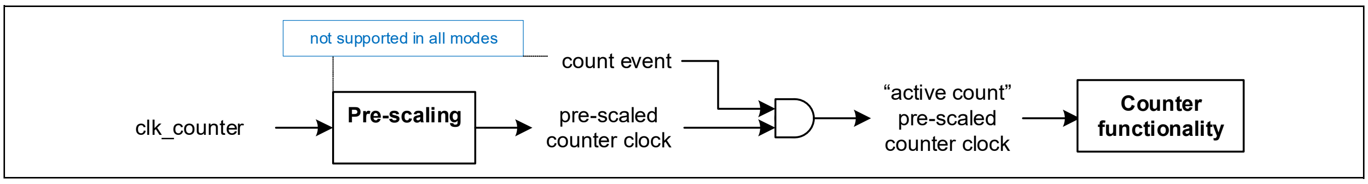


Figure 16-2. Counter clock generation

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.*

16.2.3 Trigger inputs

Each TCPWM block has 14 Trigger_In signals, which come from other on-chip resources. 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"** on page 73 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

Note: *In the TCPWM_CMD register, the COUNTER_START, COUNTER_STOP, COUNTER_RELOAD, and COUNTER_CAPTURE sections are used to trigger start, stop/kill, reload, and capture 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_hf_counter which is the gated version of the clkl_hf and has the same frequency as clk_hf.

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 16-3**).
- 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 16-4**).

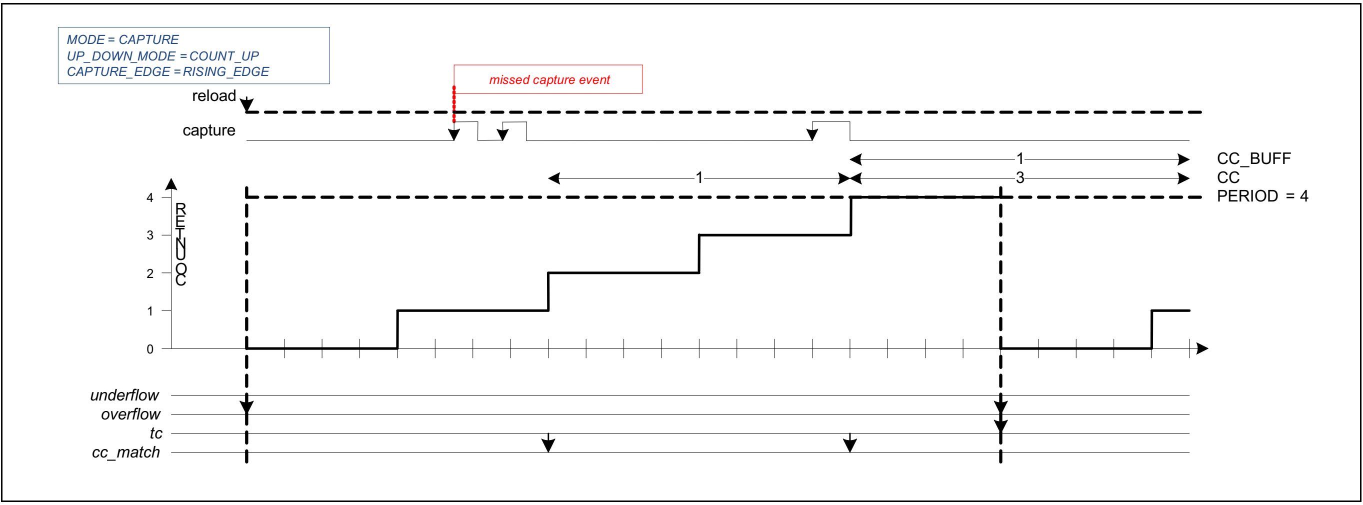


Figure 16-3. Multiple rising edge capture

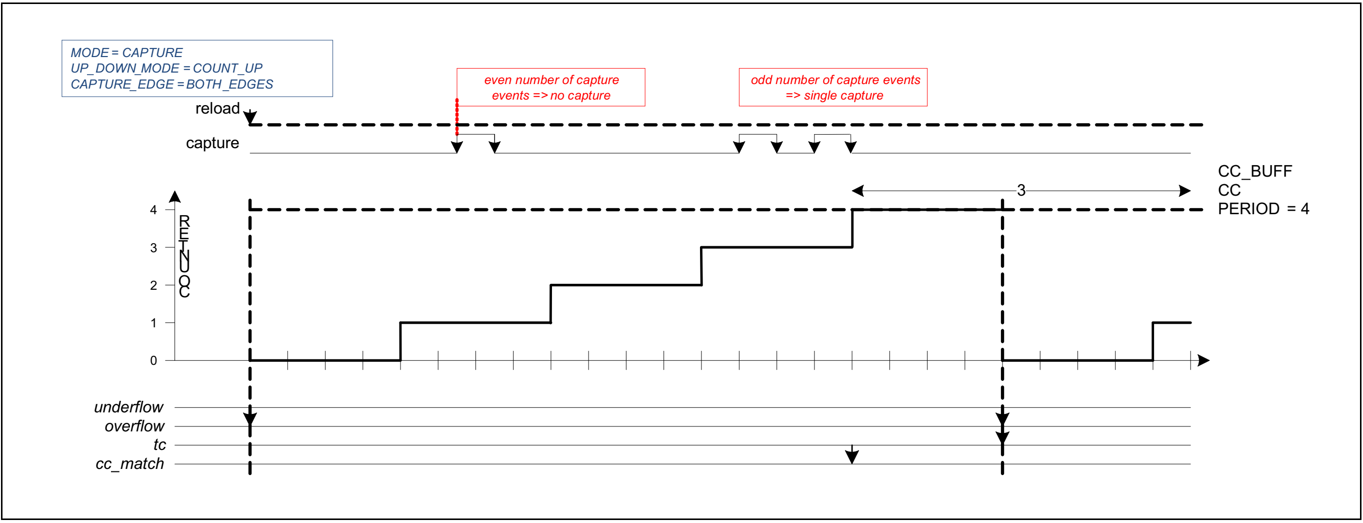

MODE = CAPTURE
UP_DOWN_MODE = COUNT_UP
CAPTURE_EDGE = BOTH_EDGES
even number of capture events => no capture
odd number of capture events => single capture
reload
capture
CC_BUFF
CC
PERIOD = 4
3
COUNTER
4
3
2
1
0
underflow
overflow
tc
cc_match

Figure 16-4. Multiple both edge capture

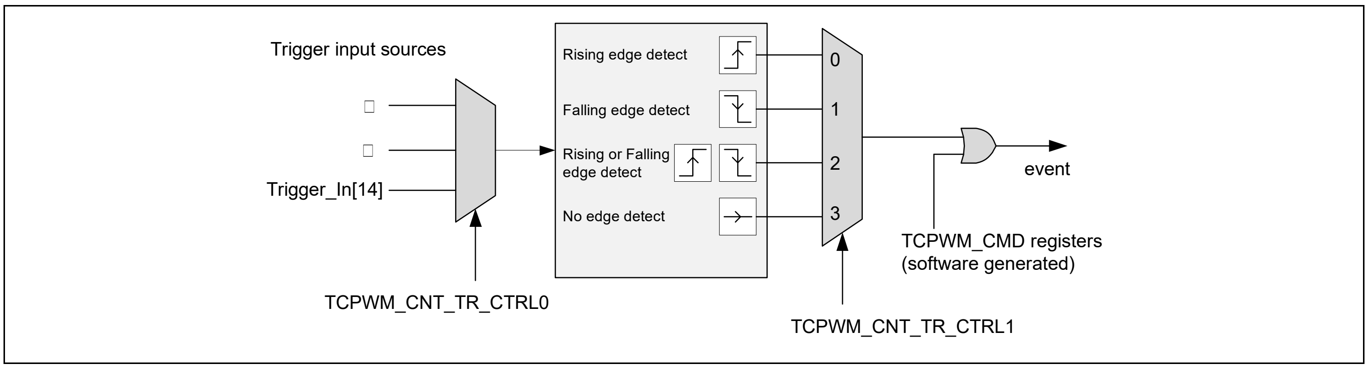


Figure 16-5. TCPWM input events

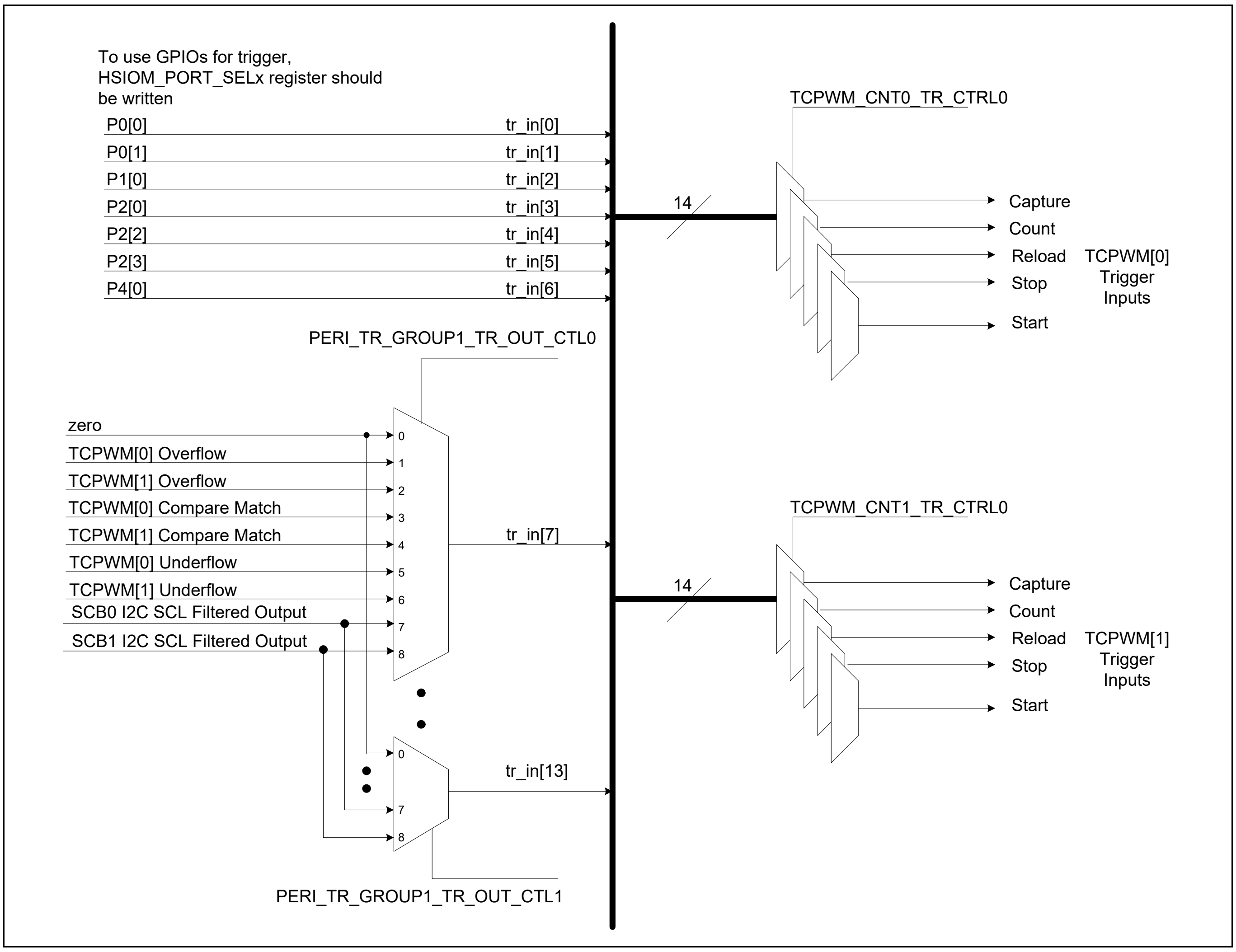


Figure 16-6. TCPWM trigger sources for PSoC™ 4000T MCU

Note:

- All trigger inputs are synchronized to “clk_hf”.
- 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.

16.2.4 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 about to pass (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.*

16.2.5 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 16-1**.

Table 16-1. 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.

16.2.6 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 line_out and line_compl_out, by configuring the TCPWM_CNT_TR_CTRL2 register (see **Table 16-2**).

Table 16-2. 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	

16.2.7 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 16-3** for details.

Table 16-3. 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	In this mode, the power to this block is switched off. Configuration registers will lose their state.

16.3 Operation modes

The counter block can function in six operational modes, as shown in **Table 16-4**. The MODE [26:24] field of the counter control register (TCPWM_CNTx_CTRL) configures the counter in the specific operational mode.

Table 16-4. 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-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 16-5**.

Table 16-5. 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.

16.3.1 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 16-6. 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 16-7. 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 16-8 lists the trigger outputs and the conditions when they are triggered.

Table 16-8. 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 16-8**.*

Table 16-9. 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 16-10. Timer mode PWM outputs

PWM outputs	Description
line_out	Not used
line_compl_out	Not used

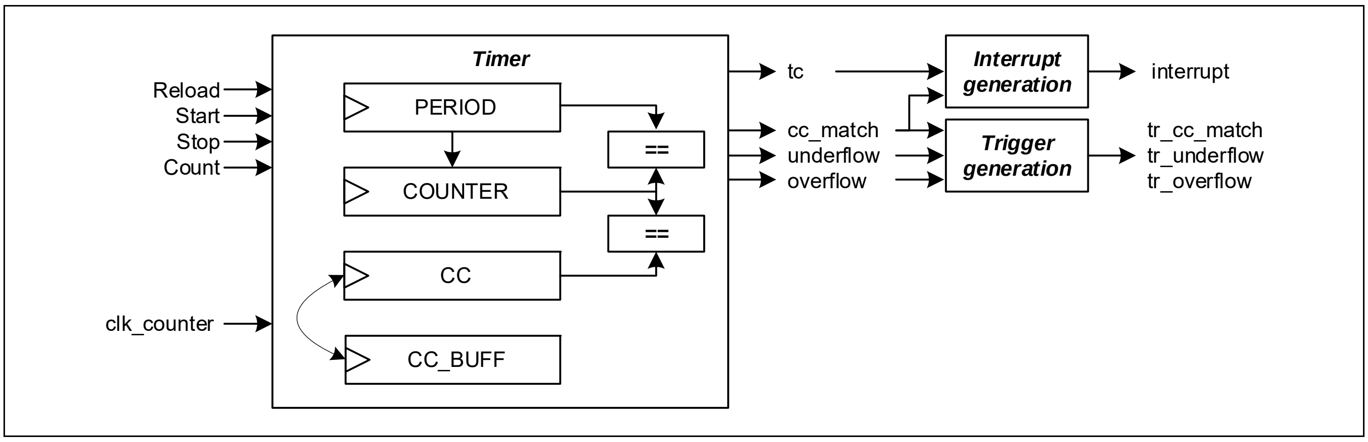


Figure 16-7. Timer functionality

Note:

- The timer functionality uses only PERIOD (and not PERIOD_BUFF).
- Do not write to COUNTER when the counter is running.

Figure 16-8 illustrates a timer in up-counting mode. The counter is initialized (to 0) and started with a software-based reload event.

Note:

- 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.

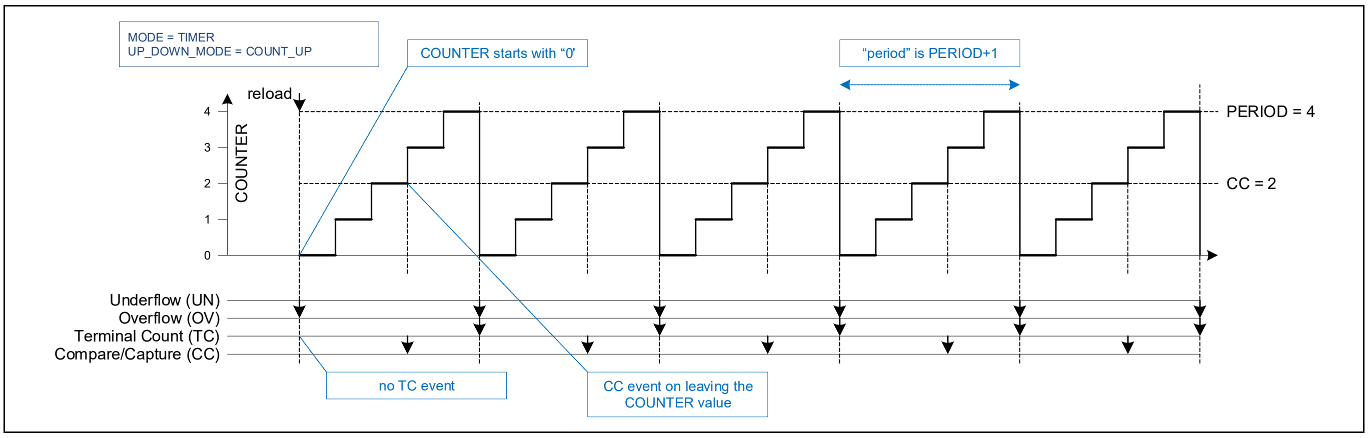


Figure 16-8. Timer in up-counting mode

Figure 16-9 illustrates a timer in "one-shot" operation mode. Note that the counter is stopped on a tc event.

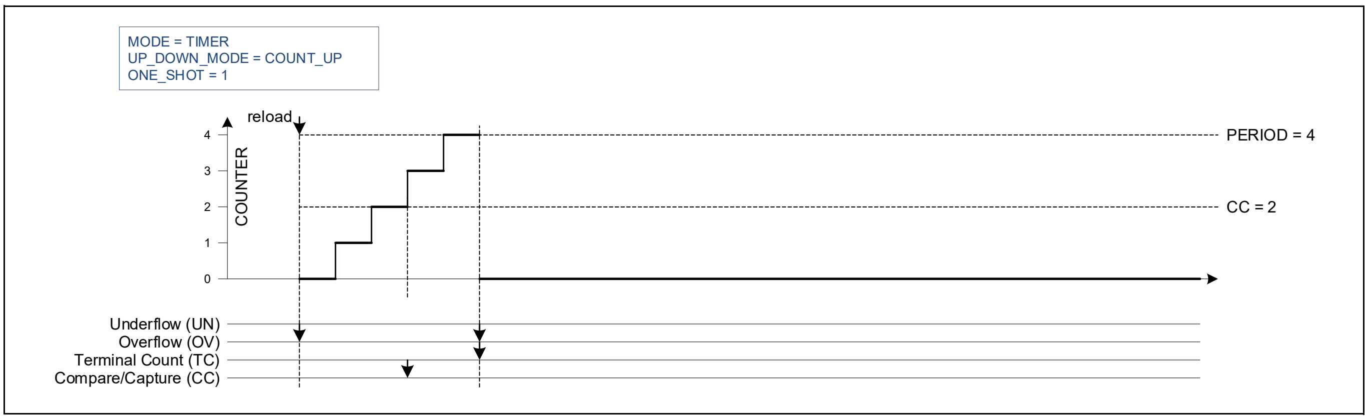


Figure 16-9. Timer in one-shot mode

Figure 16-10 illustrates clock pre-scaling. Note that the counter is only incremented every other counter cycle.

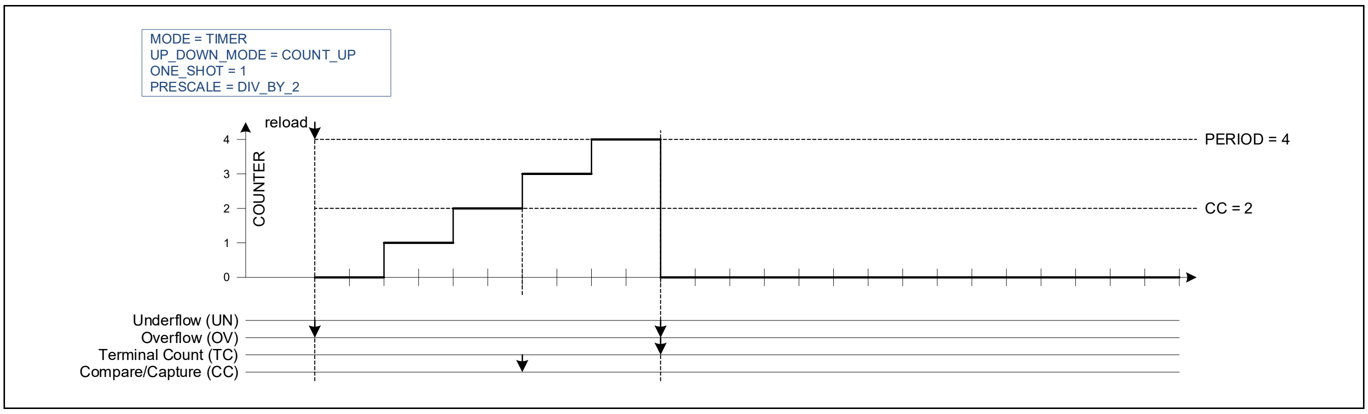


Figure 16-10. Timer clock pre-scaling

Figure 16-11 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).

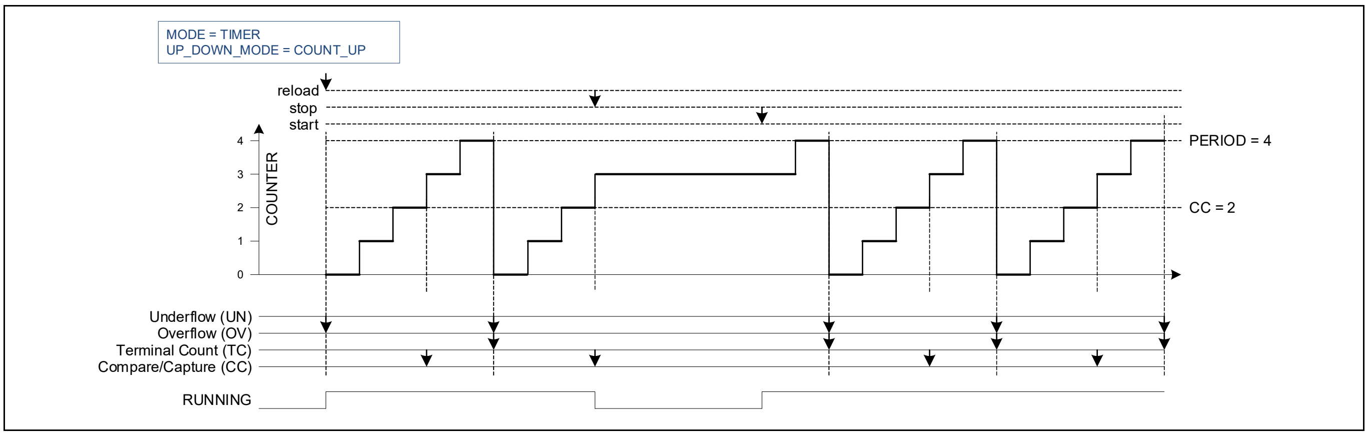


Figure 16-11. Counter start/stopped/continued

Figure 16-12 illustrates a timer that uses both CC and CC_BUFF registers. Note that CC and CC_BUFF are exchanged on a cc_match event.

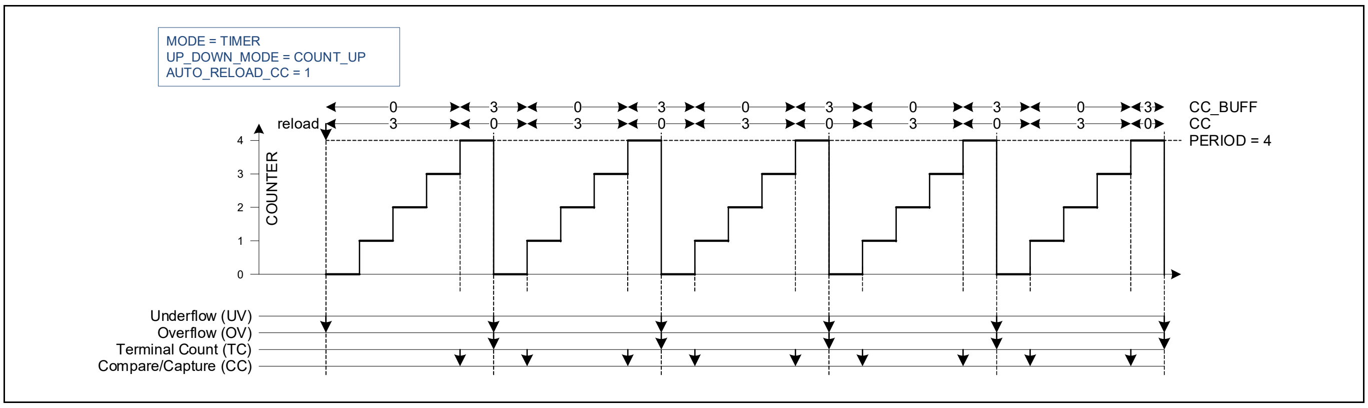


Figure 16-12. Use of CC and CC_BUFF register bits

Figure 16-13 illustrates a timer in down counting mode. The counter is initialized (to PERIOD) and started with a software-based reload event.

Note:

- 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.

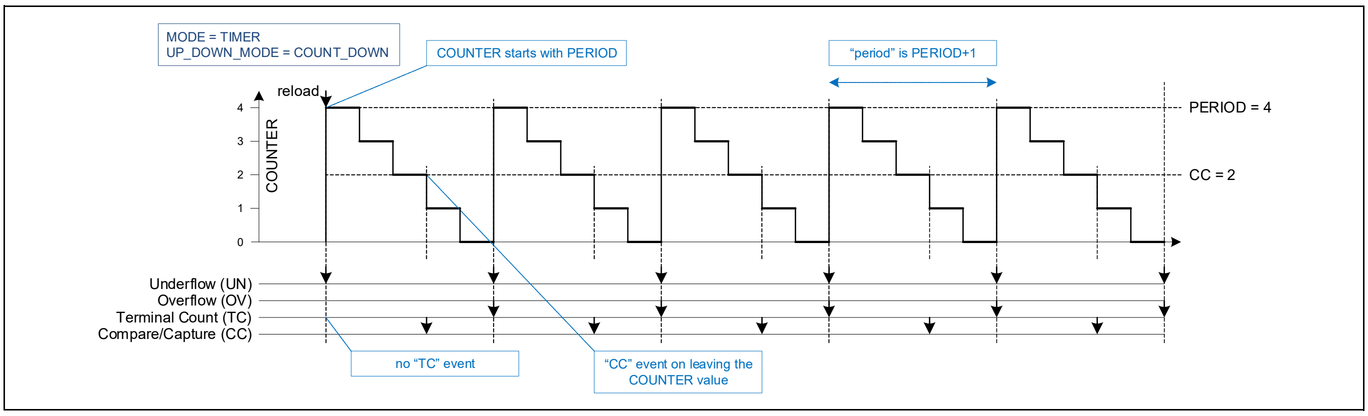


Figure 16-13. Timer in down-counting mode

Figure 16-14 illustrates a timer in up/down counting mode 1. The counter is initialized (to 1) and started with a software-based reload event.

Note:

- 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.

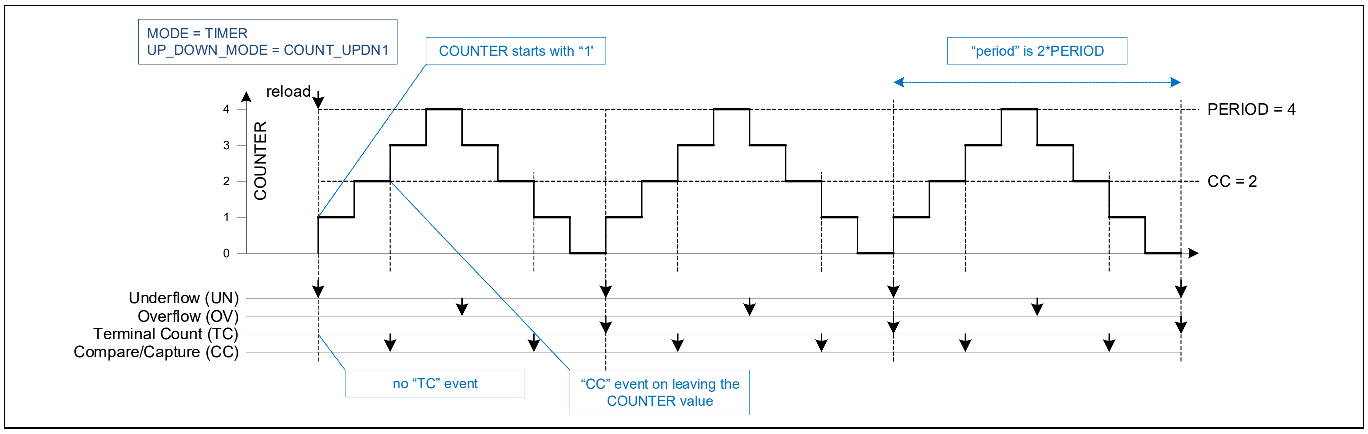


Figure 16-14. Timer in up/down counting mode 1

Figure 16-15 illustrates a timer in up/down counting mode 1, with different CC values.

Note:

- 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).

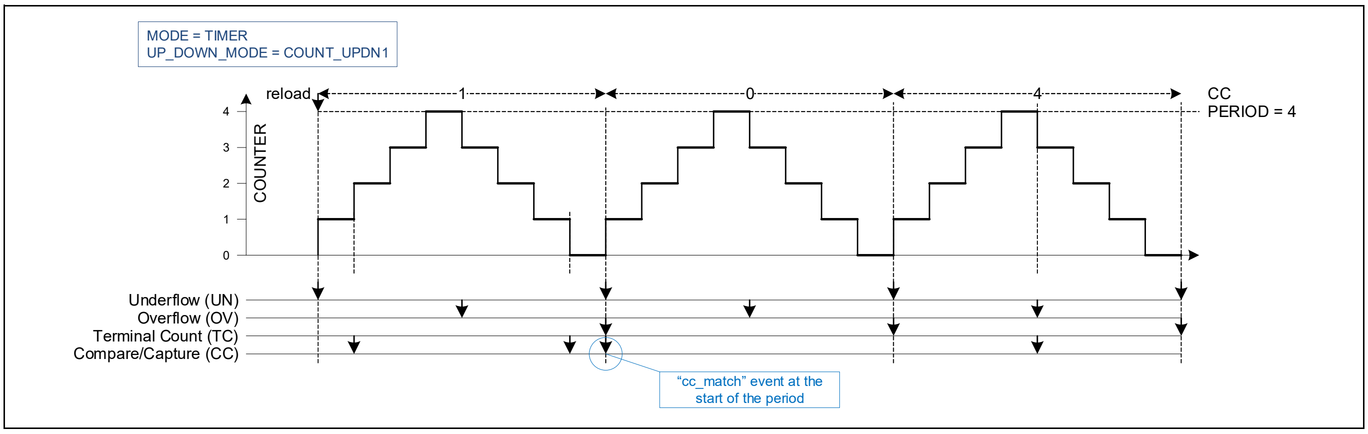


Figure 16-15. Up/down counting mode with different CC values

Figure 16-16 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.

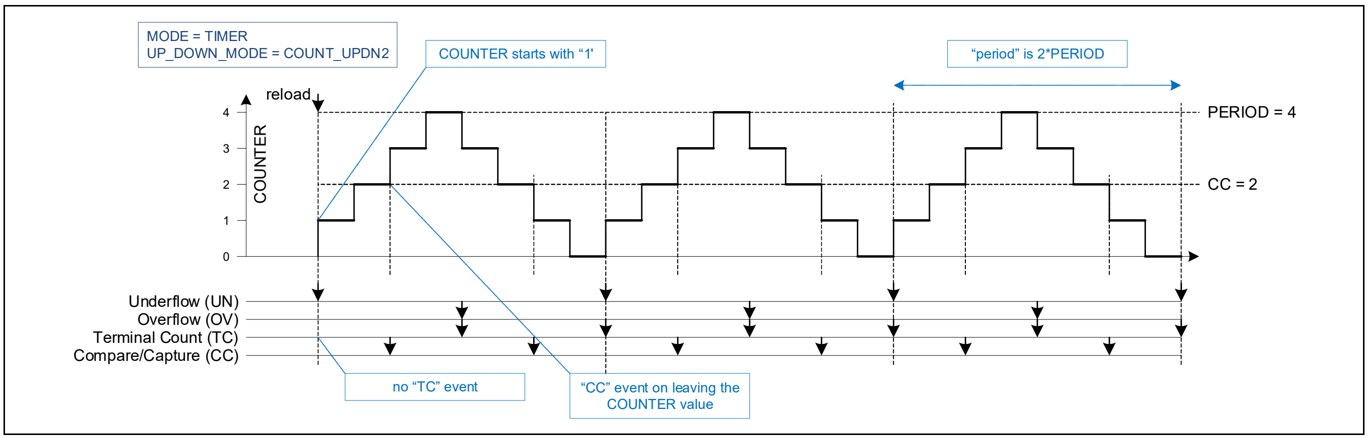


Figure 16-16. Up/down counting mode 2

16.3.1.1 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 '0' to the COUNTER_ENABLED field of the TCPWM_CTRL 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-bit period in the TCPWM_CNT_PERIOD register.
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.
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 COUNTER_ENABLED field of the TCPWM_CTRL register. A reload trigger must be provided through firmware (TCPWM_CMD register) to start the counter if the hardware reload signal is not enabled.

16.3.2 Capture mode

The capture functionality increments/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 (COUNTER_CAPTURE[4:0] field of the TCPWM_CMD register).

Table 16-11. 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 16-12. 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 16-13. 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 16-14. 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 16-15. Capture mode PWM outputs

PWM outputs	Description
line_out	Not used
line_compl_out	Not used

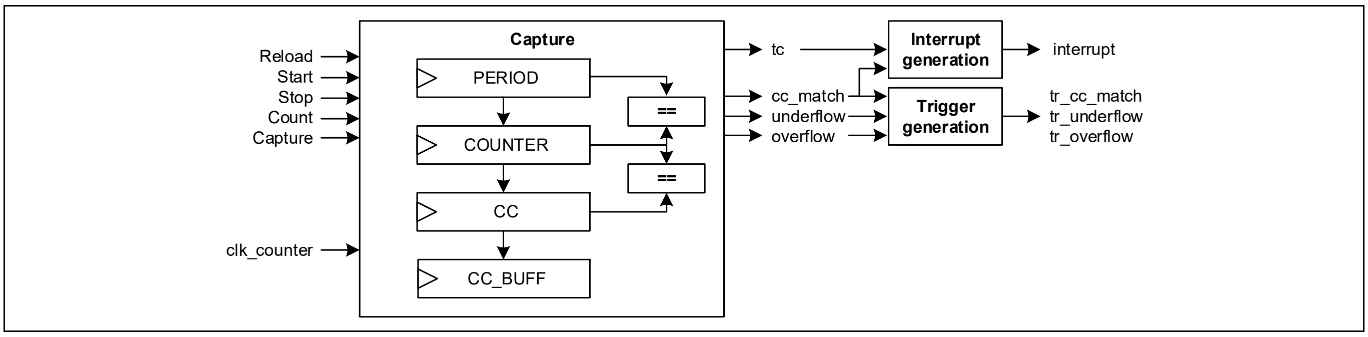


Figure 16-17. Capture functionality

Figure 16-18 illustrates capture behavior in the up counting mode.

Note:

- 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.

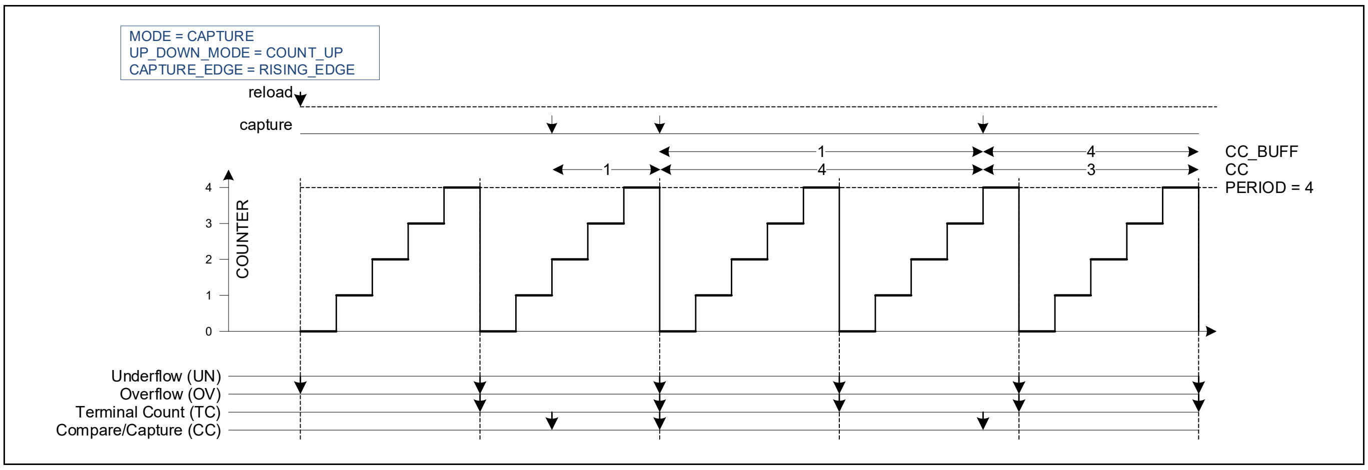


Figure 16-18. Capture in up counting mode

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 16-19**, in which a pre-scaler by a factor of 4 is used.

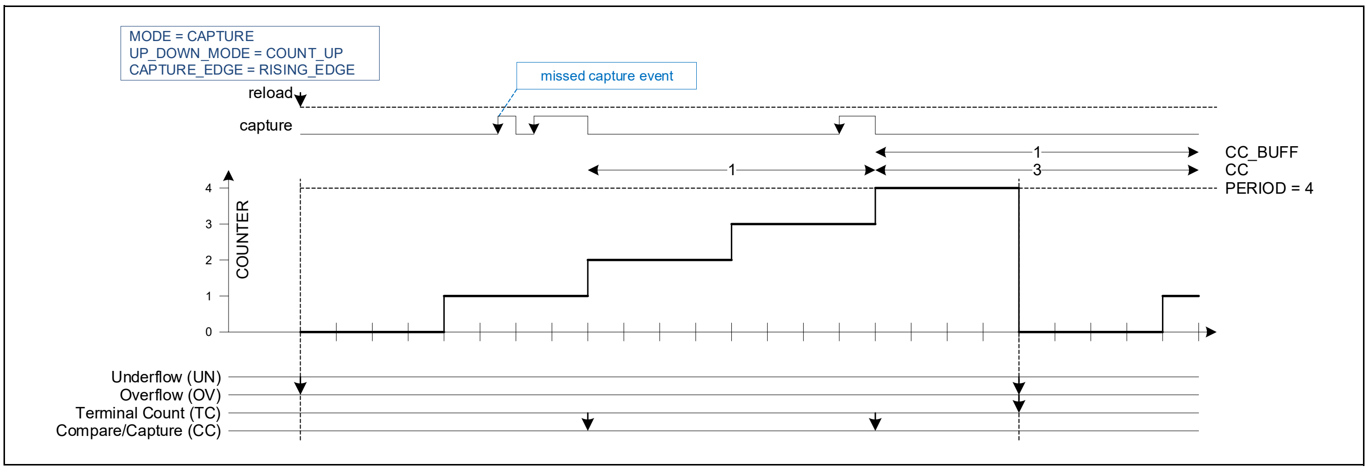


Figure 16-19. Multiple events detected before active-count

16.3.2.1 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 '0' to the COUNTER_ENABLED field of the TCPWM_CTRL 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 COUNTER_ENABLED field of the TCPWM_CTRL register. A reload trigger must be provided through firmware (TCPWM_CMD register) to start the counter if the hardware reload signal is not enabled.

16.3.3 Quadrature decoder mode

Quadrature functionality increments and decrements a counter between 0 and 0xFFFF. 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 16-16**).

Table 16-16. 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 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 16-17. 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 16-18. Quadrature mode trigger output description

Trigger outputs	Description
cc_match (CC)	Counter value COUNTER equals 0 or 0xFFFF or a reload/index event.
Underflow (UN)	Not used
Overflow (OV)	Not used

Table 16-19. Quadrature mode interrupt outputs

Interrupt outputs	Description
cc_match (CC)	Counter value COUNTER equals 0 or 0xFFFF or a reload/index event.
tc	Reload/index event.

Table 16-20. Quadrature mode PWM outputs

PWM outputs	Description
line_out	Not used
line_compl_out	Not used

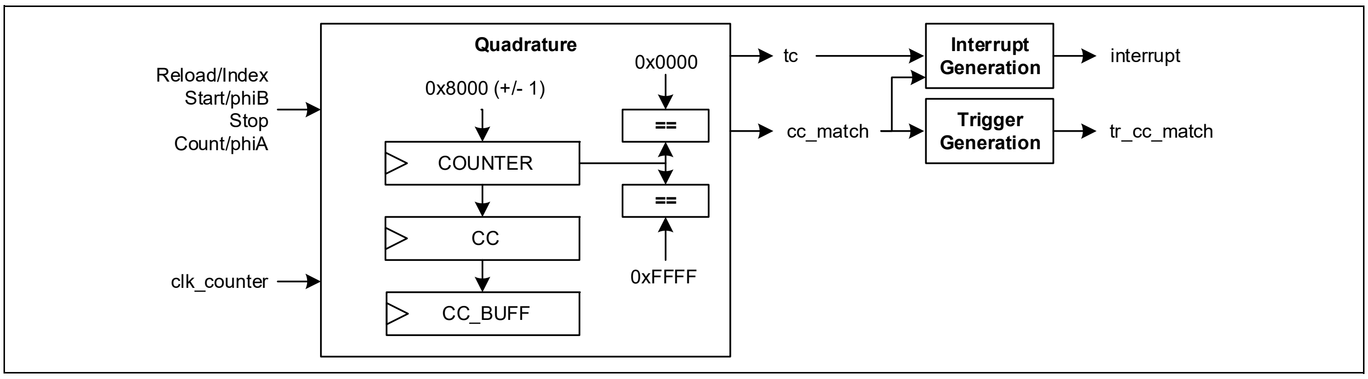


Figure 16-20. Quadrature functionality (16-bit example)

Quadrature functionality is described as follows:

- A software-generated reload event starts quadrature operation. As a result, COUNTER is set to 0x8000 (16-bit), which is the counter midpoint (the COUNTER is set to 0x7FFF if the reload event coincides with a decrement event; the COUNTER is set to 0x8001 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. 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. In addition, the cc_match event is generated.
  - When the counter value COUNTER is 0xFFFF, CC is copied to CC_BUFF, COUNTER (0xFFFF) is copied to CC, and COUNTER is set to 0x8000. In addition, the cc_match event is generated.

Note: *When the counter reaches 0x0000 or 0xFFFF, the counter is automatically set to 0x8000 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 (increment) or 0x8000–1 (decrement).

Counter increments (incr1 event) and decrements (decr1 event) are determined by the quadrature encoding scheme as illustrated by **Figure 16-21**.

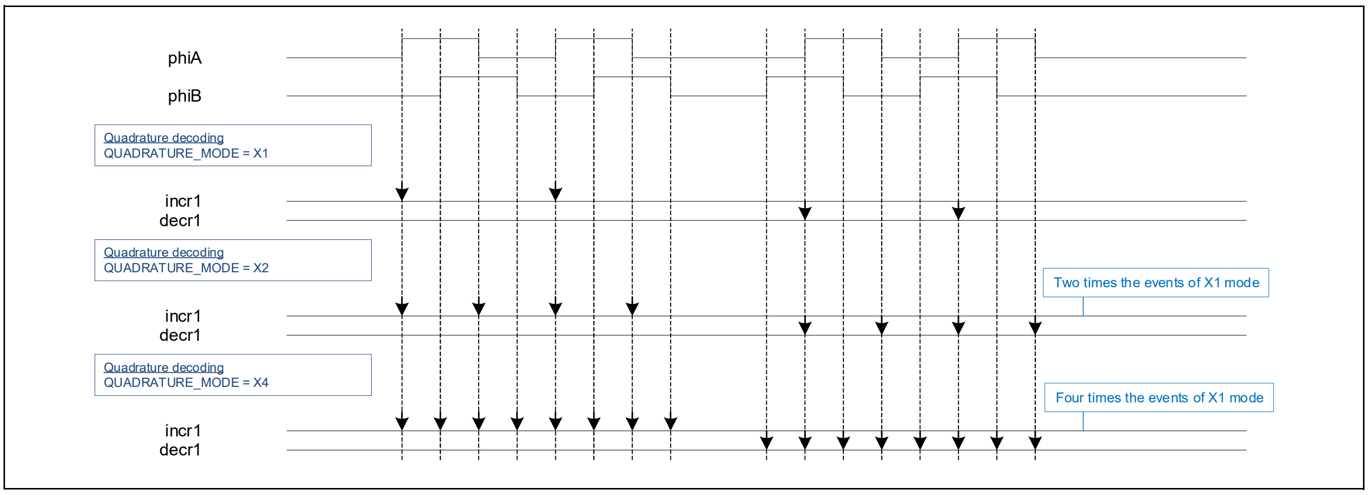


Figure 16-21. Quadrature mode waveforms

Figure 16-22 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.

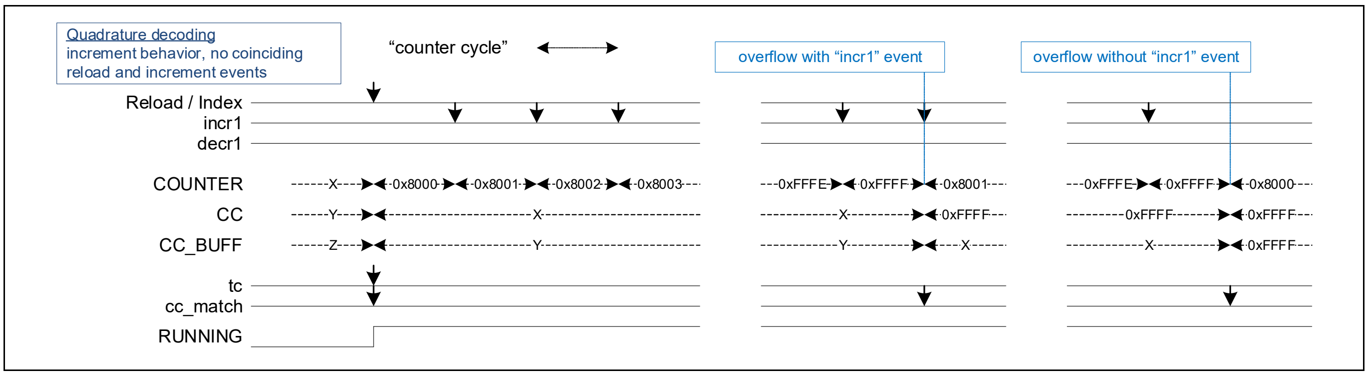


Figure 16-22. Quadrature mode reload/index timing

Figure 16-23 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.

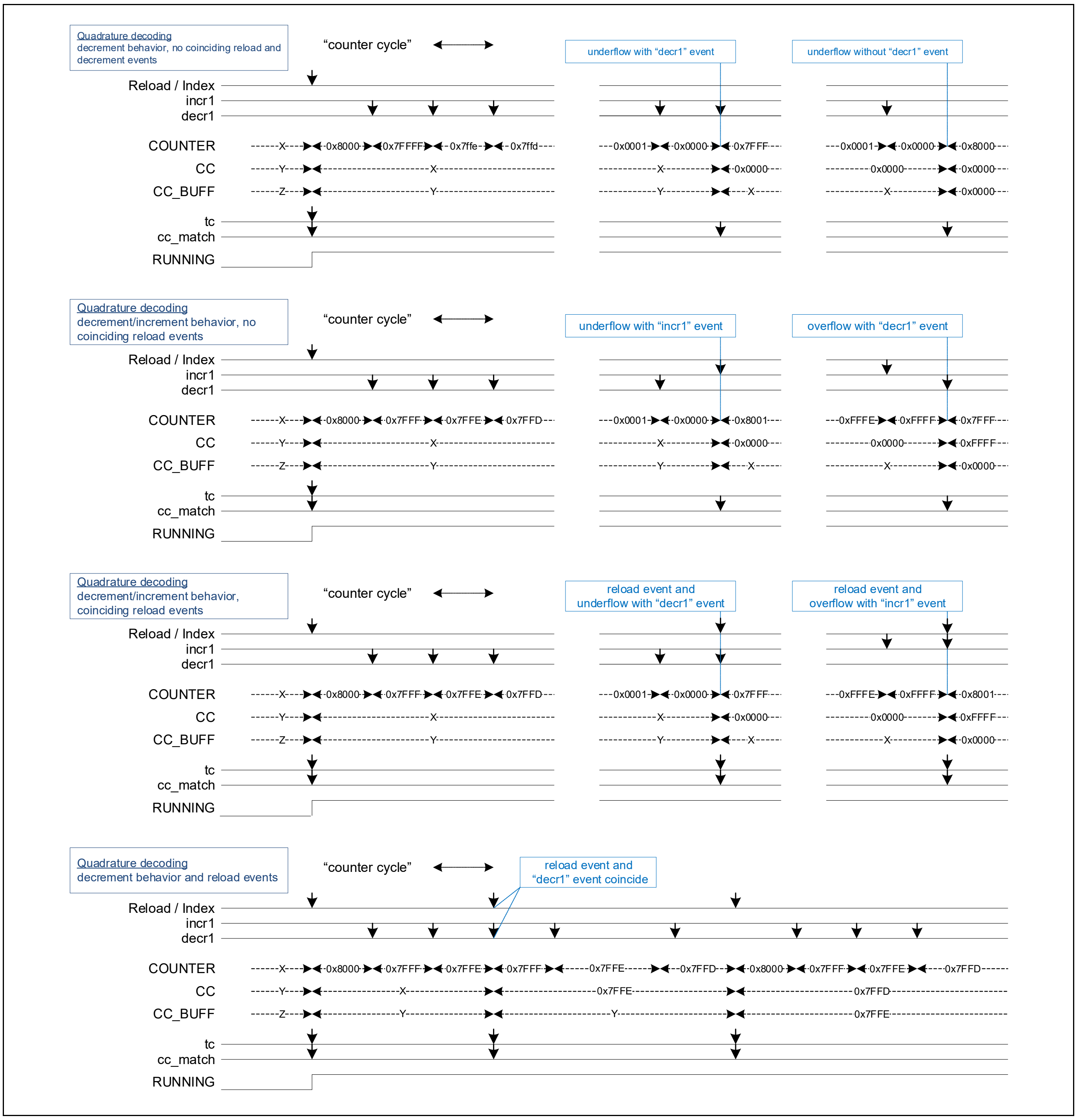

Quadrature decoding
decrement behavior, no coinciding reload and decrement events
"counter cycle"
underflow with "decr1" event
underflow without "decr1" event
Quadrature decoding
decrement/increment behavior, no coinciding reload events
underflow with "incr1" event
overflow with "decr1" event
Quadrature decoding
decrement/increment behavior, coinciding reload events
reload event and underflow with "decr1" event
reload event and overflow with "incr1" event
Quadrature decoding
decrement behavior and reload events
reload event and "decr1" event coincide
Reload / Index
incr1
decr1
COUNTER
CC
CC_BUFF
tc
cc_match
RUNNING
X
Y
Z
0x8000
0x7FFFF
0x7ffe
0x7ffd
0x0001
0x0000
0x7FFF
0x7FFE
0x7FFD
0x8001
0xFFFE
0xFFFF

Figure 16-23. Quadrature mode timing cases

16.3.3.1 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 '0' to the COUNTER_ENABLED field of the TCPWM_CTRL 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 COUNTER_ENABLED field of the TCPWM_CTRL register. A reload trigger must be provided through firmware (TCPWM_CMD register) to start the counter if the hardware reload signal is not enabled.

16.3.4 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 "line_out" and "line_compl_out" are output from the PWM.

Table 16-21. 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.

Table 16-21. PWM mode trigger input description

Trigger inputs	Usage
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 16-22. 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.

Table 16-22. PWM mode supported features

Supported features	Description
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 16-23. 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 16-24. 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 16-25. PWM mode PWM outputs

PWM outputs	Description
line_out	PWM output.
line_compl_out	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.

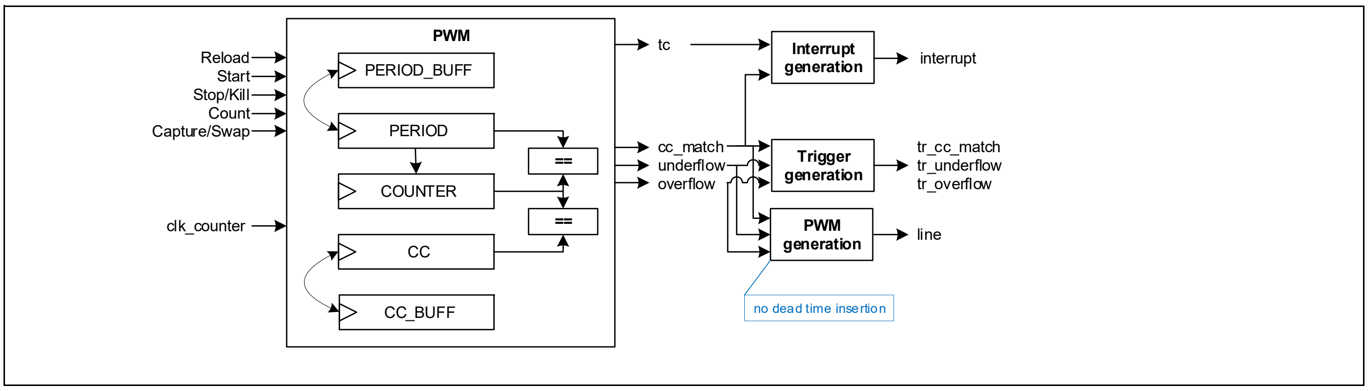


Figure 16-24. PWM mode functionality

The generation of PWM output signals is a multi-step process and is illustrated in **Figure 16-25**. 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 line.

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 line will be SET to '1' first and then CLEARED to '0' immediately after. This can be seen in **Figure 16-27**.*

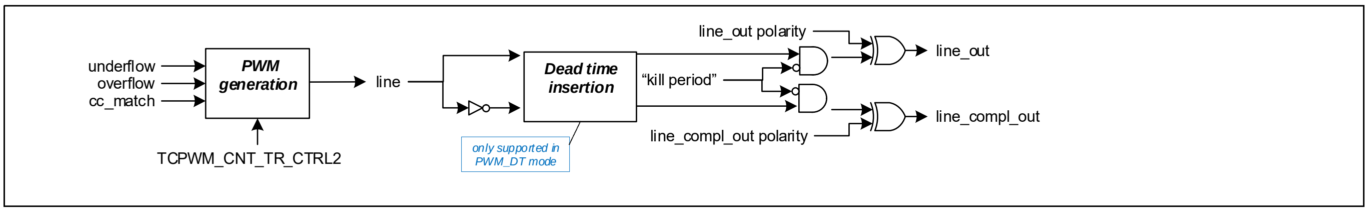


Figure 16-25. PWM output generation

line_out polarity and line_compl_out polarity as seen in **Figure 16-25**, allow the PWM outputs to be inverted. line_out polarity is controlled through CTRL.QUADRATURE_MODE[0] and line_compl_out 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 16-26 illustrates a PWM in up counting mode. The counter is initialized (to 0) and started with a software-based reload event.

Note:

- 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.

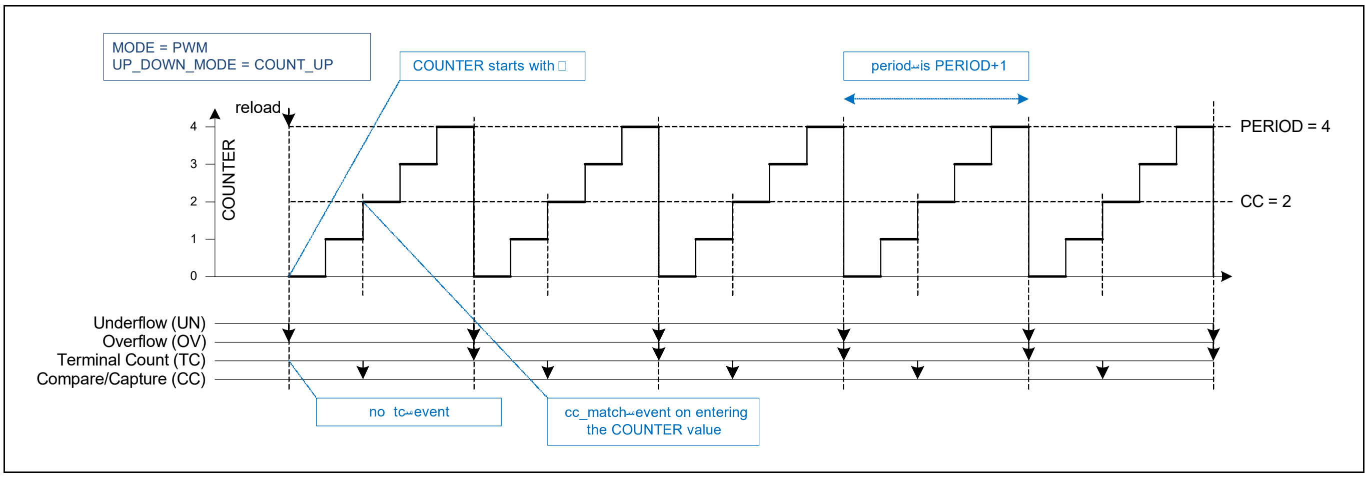


Figure 16-26. PWM in up counting mode

Figure 16-27 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).

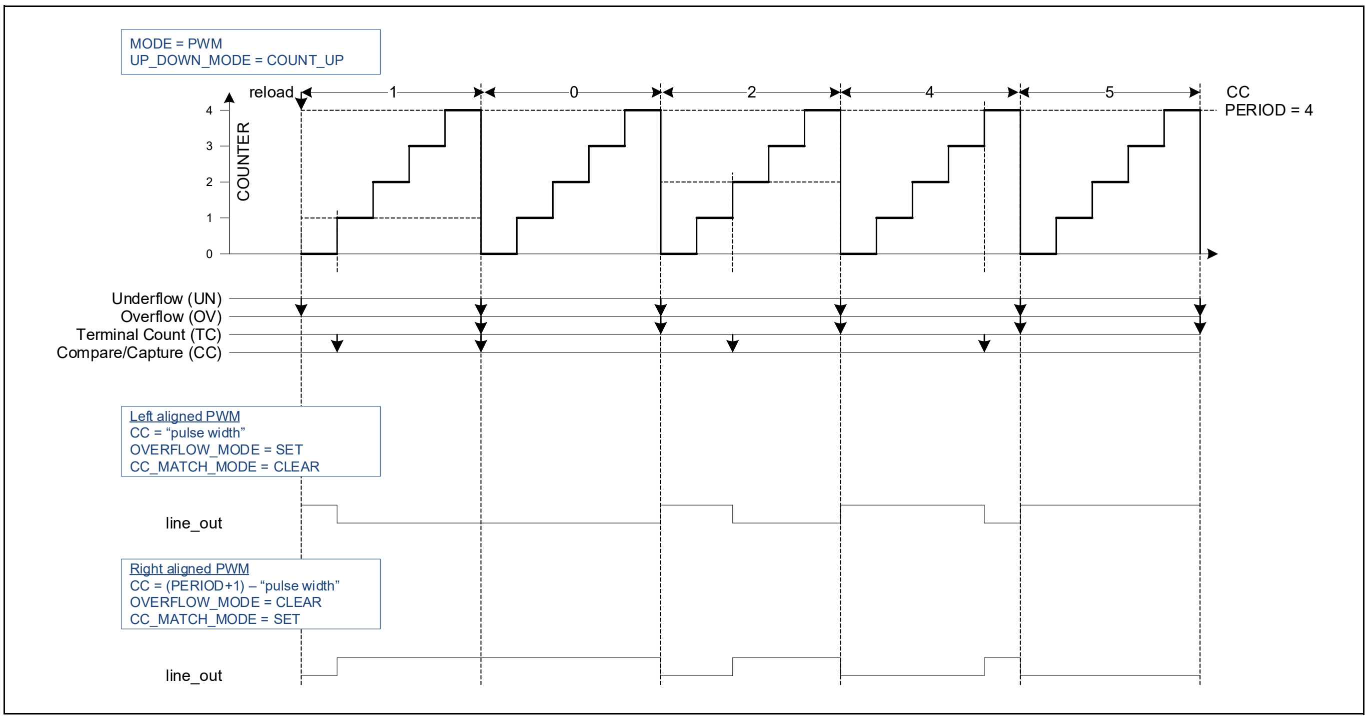


Figure 16-27. PWM left- and right-aligned outputs

Figure 16-28 illustrates a PWM in down counting mode. The counter is initialized (to PERIOD) and started with a software-based reload event.

Note:

- 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.

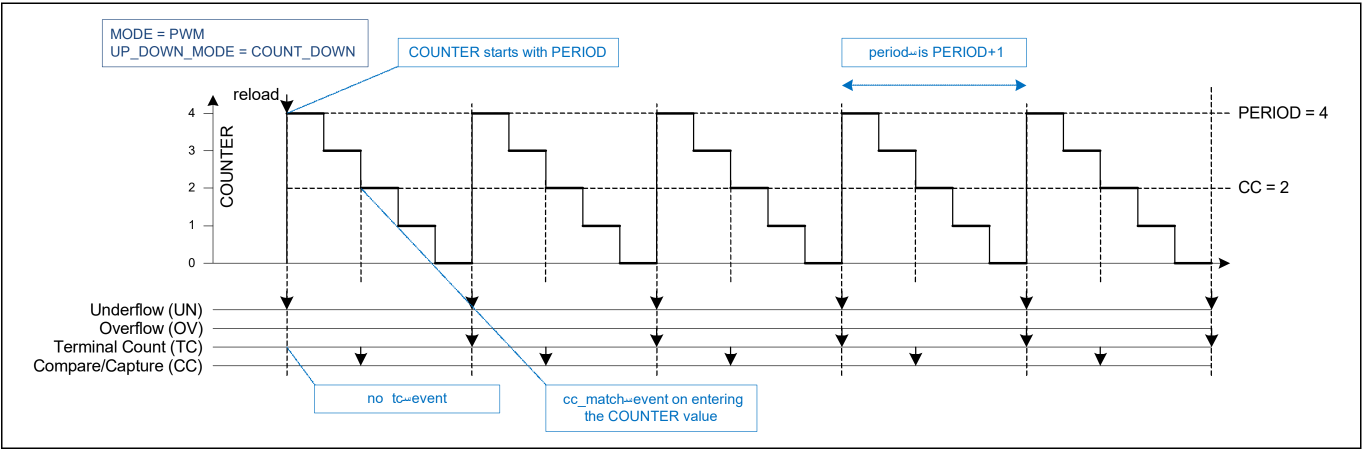


Figure 16-28. PWM in Down Counting Mode

Figure 16-29 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.

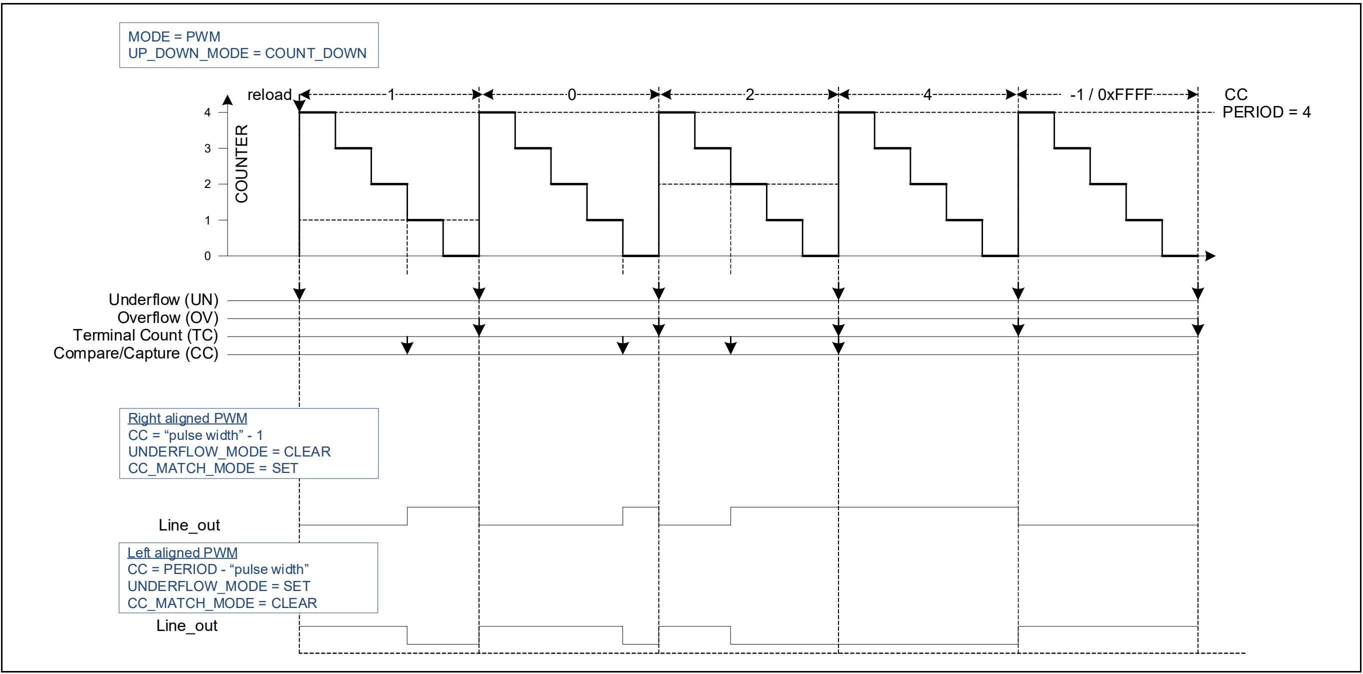


Figure 16-29. Right- and left-aligned down counting PWM

Figure 16-30 illustrates a PWM in up/down counting mode. The counter is initialized (to 1) and started with a software-based reload event.

Note:

- 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.

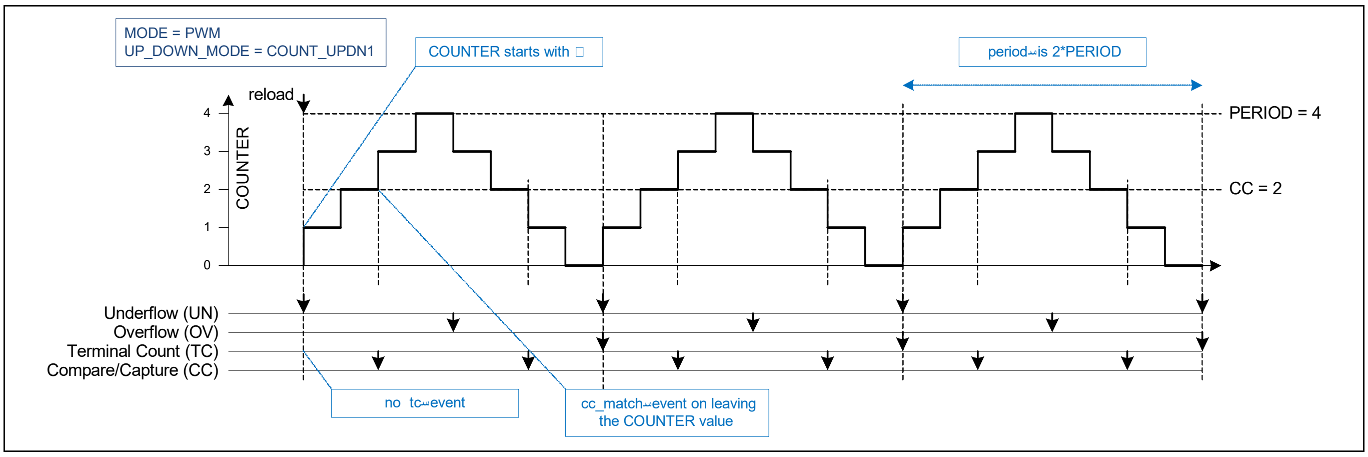


Figure 16-30. Up/down counting PWM

Figure 16-31 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.

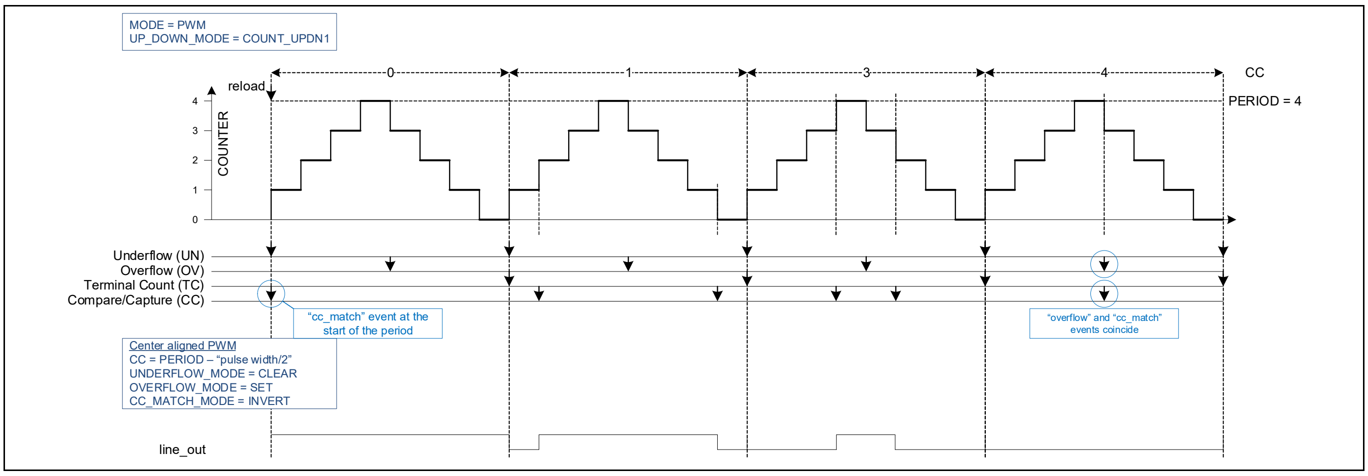


Figure 16-31. Up/down counting center-aligned PWM

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 16-32, **Figure 16-33**, and **Figure 16-34** illustrate the above three modes.

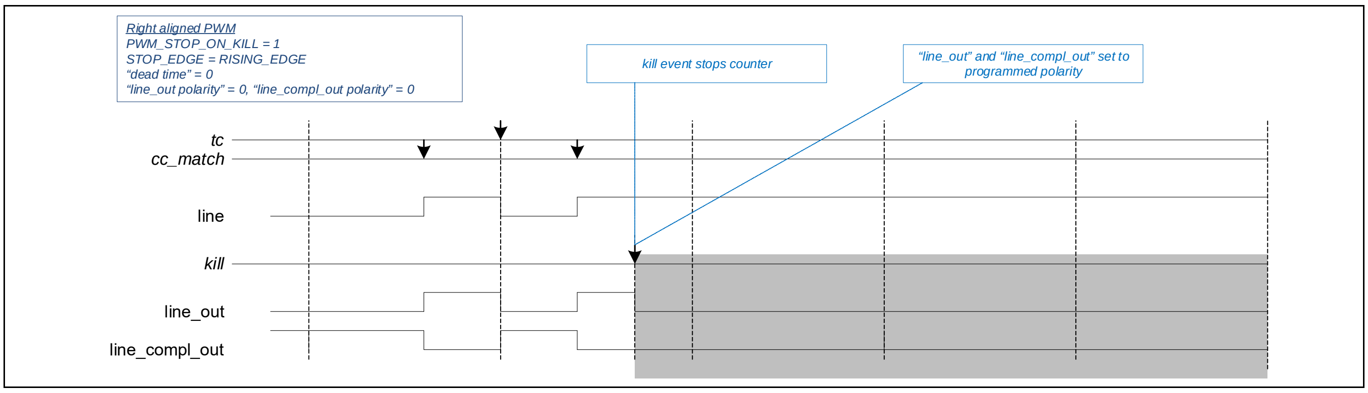


Figure 16-32. PWM stop on kill

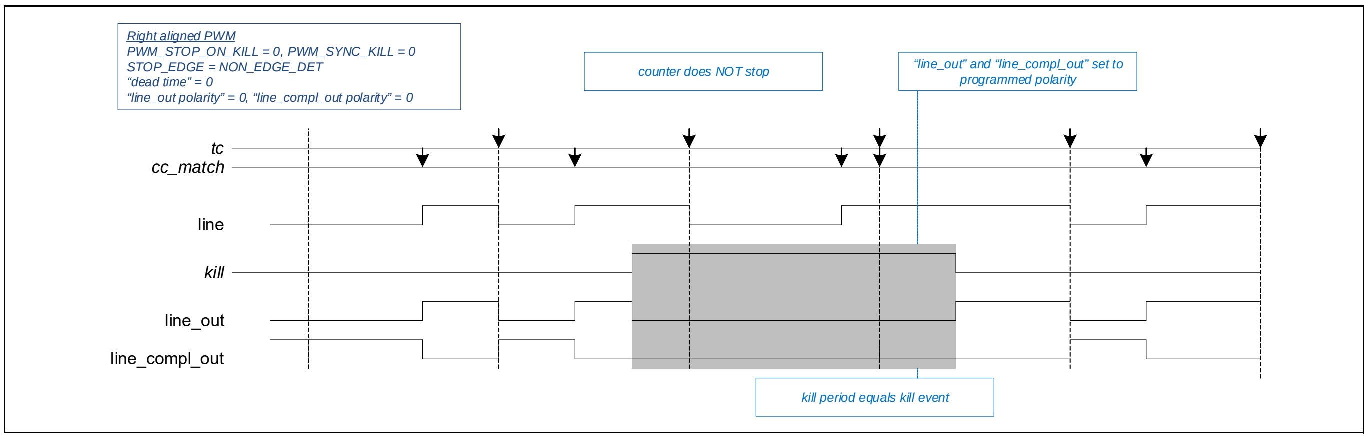

Right aligned PWM
PWM_STOP_ON_KILL = 0, PWM_SYNC_KILL = 0
STOP_EDGE = NON_EDGE_DET
"dead time" = 0
"line_out polarity" = 0, "line_compl_out polarity" = 0
counter does NOT stop
"line_out" and "line_compl_out" set to programmed polarity
tc
cc_match
line
kill
line_out
line_compl_out
kill period equals kill event

Figure 16-33. PWM async kill

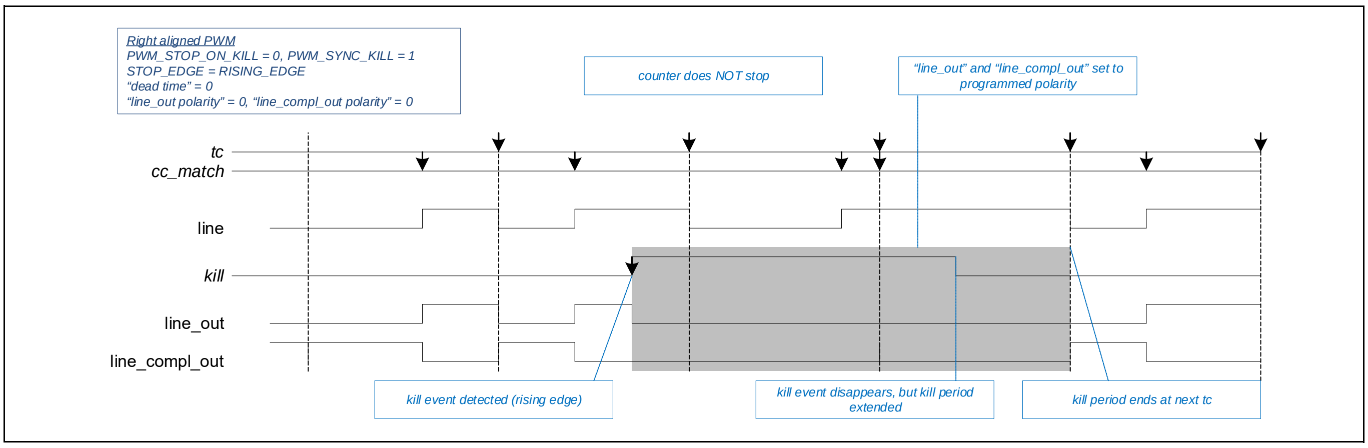

Right aligned PWM
PWM_STOP_ON_KILL = 0, PWM_SYNC_KILL = 1
STOP_EDGE = RISING_EDGE
"dead time" = 0
"line_out polarity" = 0, "line_compl_out polarity" = 0
counter does NOT stop
"line_out" and "line_compl_out" set to programmed polarity
tc
cc_match
line
kill
line_out
line_compl_out
kill event detected (rising edge)
kill event disappears, but kill period extended
kill period ends at next tc

Figure 16-34. PWM sync kill

Figure 16-35 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.

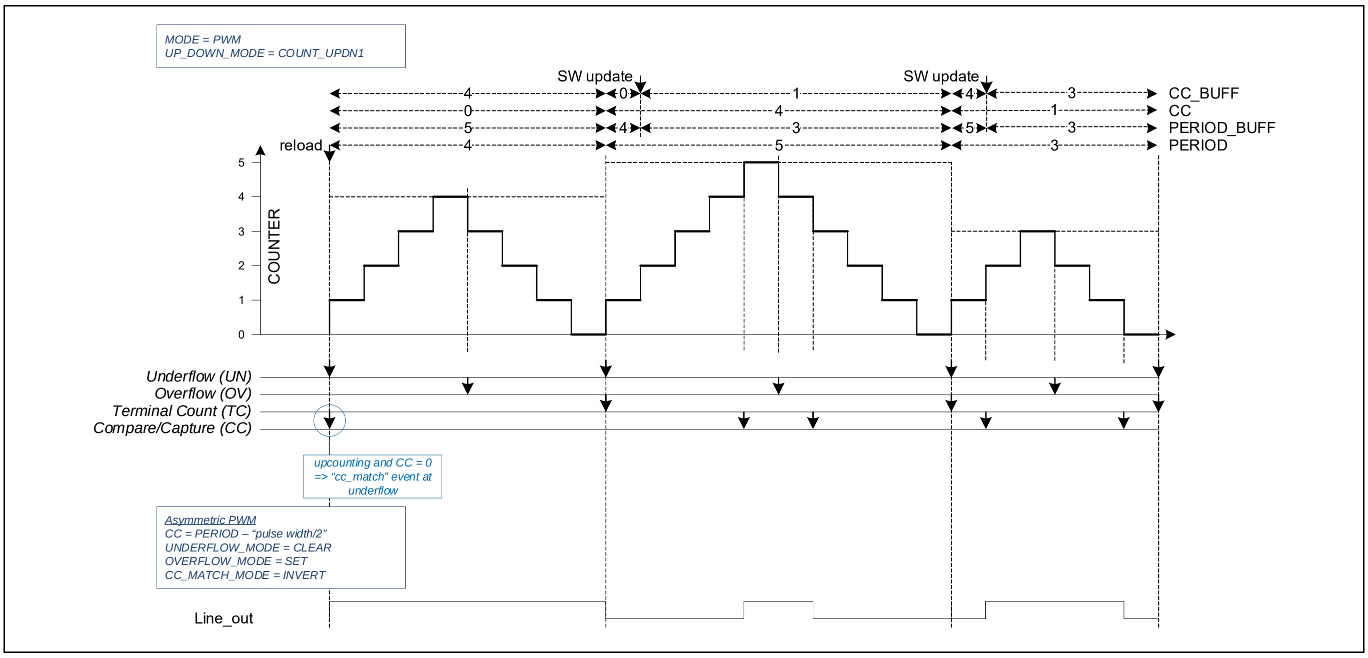


Figure 16-35. PWM mode CC swap event

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.

Figure 16-25 uses CTRL.QUADRATURE_MODE[0] for "line_out" polarity and CTRL.QUADRATURE_MODE[1] for "line_compl_out" polarity. **Figure 16-36** illustrates how the polarity settings control the PWM output signals "line_out" and "line_compl_out".

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.*

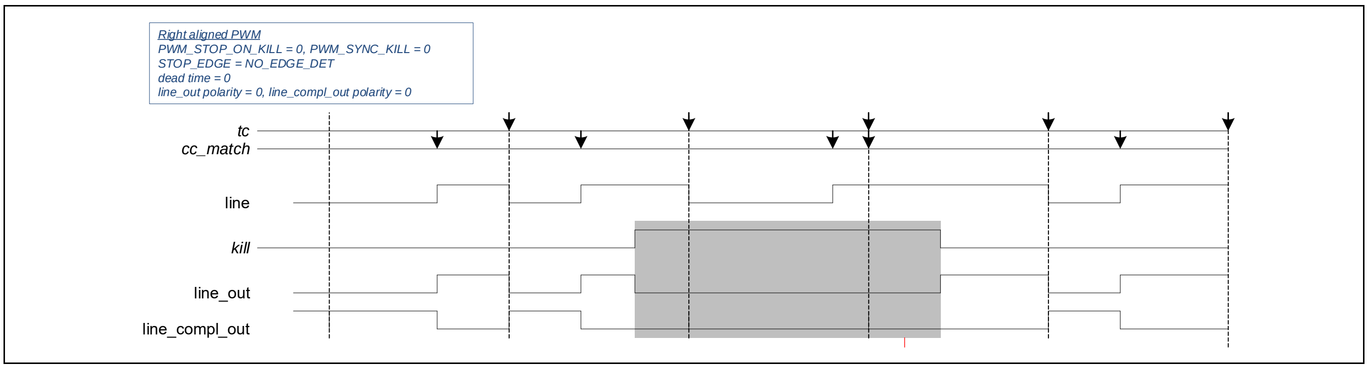


Figure 16-36. PWM Outputs When Killed

16.3.4.1 Asymmetric PWM

This PWM mode supports the generation of an asymmetric PWM. For an asymmetric PWM, the line 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 16-37 illustrates how the COUNT_UPDN2 mode is used to generate an asymmetric PWM.

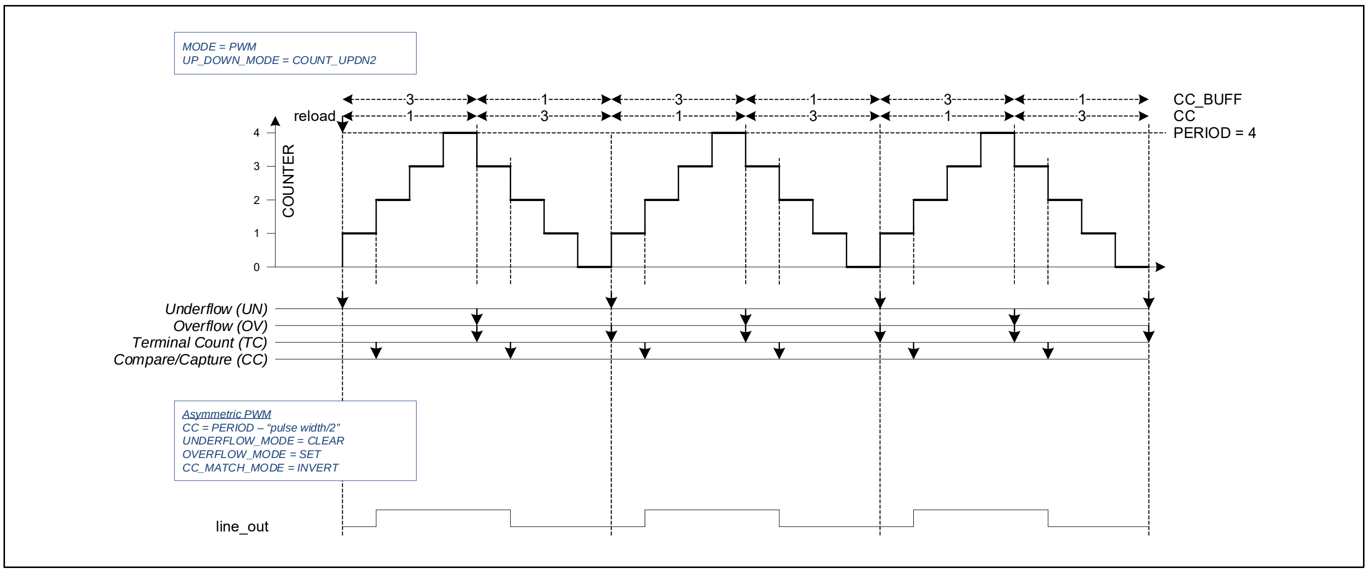


Figure 16-37. Asymmetric PWM

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 16-38 illustrates how the COUNT_UPDN2 mode is used to generate an asymmetric PWM.

Note:

- 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 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).

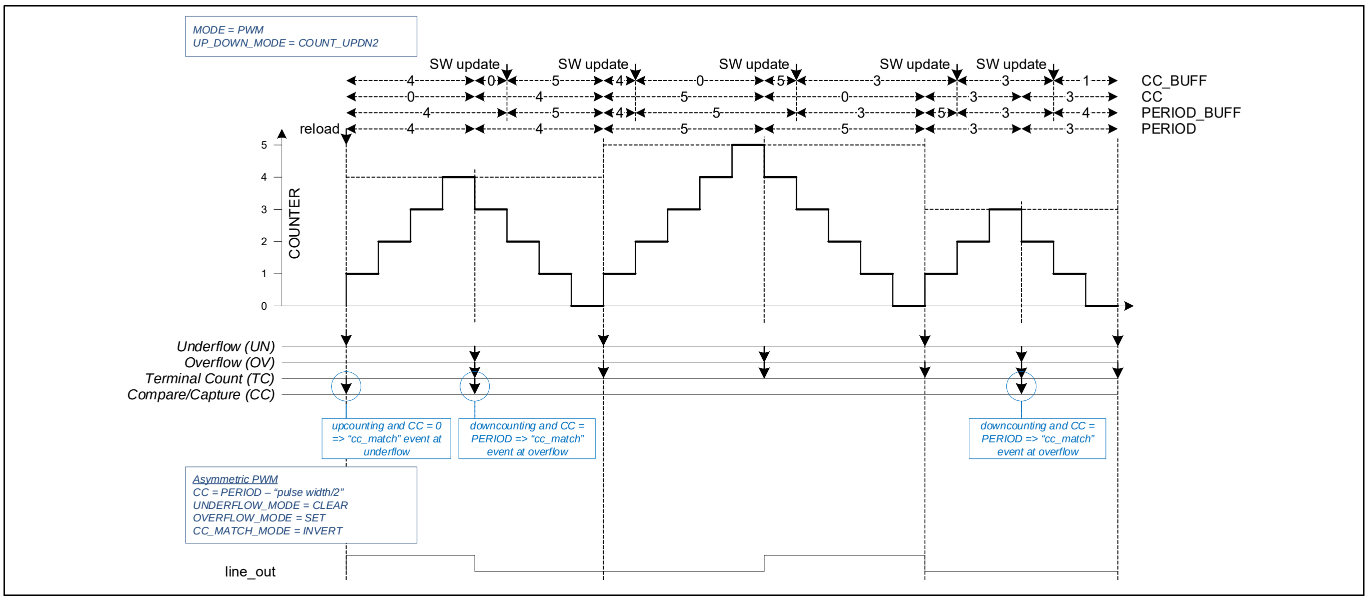

MODE = PWM
UP_DOWN_MODE = COUNT_UPDN2
SW update
CC_BUFF
CC
PERIOD_BUFF
PERIOD
reload
COUNTER
Underflow (UN)
Overflow (OV)
Terminal Count (TC)
Compare/Capture (CC)
upcounting and CC = 0 => "cc_match" event at underflow
downcounting and CC = PERIOD => "cc_match" event at overflow
downcounting and CC = PERIOD => "cc_match" event at overflow
Asymmetric PWM
CC = PERIOD – "pulse width/2"
UNDERFLOW_MODE = CLEAR
OVERFLOW_MODE = SET
CC_MATCH_MODE = INVERT
line_out

Figure 16-38. Asymmetric PWM when Compare = 0 or Period

16.3.4.2 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 '0' to the COUNTER_ENABLED field of the TCPWM_CTRL 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-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-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. line_out and line_compl_out 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 COUNTER_ENABLED field of the TCPWM_CTRL register. A reload trigger must be provided through firmware (TCPWM_CMD register) to start the counter if the hardware reload signal is not enabled.

16.3.5 Pulse width modulation with dead time mode

Dead time is used to delay the transitions of both "line" and "line_n" signals. It separates the transition edges of these two signals by a specified time interval. A maximum dead time of 255 clocks can be generated using this feature. 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.

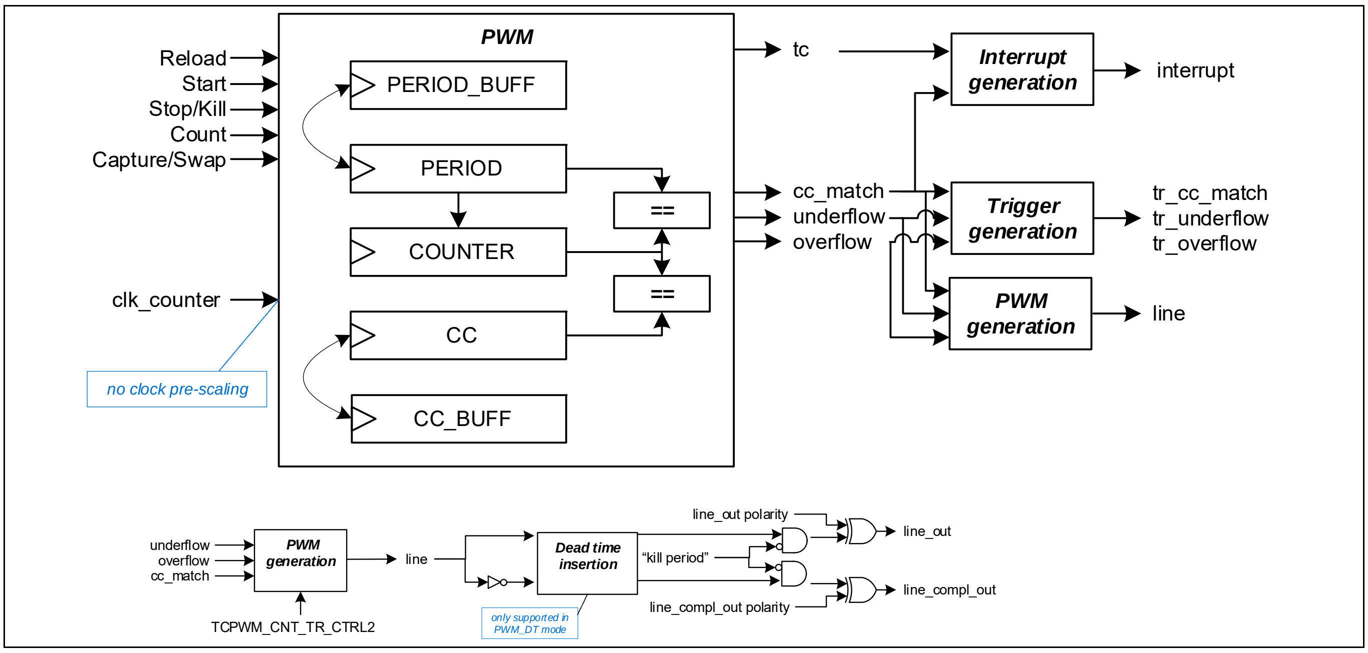


Figure 16-39. PWM with dead time functionality

Dead time insertion is a step that operates on a preliminary PWM output signal line, as illustrated in **Figure 16-39**. **Figure 16-40** illustrates dead time insertion for different dead times and different output signal polarity settings.

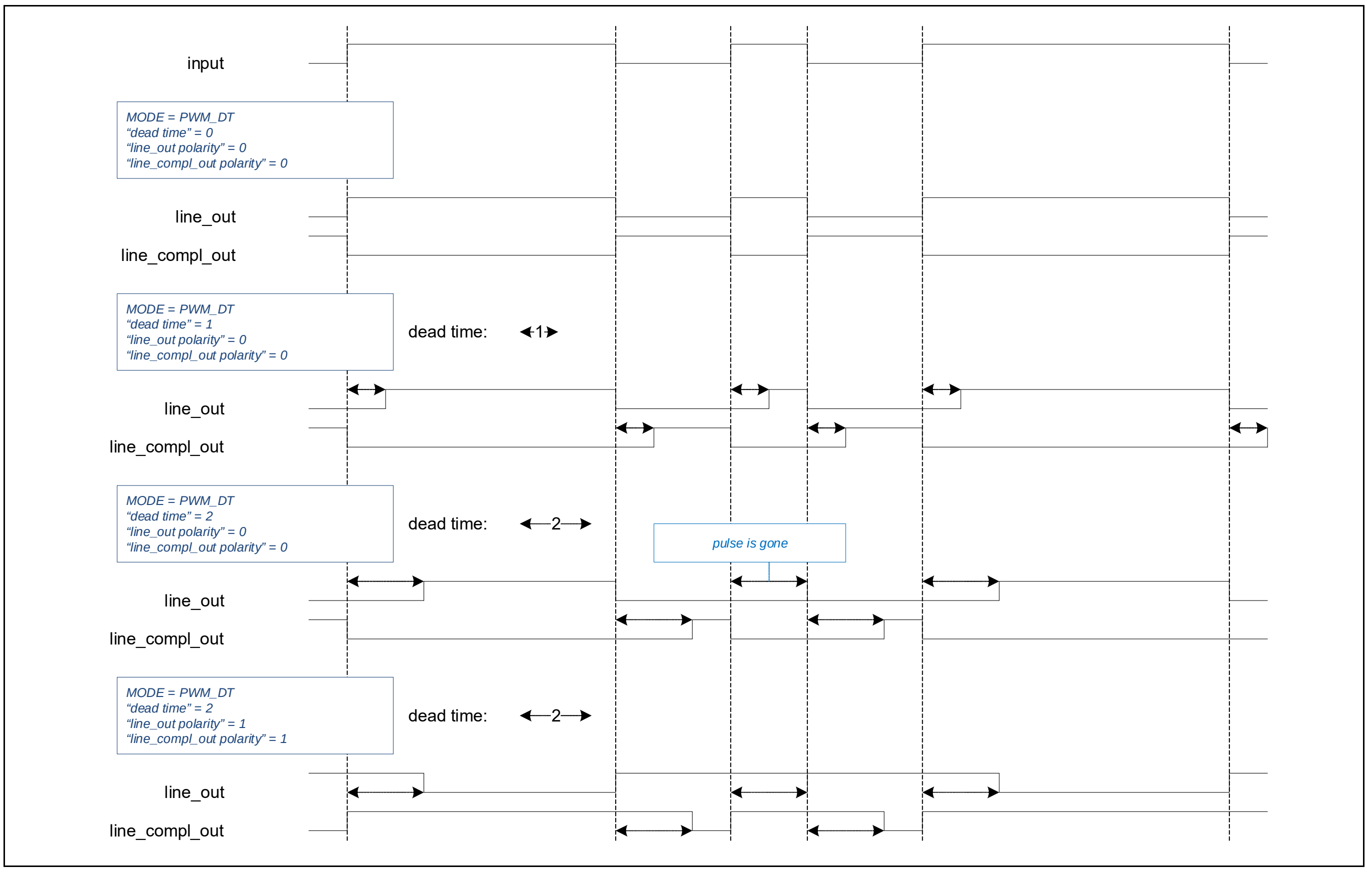


Figure 16-40. Dead-time timing

Figure 16-41 illustrates how the polarity settings and stop/kill functionality combined control the PWM output signals "line_out" and "line_compl_out".

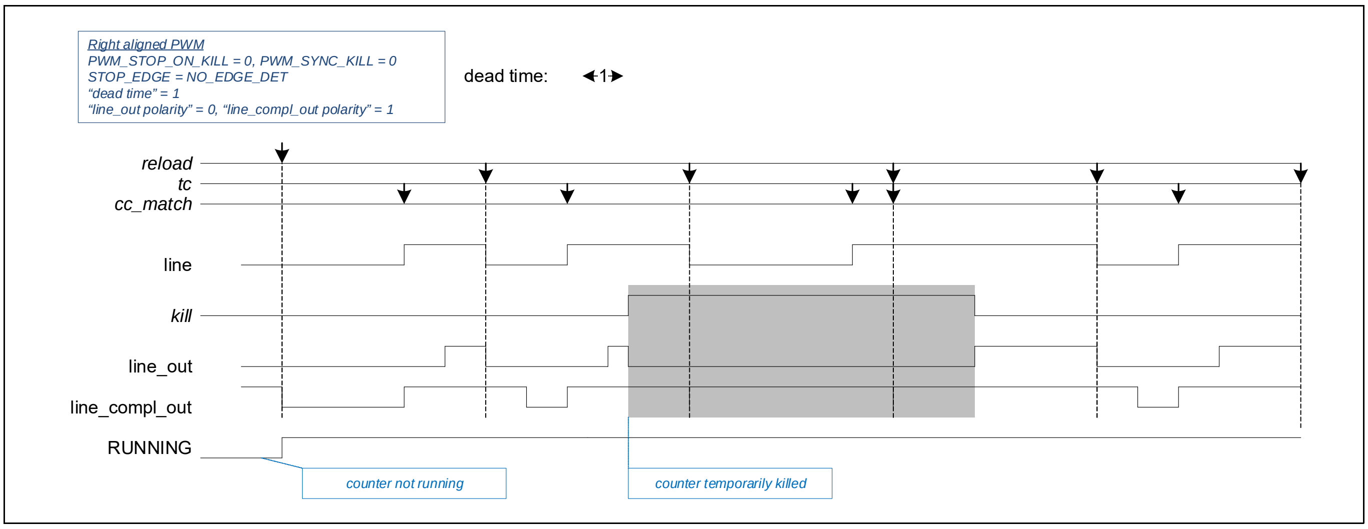


Figure 16-41. Dead time and kill

16.3.5.1 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 ‘0’ to the COUNTER_ENABLED field of the TCPWM_CTRL 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-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-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. line_out and line_compl_out 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 COUNTER_ENABLED field of the TCPWM_CTRL register. A start trigger must be provided through firmware (TCPWM_CMD register) to start the counter if the hardware start signal is not enabled.

16.3.6 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 16-26. 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 “clk_counter” while in Quadrature mode and “clk_hf_counter” for all other modes.*

Table 16-27. 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 16-28. 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 16-29. 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 16-30. PWM_PR PWM outputs

PWM outputs	Description
line_out	PWM output.
line_compl_out	Complementary PWM output.

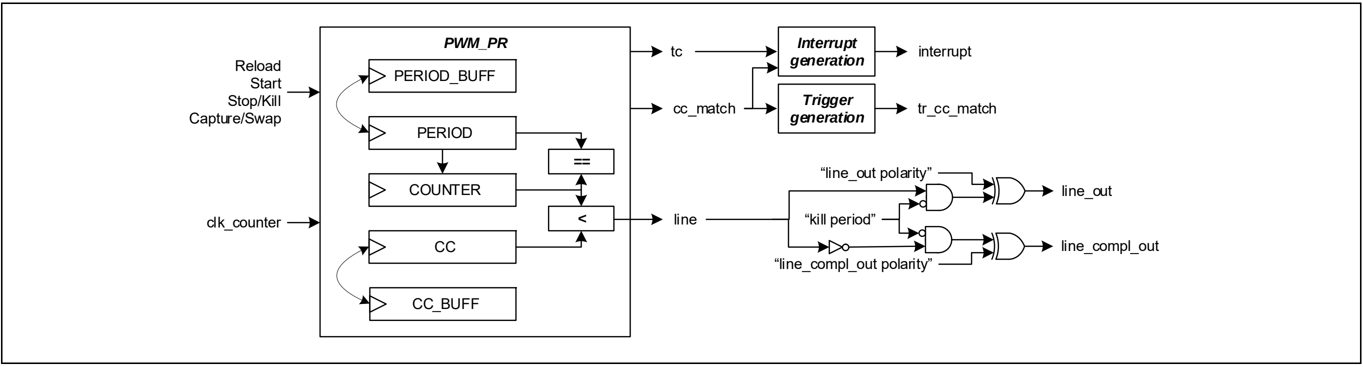


Figure 16-42. PWM_PR functionality

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.
  - 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 line 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-bit values. As a result, for CC greater or equal to 0x8000, pwm_dt_input is always 1. The line 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 16-43 illustrates PWM_PR functionality.

Note:

- The grey shaded areas represent the counter region in which the line 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 line value of ‘1’.

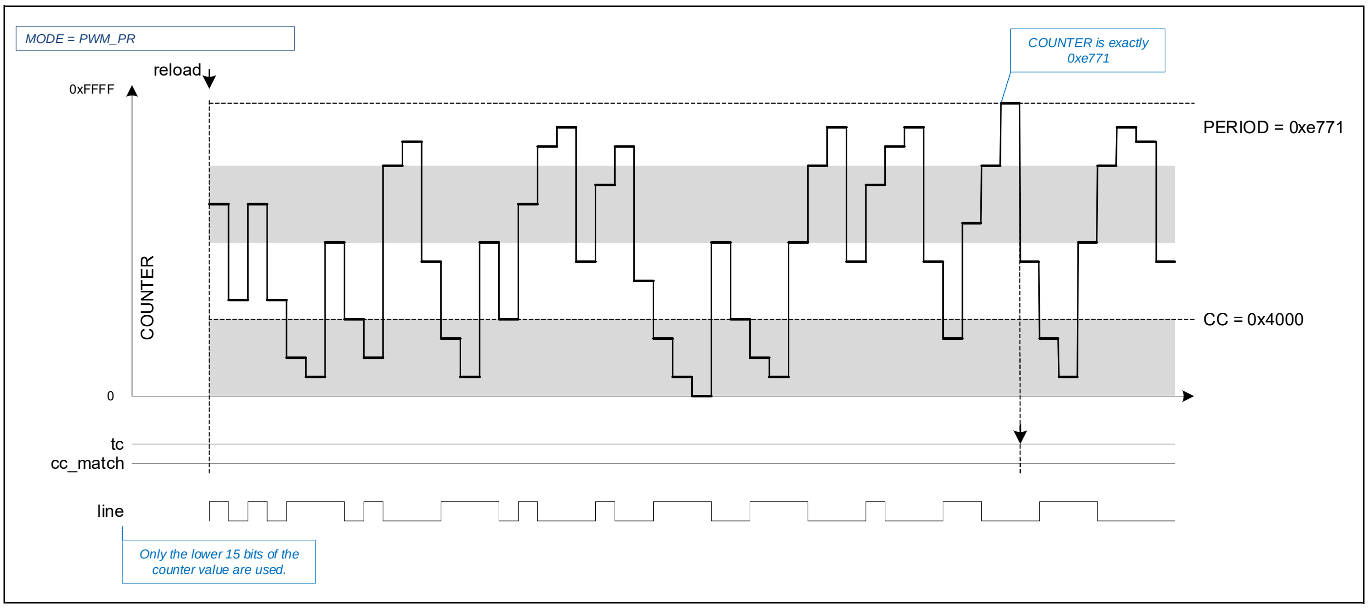

MODE = PWM_PR
reload
0xFFFF
COUNTER
0
COUNTER is exactly 0xe771
PERIOD = 0xe771
CC = 0x4000
tc
cc_match
line
Only the lower 15 bits of the counter value are used.

Figure 16-43. PWM_PR output

16.3.6.1 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 ‘0’ to the COUNTER_ENABLED field of the TCPWM_CTRL 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. line_out and line_compl_out 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 COUNTER_ENABLED field of the TCPWM_CTRL register. A reload trigger must be provided through firmware (TCPWM_CMD register) to start the counter if the hardware reload signal is not enabled.

16.4 TCPWM registers

Table 16-31. List of TCPWM registers

Register	Comment	Features
TCPWM_CTRL	TCPWM Control Register	Enables the counter block
TCPWM_CMD	TCPWM Command Register	Generates software events
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, switching, kill feature, 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 if the counter is running
TCPWM_CNT_COUNTER	Count Register	Contains the 16-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; switches 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; switches 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

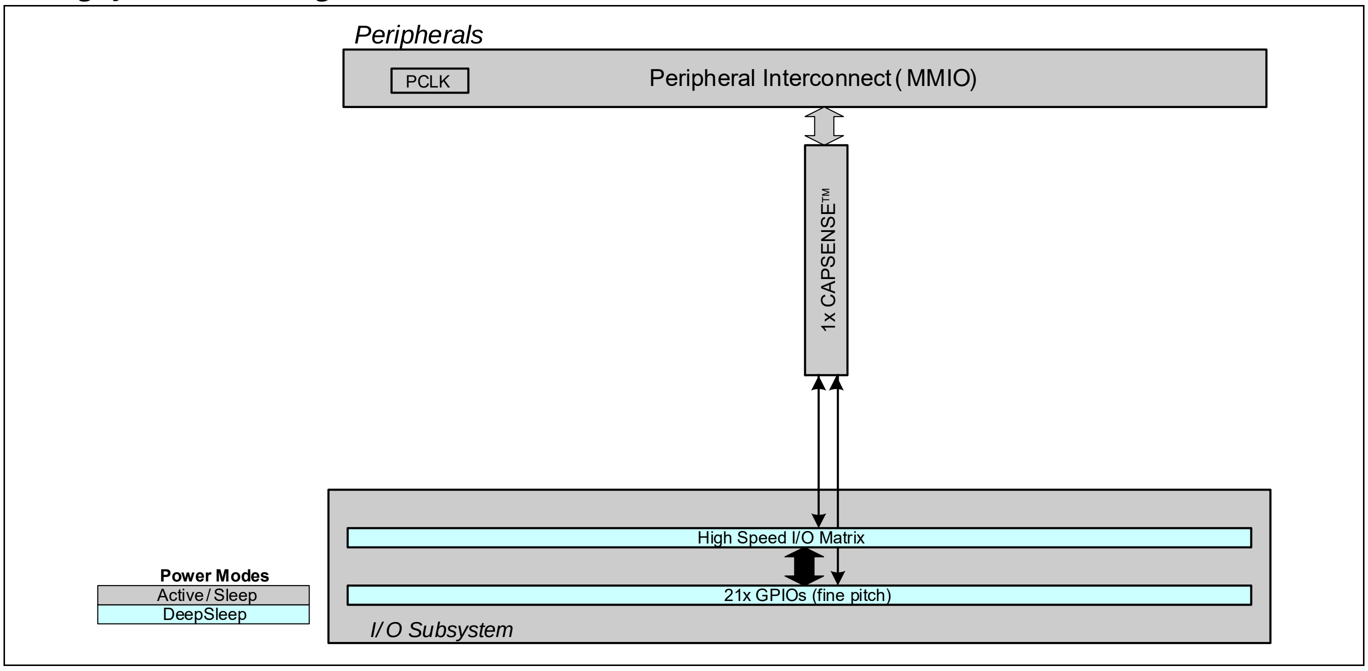
Section E: Analog system

This section encompasses the following chapter:

- **"CAPSENSE™"** on page 181

Top level architecture

Analog system block diagram



17 CAPSENSE™

CAPSENSE™ is supported in PSoC™ 4000T via the MSCLP CAPSENSE™ block. There is one MSCLP block in the PSoC™ 4000T which can be used to scan sense inputs autonomously (without CPU sequencing and intervention) in deep sleep and active modes. CAPSENSE™ function can thus be provided on a pin or group of pins in a system via autonomous scanning or via firmware control.

The PSoC™ 4000T MSCLP block provides the following improvements over previous generation capacitive sensing blocks:

- Improved SNR based on the all new ratio-metric analog architecture and advanced hardware filtering to enable modern sleek user interface solutions with superior liquid tolerance and provides robust and reliable touch HMI solution for harsh environments.
- Higher sensitivity to support smaller sensors, higher proximity detection range, and a much wider range of overlay thicknesses and materials.
- Ultra-low power operation through "Always-On" sensing which provides hardware-based sensor-data-processing for automatic touch detection in device Deep Sleep mode, to allow wake-on-touch operation.
- Autonomous, i.e. CPU independent, channel sequencing and scanning, for low power optimization.
- Improved shield drive method and support for wider range of shield electrode capacitances for superior liquid tolerance.
- Higher sensor capacitance range to support easier layout and wider variety of sensors.
- Improved EMI performance
- A driver is provided for the CAPSENSE™ block for easy usability.
- The CAPSENSE™ block provides multiple sensing methods such as mutual capacitance sensing, self capacitance sensing, and inductive sensing.

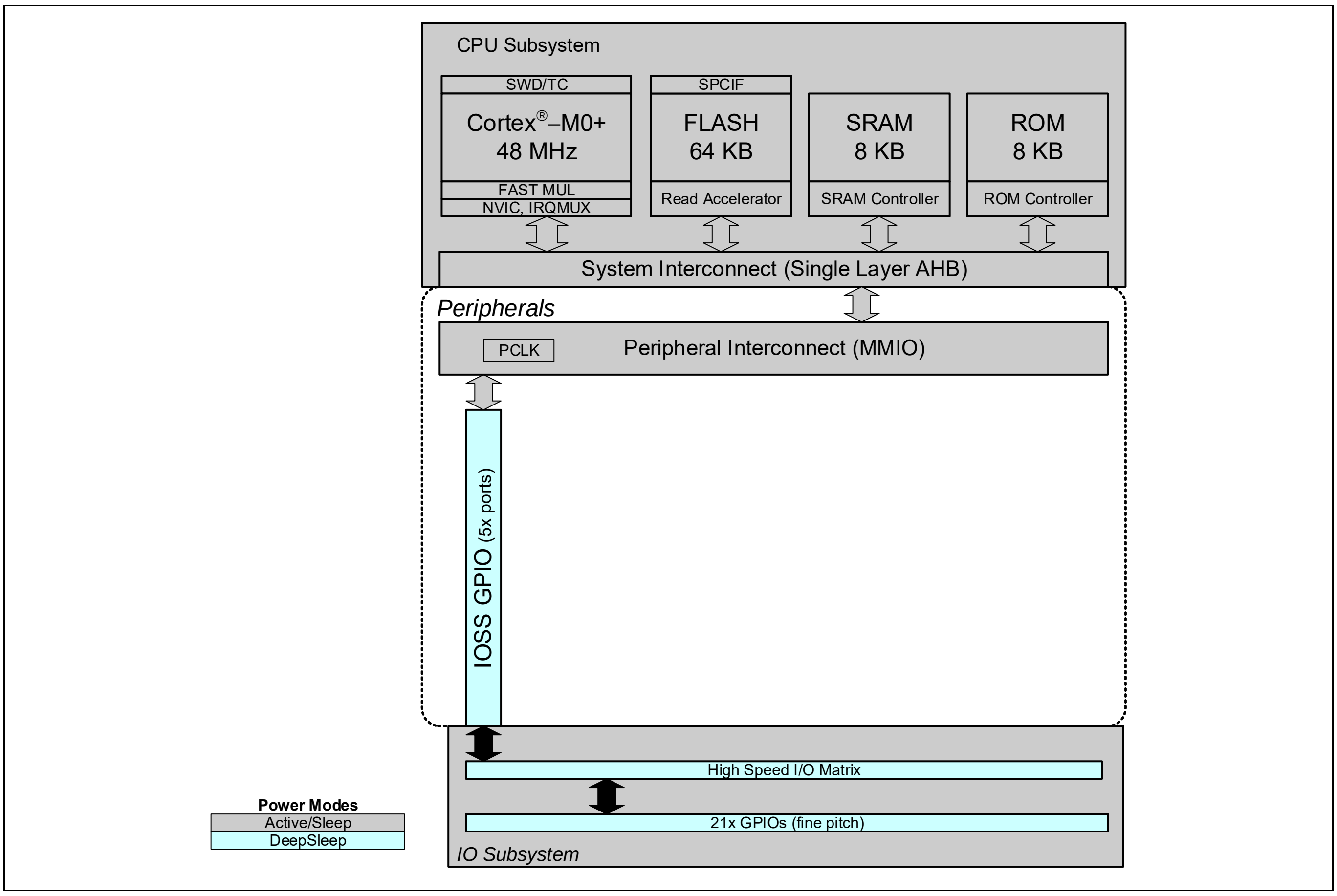
Section F: Program and debug

This section encompasses the following chapters:

- **"Program and debug interface"** on page 183
- **"Nonvolatile memory programming"** on page 192

Top level architecture

Program and debug block diagram



18 Program and debug interface

The PSoC™ 4 Program and Debug interface provides a communication gateway for an external device to perform programming or debugging. The external device can be a Infineon®-supplied programmer and debugger, or a third-party device that supports programming and debugging. The serial wire debug (SWD) interface is used as the communication protocol between the external device and PSoC™ 4.

18.1 Features

- Programming and debugging through the SWD interface
- Four hardware breakpoints and two hardware watchpoints while debugging
- Read and write access to all memory and registers in the system while debugging, including the Cortex®-M0+ register bank when the core is running or halted

18.2 Functional description

Figure 18-1 shows the block diagram of the program and debug interface in PSoC™ 4. The Cortex®-M0+ 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™ 4 “target” using the two pins of the SWD interface - the bidirectional data pin (SWDIO) and the host-driven clock pin (SWDCK). The SWD physical port pins (SWDIO and SWDCK) communicate with the DAP through the high-speed I/O matrix (HSIOM). See the **“I/O system”** on page 44 for details on HSIOM.

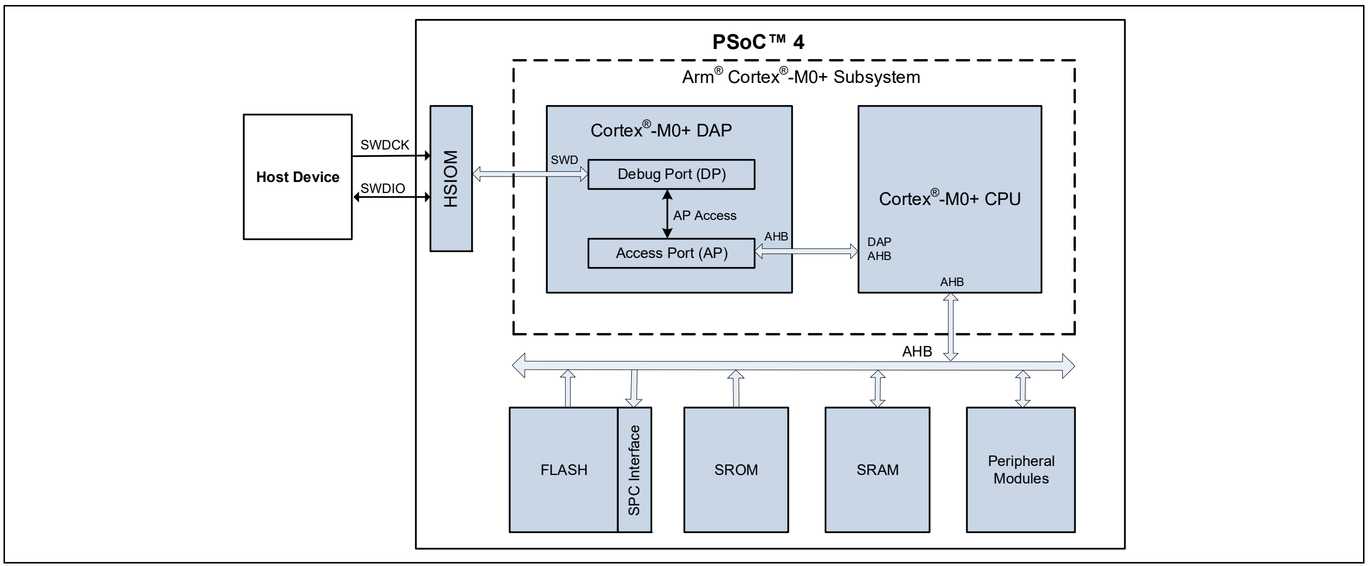


Figure 18-1. Program and debug interface

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 device has two AHB masters – Arm® Cortex®-M0+ (CM0+) CPU core and DAP. The external device can effectively take control of the entire device through the DAP to perform programming and debugging operations.

18.3 Serial wire debug (SWD) interface

PSoC™ 4's Cortex®-M0+ 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™ 4 target.
- **Target Acknowledge Response Phase** – The PSoC™ 4 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 18-2 shows the timing diagrams of read and write SWD packets.

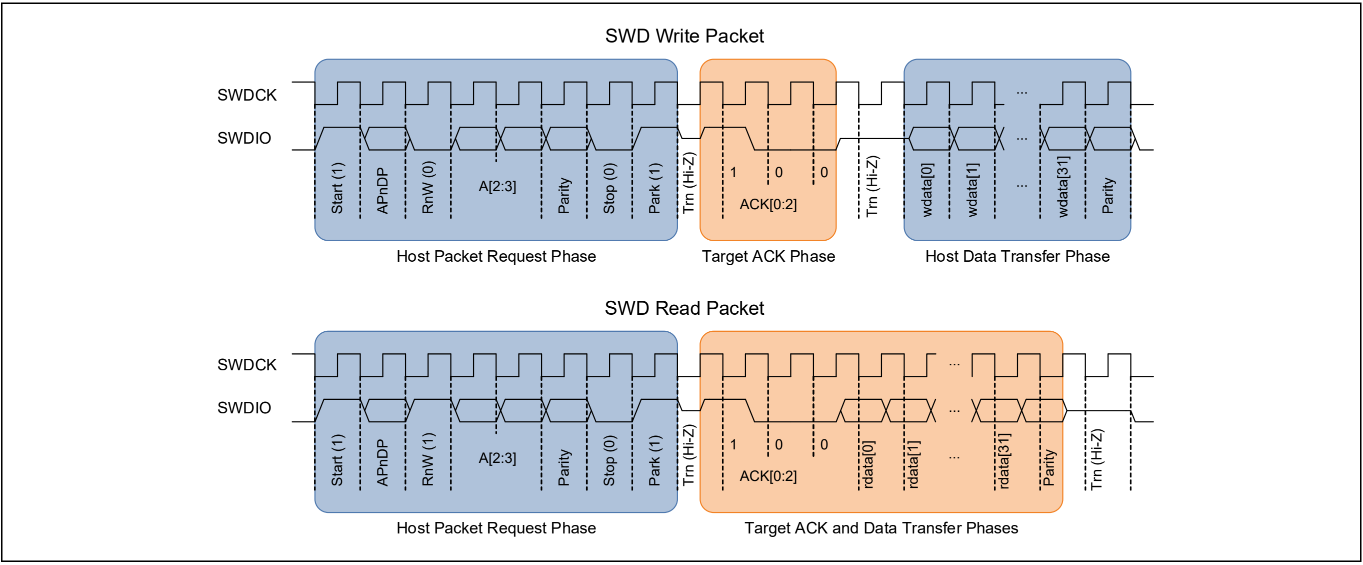


Figure 18-2. SWD write and read packet timing diagrams

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 – 1b1 or a DP access – 1b0.
   c) The "Read not Write" bit (RnW) controls which direction the data transfer is in. 1b1 represents a 'read from' the target, or 1b0 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. See **Table 18-3** and **Table 18-4** for definitions.
   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.
   f) If the parity bit is not correct, the header is ignored by PSoC™ 4; there is no ACK response (ACK = 3b111). The programming operation should be aborted and retried again by following a device reset.
   g) The stop bit is always logic 0.
   h) The park bit is always logic 1.

2. Target Acknowledge Response Phase: SWDIO driven by the target

*Note: The ACK[2:0] bits represent the target to host response, indicating failure or success, among other results. See **Table 18-3** for definitions. 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.
   c) 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. It is recommended to generate three or more dummy clock cycles 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.

18.3.1 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 18-1** and **Figure 18-2** illustrate the timing of SWDIO bit writes and reads.

Table 18-1. SWDIO bit write and read timing

SWD packet phase	SWDIO edge	
	Falling	**Rising**
Host Packet Request	Host Write	Target Read
Host Data Transfer	Host Write	Target Read
Target Ack Response	Host Read	Target Write
Target Data Transfer	Host Read	Target Write

18.3.2 ACK details

The acknowledge (ACK) bit-field 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 18-3** shows the ACK bit-field decoding details.

Table 18-2. SWD transfer ACK response decoding

Response	ACK[2:0]
OK	3b001
WAIT	3b010
FAULT	3b100
NO ACK	3b111

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™ 4. 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.
- For a WAIT response, it means that the PSoC™ 4 is processing the previous transaction. The host can try for a maximum of four continuous WAIT responses to see if 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.

18.3.3 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 18-2**. 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™ 4 should drive the SWDIO line during the Trn period.

18.4 Cortex®-M0+ debug and access port (DAP)

The Cortex®-M0+ program and debug interface includes a Debug Port (DP) and an Access Port (AP), which combine to form the DAP. The debug port implements the state machine for the SWD interface protocol that enables communication with the host device. It also includes registers for the configuration of access port, DAP identification code, and so on. The access port contains registers that enable the external device to access the Cortex®-M0+ DAP-AHB interface. Typically, the DP registers are used for a one time configuration or for error detection purposes, and the AP registers are used to perform the programming and debugging operations. Complete architecture details of the DAP is available in the **Arm® debug interface v5 architecture specification**.

18.4.1 Debug port (DP) registers

Table 18-3 shows the Cortex®-M0+ DP registers used for programming and debugging, along with the corresponding SWD address bit selections. The APnDP bit is always zero for DP register accesses. Two address bits (A[3:2]) are used for selecting among the different DP registers. Note that for the same address bits, different DP registers can be accessed depending on whether it is a read or a write operation. See the **Arm® debug interface v5 architecture specification** for details on all of the DP registers.

Table 18-3. Main debug port (DP) registers

Register	APnDP	Address A[3:2]	RnW	Full name	Register functionality
ABORT	0 (DP)	2b00	0 (W)	AP Abort Register	This register is used to force a DAP abort and to clear the error and sticky flag conditions.
IDCODE	0 (DP)	2b00	1 (R)	Identification Code Register	This register holds the SWD ID of the Cortex®-M0+ CPU, which is 0x0BC11477.
CTRL/STAT	0 (DP)	2b01	X (R/W)	Control and Status Register	This register allows control of the DP and contains status information about the DP.
SELECT	0 (DP)	2b10	0 (W)	AP Select Register	This register is used to select the current AP. In PSoC™ 4, there is only one AP, which interfaces with the DAP AHB.
RDBUFF	0 (DP)	2b11	1 (R)	Read Buffer Register	This register holds the result of the last AP read operation.

18.4.2 Access port (AP) registers

Table 18-4 lists the main Cortex®-M0+ AP registers that are used for programming and debugging, along with the corresponding SWD address bit selections. The APnDP bit is always one for AP register accesses. Two address bits (A[3:2]) are used for selecting the different AP registers.

Table 18-4. Main access port (AP) registers

Register	APnDP	Address A[3:2]	RnW	Full name	Register functionality
CSW	1 (AP)	2b00	X (R/W)	Control and Status Word Register (CSW)	This register configures and controls accesses through the memory access port to a connected memory system (which is the PSoC™ 4 Memory map)
TAR	1 (AP)	2b01	X (R/W)	Transfer Address Register	This register is used to specify the 32-bit memory address to be read from or written to
DRW	1 (AP)	2b11	X (R/W)	Data Read and Write Register	This register holds the 32-bit data read from or to be written to the address specified in the TAR register

18.5 Programming the PSoC™ 4 device

PSoC™ 4 is programmed using the following sequence. Refer to see the **CY8C4xxx, CYBLxxxx programming specifications** for complete details on the programming algorithm, timing specifications, and hardware configuration required for programming.

1. Acquire the SWD port in PSoC™ 4.
2. Enter the programming mode.
3. Execute the device programming routines such as Silicon ID Check, Flash Programming, Flash Verification, and Checksum Verification.

18.5.1 SWD port acquisition

18.5.1.1 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 pseudo code for the sequence is given here.

Code 1. SWD Port Acquire Pseudo Code

```
ToggleXRES(); // Toggle XRES pin to reset device

//Execute Arm’s connection sequence to acquire SWD-port
do
{
        SWD_LineReset(); //perform a line reset (50+ SWDCK clocks with SWDIO high)
        ack = Read_DAP ( IDCODE, out ID); //Read the IDCODE DP register

}while ((ack != OK) && time_elapsed < ms); //retry connection until OK ACK or timeout

if (time_elapsed >= ms) return FAIL; //check for acquire time out

if (ID != CM0P_ID) return FAIL; //confirm SWD ID of Cortex-M0+ CPU. (0x0BC11477)
```

In this pseudo code, SWD_LineReset() is the standard Arm® command to reset the debug access port. It consists of more than 49 SWDCK clock cycles with SWDIO 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 (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.

18.5.2 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 **CY8C4xxx, CYBLxxxx programming specifications**. The minimum required clock frequency for the Port Acquire step and this step to succeed is 1.5 MHz.

18.5.3 SWD programming routines 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 programming"** on page 192. The exact sequence of calling the programming routines is given in the **CY8C4xxx, CYBLxxxx programming specifications**.

18.6 PSoC™ 4 SWD debug interface

Cortex®-M0+ DAP debugging features are classified into two types: invasive debugging and noninvasive debugging. Invasive debugging includes program halting and stepping, breakpoints, and data watchpoints. Noninvasive debugging includes instruction address profiling and device memory access, which includes the flash memory, SRAM, and other peripheral registers.

The DAP has three major debug subsystems:

- Debug Control and Configuration registers
- Breakpoint Unit (BPU) – provides breakpoint support
- Debug Watchpoint (DWT) – provides watchpoint support. Trace is not supported in Cortex®-M0+ Debug.

See the **Armv6-M architecture reference manual** for complete details on the debug architecture.

18.6.1 Debug control and configuration registers

The debug control and configuration registers are used to execute firmware debugging. The registers and their key functions are as follows. See the **Armv6-M architecture reference manual** for complete bit level definitions of these registers.

- Debug Halting Control and Status Register (CM0P_DHCSR) – This register contains the control bits to enable debug, halt the CPU, and perform a single-step operation. It also includes status bits for the debug state of the processor.
- Debug Fault Status Register (CM0P_DFSR) – This register describes the reason a debug event has occurred and includes debug events, which are caused by a CPU halt, breakpoint event, or watchpoint event.
- Debug Core Register Selector Register (CM0P_DCRSR) – This register is used to select the general-purpose register in the Cortex®-M0+ CPU to which a read or write operation must be performed by the external debugger.
- Debug Core Register Data Register (CM0P_DCRDR) – This register is used to store the data to write to or read from the register selected in the CM0P_DCRSR register.
- Debug Exception and Monitor Control Register (CM0P_DEMCR) – This register contains the enable bits for global debug watchpoint (DWT) block enable, reset vector catch, and hard fault exception catch.

18.6.2 Breakpoint unit (BPU)

The BPU provides breakpoint functionality on instruction fetches. The Cortex®-M0+ DAP in PSoC™ 4 supports up to four hardware breakpoints. Along with the hardware breakpoints, any number of software breakpoints can be created by using the BKPT instruction in the Cortex®-M0+. The BPU has two types of registers.

- The breakpoint control register (CM0P_BP_CTRL) is used to enable the BPU and store the number of hardware breakpoints supported by the debug system (four for CM0 DAP in the PSoC™ 4).
- Each hardware breakpoint has a Breakpoint Compare Register (CM0P_BP_COMPx). It contains the enable bit for the breakpoint, the compare address value, and the match condition that will trigger a breakpoint debug event. The typical use case is that when an instruction fetch address matches the compare address of a breakpoint, a breakpoint event is generated and the processor is halted.

18.6.3 Data watchpoint (DWT)

The DWT provides watchpoint support on a data address access or a program counter (PC) instruction address. The DWT supports two watchpoints. It also provides external program counter sampling using a PC sample register, which can be used for noninvasive coarse profiling of the program counter. The most important registers in the DWT are as follows:

- The watchpoint compare (CM0P_DWT_COMPx) registers store the compare values that are used by the watchpoint comparator for the generation of watchpoint events. Each watchpoint has an associated DWT_COMPx register.
- The watchpoint mask (CM0P_DWT_MASKx) registers store the ignore masks applied to the address range matching in the associated watchpoints.
- The watchpoint function (CM0P_DWT_FUNCTIONx) registers store the conditions that trigger the watchpoint events. They may be program counter watchpoint event or data address read/write access watchpoint events. A status bit is also set when the associated watchpoint event has occurred.
- The watchpoint comparator PC sample register (CM0P_DWT_PCSR) stores the current value of the program counter. This register is used for coarse, non-invasive profiling of the program counter register.

18.6.4 Debugging the PSoC™ 4 device

The host debugs the target PSoC™ 4 by accessing the debug control and configuration registers, registers in the BPU, and registers in the DWT. All registers are accessed through the SWD interface; the SWD debug port (SW-DP) in the Cortex®-M0+ DAP converts the SWD packets to appropriate register access through the DAP-AHB interface.

The first step in debugging the target PSoC™ 4 is to acquire the SWD port. The acquire sequence consists of an SWD line reset sequence and read of the DAP SWDID through the SWD interface. The SWD port is acquired when the correct CM0 DAP SWDID is read from the target device. For the debug transactions to occur on the SWD interface, the corresponding pins should not be used for any other purpose. See the **"I/O system"** on page 44 to understand how to configure the SWD port pins, allowing them to be used only for SWD interface or for other functions such as LCD and GPIO. If debugging is required, the SWD port pins should not be used for other purposes. If only programming support is needed, the SWD pins can be used for other purposes.

When the SWD port is acquired, the external debugger sets the C_DEBUGEN bit in the DHCSR register to enable debugging. Then, the different debugging operations such as stepping, halting, breakpoint configuration, and watchpoint configuration are carried out by writing to the appropriate registers in the debug system.

Debugging the target device is also affected by the overall device protection setting, which is explained in the **"Device security"** on page 41. Only the OPEN protected mode supports device debugging. The external debugger and the target device connection is not lost for a device transition from Active mode to either Sleep or Deep-Sleep modes. When the device enters the Active mode from either Deep-Sleep or Sleep modes, the debugger can resume its actions without initiating a connect sequence again.

18.7 Registers

Table 18-5. List of registers

Register name	Description
CM0P_DHCSR	Debug Halting Control and Status Register
CM0P_DFSR	Debug Fault Status Register
CM0P_DCRSR	Debug Core Register Selector Register
CM0P_DCRDR	Debug Core Register Data Register
CM0P_DEMCR	Debug Exception and Monitor Control Register
CM0P_BP_CTRL	Breakpoint control register
CM0P_BP_COMPx	Breakpoint Compare Register
CM0P_DWT_COMPx	Watchpoint Compare Register
CM0P_DWT_MASKx	Watchpoint Mask Register
CM0P_DWT_FUNCTIONx	Watchpoint Function Register
CM0P_DWT_PCSR	Watchpoint Comparator PC Sample Register

19 Nonvolatile memory programming

Nonvolatile memory programming refers to the programming of flash memory in the PSoC™ 4 device. This chapter explains the different functions that are part of device programming, such as erase, write, program, and checksum calculation. Infineon®-supplied programmers and other third-party programmers can use these functions to program the PSoC™ 4 device with the data in an application hex file. They can also be used to perform bootload operations where the CPU will update a portion of the flash memory.

19.1 Features

- Supports programming through the debug and access port (DAP) and Cortex®-M0+ CPU
- Supports both blocking and non-blocking flash program and erase operations from the Cortex®-M0+ CPU

19.2 Functional description

Flash programming operations are implemented as system calls. System calls are executed out of SROM in the privileged mode of operation. The user has no access to read or modify the SROM code. The DAP or the CM0+ CPU requests the system call by writing the function opcode and parameters to the System Performance Controller Interface (SPCIF) input registers, and then requesting the SROM to execute the function. Based on the function opcode, the System Performance Controller (SPC) executes the corresponding system call from SROM and updates the SPCIF status register. The DAP or the CPU should read this status register for the pass/fail result of the function execution. As part of function execution, the code in SROM interacts with the SPCIF to do the actual flash programming operations.

PSoC™ 4 flash is programmed using a Program Erase Program (PEP) sequence. The flash cells are all programmed to a known state, erased, and then the selected bits are programmed. This sequence increases the life of the flash by balancing the stored charge. When writing to flash the data is first copied to a page latch buffer. The flash write functions are then used to transfer this data to flash.

External programmers program the flash memory in PSoC™ 4 using the SWD protocol by sending the commands to the Debug and Access Port (DAP). The programming sequence for the PSoC™ 4 device with an external programmer is given by **CY8C4xxx, CYBLxxxx Programming Specifications**. Flash memory can also be programmed by the CM0+ CPU by accessing the relevant registers through the AHB 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 requirements, 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 SPCIF.

*Note: It can take as much as 20 milliseconds to write to flash. During this time, the device should not be reset, or unexpected changes may be made to portions of the flash. Reset sources (see the **"Reset system"** on page 76) include XRES pin, software reset, and watchdog; make sure that these are not inadvertently activated. In addition, the low-voltage detect circuits should be configured to generate an interrupt instead of a reset.*

Note: PSoC™ 4 implements a User Supervisory Flash (SFlash), which can be used to store application-specific information. These rows are not part of the hex file; their programming is optional.

19.3 System call implementation

A system call consists of the following items:

- Opcode: A unique 8-bit opcode
- Parameters: Two 8-bit parameters are mandatory for all system calls. These parameters are referred to as key1 and key2, and are defined as follows:
  key1 = 0xB6
  key2 = 0xD3 + Opcode
  The two keys are passed to ensure that the user system call is not initiated by mistake. If the key1 and key2 parameters are not correct, the SROM does not execute the function, and returns an error code. Apart from these two parameters, additional parameters may be required depending on the specific function being called.
- Return Values: Some system calls also return a value on completion of their execution, such as the silicon ID or a checksum.
- Completion Status: Each system call returns a 32-bit status that the CPU or DAP can read to verify success or determine the reason for failure.

19.4 Blocking and non-blocking system calls

System call functions can be categorized as blocking or non-blocking based on the nature of their execution. Blocking system calls are those where the CPU cannot execute any other task in parallel other than the execution of the system call. When a blocking system call is called from a process, the CPU jumps to the code corresponding in SROM. When the execution is complete, the original thread execution resumes. Non-blocking system calls allow the CPU to execute some other code in parallel and communicate the completion of interim system call tasks to the CPU through an interrupt.

Non-blocking system calls are only used when the CPU initiates the system call. The DAP will only use system calls during the programming mode and the CPU is halted during this process.

The three non-blocking system calls are Non-Blocking Write Row, Non-Blocking Program Row, and Resume Non-Blocking, respectively. All other system calls are blocking.

Because the CPU cannot execute code from flash while doing an erase or program operation on the flash, the non-blocking system calls can only be called from a code executing out of SRAM. If the non-blocking functions are called from flash memory, the result is undefined and may return a bus error and trigger a hard fault when the flash fetch operation is being done.

The System Performance Controller (SPC) is the block that generates the properly sequenced high-voltage pulses required for erase and program operations of the flash memory. When a non-blocking function is called from SRAM, the SPC timer triggers its interrupt when each of the sub-operations in a write or program operation is complete. Call the Resume Non-Blocking function from the SPC interrupt service routine (ISR) to ensure that the subsequent steps in the system call are completed. Because the CPU can execute code only from the SRAM when a non-blocking write or program operation is being done, the SPC ISR should also be located in the SRAM. The SPC interrupt is triggered once in the case of a non-blocking program function or thrice in a non-blocking write operation. The Resume Non-Blocking function call done in the SPC ISR is called once in a non-blocking program operation and thrice in a non-blocking write operation.

The pseudo code for using a non-blocking write system call and executing user code out of SRAM is given later in this chapter.

19.4.1 Performing a system call

The steps to initiate a system call are as follows:

1. Set up the function parameters: The two possible methods for preparing the function parameters (key1, key2, additional parameters) are:
   a) Write the function parameters to the CPUSS_SYSARG register: This method is used for functions that retrieve their parameters from the CPUSS_SYSARG register. The 32-bit CPUSS_SYSARG register must be written with the parameters in the sequence specified in the respective system call table.
   b) Write the function parameters to SRAM: This method is used for functions that retrieve their parameters from SRAM. The parameters should first be written in the specified sequence to consecutive SRAM locations. Then, the starting address of the SRAM, which is the address of the first parameter, should be written to the CPUSS_SYSARG register. This starting address should always be a word-aligned (32-bit) address. The system call uses this address to fetch the parameters.
2. Specify the system call using its opcode and initiating the system call: The 8-bit opcode should be written to the SYSCALL_COMMAND bits ([15:0]) in the CPUSS_SYSREQ register. The opcode is placed in the lower eight bits [7:0] and 0x00 be written to the upper eight bits [15:8]. To initiate the system call, set the SYSCALL_REQ bit (31) in the CPUSS_SYSREQ register. Setting this bit triggers a non-maskable interrupt that jumps the CPU to the SROM code referenced by the opcode parameter.
3. Wait for the system call to finish executing: When the system call begins execution, it sets the PRIVILEGED bit in the CPUSS_SYSREQ register. This bit can be set only by the system call, not by the CPU or DAP. The DAP should poll the PRIVILEGED and SYSCALL_REQ bits in the CPUSS_SYSREQ register continuously to check whether the system call is completed. Both these bits are cleared on completion of the system call. The maximum execution time is one second. If these two bits are not cleared after one second, the operation should be considered a failure and aborted without executing the following steps. Note that unlike the DAP, the CPU application code cannot poll these bits during system call execution. This is because the CPU executes code out of the SROM during the system call. The application code can check only the final function pass/fail status after the execution returns from SROM.
4. Check the completion status: After the PRIVILEGED and SYSCALL_REQ bits are cleared to indicate completion of the system call, the CPUSS_SYSARG register should be read to check for the status of the system call. If the 32-bit value read from the CPUSS_SYSARG register is 0xAXXXXXXX (where ‘X’ denotes don’t care hex values), the system call was successfully executed. For a failed system call, the status code is 0xF00000YY where YY indicates the reason for failure. See **Table 19-1** for the complete list of status codes and their description.
5. Retrieve the return values: For system calls that return values such as silicon ID and checksum, the CPU or DAP should read the CPUSS_SYSREQ and CPUSS_SYSARG registers to fetch the values returned.

19.5 System calls

Table 19-1 lists all the system calls supported in PSoC™ 4 along with the function description and availability in device protection modes. See the **"Device security"** on page 41 for more information on the device protection settings. Note that some system calls cannot be called by the CPU as given in the table. Detailed information on each of the system calls follows the table.

Table 19-1. List of system calls

System call	Description	DAP access			CPU access
		Open	Protected	Kill	
Silicon ID	Returns the device Silicon ID, Family ID, and Revision ID	4	4	–	4
Load Flash Bytes	Loads data to the page latch buffer to be programmed later into the flash row, in 1 byte granularity, for a row size of 128 bytes	4	–	–	4
Write Row	Erases and then programs a row of flash with data in the page latch buffer	4	–	–	4
Program Row	Programs a row of flash with data in the page latch buffer	4	–	–	4
Erase All	Erases all user code in the flash array; the flash row-level protection data in the supervisory flash area	4	–	–	
Checksum	Calculates the checksum over the entire flash memory (user and supervisory area) or checksums a single row of flash	4	4	–	4
Write Protection	This programs both flash row-level protection settings and chip-level protection settings into the supervisory flash (row 0)	4	4	–	
Non-Blocking Write Row	Erases and then programs a row of flash with data in the page latch buffer. During program/erase pulses, the user may execute code from SRAM. This function is meant only for CPU access	–	–	–	4
Non-Blocking Program Row	Programs a row of flash with data in the page latch buffer. During program/erase pulses, the user may execute code from SRAM. This function is meant only for CPU access	–	–	–	4
Resume Non-Blocking	Resumes a non-blocking write row or non-blocking program row. This function is meant only for CPU access	–	–	–	4

19.5.1 Silicon ID

This function returns a 12-bit family ID, 16-bit silicon ID, and an 8-bit revision ID, and the current device protection mode. These values are returned to the CPUSS_SYSARG and CPUSS_SYSREQ registers. Parameters are passed through the CPUSS_SYSARG and CPUSS_SYSREQ registers.

Parameters

Address	Value to be written	Description
	CPUSS_SYSARG register	
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xD3	Key2
Bits [31:16]	0x0000	Not used
CPUSS_SYSREQ register		
Bits [15:0]	0x0000	Silicon ID opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	Return value	Description
CPUSS_SYSARG register		
Bits [7:0]	Silicon ID Lo	
Bits [15:8]	Silicon ID Hi	0x3600 to 0x36FF
Bits [19:16]	Minor Revision Id	See the **CY8C4xxx, CYBLxxxx Programming Specifications** for these values.
Bits [23:20]	Major Revision Id	
Bits [27:24]	0xXX	Not used (don't care)
Bits [31:28]	0xA	Success status code
CPUSS_SYSREQ register		
Bits [11:0]	Family ID	Family ID is 0xC6 for PSoC™ 4000T MCU
Bits [15:12]	Chip Protection	See the **"Device security"** on page 41.
Bits [31:16]	0xXXXX	Not used

19.5.2 Configure clock

This function initializes the clock necessary for flash programming and erasing operations. This API is used to ensure that the charge pump clock (clk_pump) and the HFCLK (clk_hf) are set to IMO at 48 MHz prior to calling the flash write and flash erase APIs. The flash write and erase APIs will exit without acting on the flash and return the "Invalid Pump Clock Frequency" status if the IMO is the source of the charge pump clock and is not 48 MHz.

19.5.3 Load flash bytes

This function loads the page latch buffer with data to be programmed into a row of flash. The load size can range from 1-byte to the maximum number of bytes in a flash row, which is 128 bytes. Data is loaded into the page latch buffer starting at the location specified by the "Byte Addr" input parameter. Data loaded into the page latch buffer remains until a program operation is performed, which clears the page latch contents. The parameters for this function, including the data to be loaded into the page latch, are written to the SRAM; the starting address of the SRAM data is written to the CPUSS_SYSARG register. Note that the starting parameter address should be a word-aligned address.

Parameters

Address	**Value to be written**	**Description**
SRAM Address - 32'hYY (32-bit wide, word-aligned SRAM address)		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xD7	Key2
Bits [23:16]	Byte Addr	Start address of page latch buffer to write data 0x00 – Byte 0 of latch buffer 0x7F – Byte 127 of latch buffer
Bits [31:24]	Flash Macro Select	0x00 – Flash Macro 0
SRAM Address- 32'hYY + 0x04		
Bits [7:0]	Load Size	Number of bytes to be written to the page latch buffer. 0x00 – 1 byte 0x7F – 128 bytes
Bits [15:8]	0xXX	Don't care parameter
Bits [23:16]	0xXX	Don't care parameter
Bits [31:24]	0xXX	Don't care parameter
SRAM Address- From (32'hYY + 0x08) to (32'hYY + 0x08 + Load Size)		
Byte 0	Data Byte [0]	First data byte to be loaded
.	.	.
.	.	.
Byte (Load size –1)	Data Byte [Load size –1]	Last data byte to be loaded
CPUSS_SYSARG register		
Bits [31:0]	32'hYY	32-bit word-aligned address of the SRAM that stores the first function parameter (key1)
CPUSS_SYSREQ register		
Bits [15:0]	0x0004	Load Flash Bytes opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	Return Value	Description
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:0]	0xXXXXXXX	Not used (don’t care)

19.5.4 Write row

This function erases and then programs the addressed row of flash with the data in the page latch buffer. If all data in the page latch buffer is 0, then the program is skipped. The parameters for this function are stored in SRAM. The start address of the stored parameters is written to the CPUSS_SYSARG register. This function clears the page latch buffer contents after the row is programmed.

Usage Requirements: Call the Configure Clock API before calling this function. The Configure Clock API ensures that the charge pump clock (clk_pump) and the HFCLK (clk_hf) are set to IMO at 48 MHz. Call the Load Flash Bytes function before calling this function. This function can do a write operation only if the corresponding flash row is not write protected. Refer to the CLK_IMO_CONFIG register in the PSoC™ 4000T MCU registers TRM for more information.

Parameters

Address	Value to be written	Description
SRAM Address: 32’hYY (32-bit wide, word-aligned SRAM address)		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xD8	Key2
Bits [31:16]	Row ID	Row number to write 0x0000 – Row 0
CPUSS_SYSARG register		
Bits [31:0]	32’hYY	32-bit word-aligned address of the SRAM that stores the first function parameter (key1)
CPUSS_SYSREQ register		
Bits [15:0]	0x0005	Write Row opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	Return value	Description
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:0]	0xXXXXXXX	Not used (don’t care)

19.5.5 Program row

This function programs the addressed row of the flash with data in the page latch buffer. If all data in the page latch buffer is 0, then the program is skipped. The row must be in an erased state before calling this function. It clears the page latch buffer contents after the row is programmed.

Usage Requirements: Call the Configure Clock API before calling this function. The Configure Clock API ensures that the charge pump clock (clk_pump) and the HFCLK (clk_hf) are set to IMO at 48 MHz. Call the Load Flash Bytes function before calling this function. The row must be in an erased state before calling this function. This function can do a program operation only if the corresponding flash row is not write-protected.

Parameters

Address	Value to be written	Description
SRAM Address: 32'hYY (32-bit wide, word-aligned SRAM address)		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xD9	Key2
Bits [31:16]	Row ID	Row number to program 0x0000 – Row 0
CPUSS_SYSARG register		
Bits [31:0]	32'hYY	32-bit word-aligned address of the SRAM that stores the first function parameter (key1)
CPUSS_SYSREQ register		
Bits [15:0]	0x0006	Program Row opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	Return Value	Description
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:0]	0xXXXXXXX	Not used (don't care)

19.5.6 Erase all

This function erases all the user code in the flash main arrays and the row-level protection data in supervisory flash row 0 of each flash macro.

Usage Requirements: Call the Configure Clock API before calling this function. The Configure Clock API ensures that the charge pump clock (clk_pump) and the HFCLK (clk_hf) are set to IMO at 48 MHz. This API can be called only from the DAP in the programming mode and only if the chip protection mode is OPEN. If the chip protection mode is PROTECTED, then the Write Protection API must be used by the DAP to change the protection settings to OPEN. Changing the protection setting from PROTECTED to OPEN automatically does an erase all operation.

Parameters

Address	Value to be written	Description
SRAM Address: 32’hYY (32-bit wide, word-aligned SRAM address)		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xDD	Key2
Bits [31:16]	0xXXXX	Don’t care
CPUSS_SYSARG register		
Bits [31:0]	32’hYY	32-bit word-aligned address of the SRAM that stores the first function parameter (key1)
CPUSS_SYSREQ register		
Bits [15:0]	0x000A	Erase All opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	Return Value	Description
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:0]	0xXXXXXXX	Not used (don’t care)

19.5.7 Checksum

This function reads either the whole flash memory or a row of flash and returns the 24-bit sum of each byte read in that flash region. When performing a checksum on the whole flash, the user code and supervisory flash regions are included. When performing a checksum only on one row of flash, the flash row number is passed as a parameter. Bytes 2 and 3 of the parameters select whether the checksum is performed on the whole flash memory or a row of user code flash.

Parameters

Address	Value to be written	Description
CPUSS_SYSARG register		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xDE	Key2
Bits [31:16]	Row ID	Selects the flash row number on which the checksum operation is done Row number – 16 bit flash row number or 0x8000 – Checksum is performed on entire flash memory
CPUSS_SYSREQ register		
Bits [15:0]	0x000B	Checksum opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	Return value	Description
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:24]	0xX	Not used (don’t care)
Bits [23:0]	Checksum	24-bit checksum value of the selected flash region

19.5.8 Write protection

This function programs both the flash row-level protection settings and the device protection settings in the supervisory flash row. The flash row-level protection settings are programmed separately for each flash macro in the device. Each row has a single protection bit. The total number of protection bytes is the number of flash rows divided by eight. The chip-level protection settings (1-byte) are stored in flash macro zero in the last byte location in row zero of the supervisory flash. The size of the supervisory flash row is the same as the user code flash row size.

Usage Requirements: Call the Configure Clock API before calling this function. The Configure Clock API ensures that the charge pump clock (clk_pump) and the HFCLK (clk_hf) are set to IMO at 48 MHz. The Load Flash Bytes function is used to load the flash protection bytes of a flash macro into the page latch buffer corresponding to the macro. The starting address parameter for the load function should be zero. The flash macro number should be one that needs to be programmed; the number of bytes to load is the number of flash protection bytes in that macro.

Then, the Write Protection function is called, which programs the flash protection bytes from the page latch to be the corresponding flash macro's supervisory row. In flash macro zero, which also stores the device protection settings, the device level protection setting is passed as a parameter in the CPUSS_SYSARG register.

Parameters

Address	Value to be written	Description
CPUSS_SYSARG register		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xE0	Key2
Bits [23:16]	Device Protection Byte	Parameter applicable only for Flash Macro 0 0x01 – OPEN mode 0x02 – PROTECTED mode 0x04 – KILL mode
Bits [31:24]	Flash Macro Select	0x00 – Flash Macro 0
CPUSS_SYSREQ register		
Bits [15:0]	0x000D	Write Protection opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	Return value	Description
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:24]	0xX	Not used (don't care)
Bits [23:0]	0x000000	

19.5.9 Non-blocking write row

This function is used when a flash row needs to be written by the CM0+ CPU in a non-blocking manner, so that the CPU can execute code from SRAM while the write operation is being done. The explanation of non-blocking system calls is explained in **"Blocking and non-blocking system calls"** on page 193.

The non-blocking write row system call has three phases: Pre-program, Erase, Program. Pre-program is the step in which all of the bits in the flash row are written a '1' in preparation for an erase operation. The erase operation clears all of the bits in the row, and the program operation writes the new data to the row.

While each phase is being executed, the CPU can execute code from SRAM. When the non-blocking write row system call is initiated, the user cannot call any system call function other than the Resume Non-Blocking function, which is required for completion of the non-blocking write operation. After the completion of each phase, the SPC triggers its interrupt. In this interrupt, call the Resume Non-Blocking system call.

Note: The device firmware must not attempt to put the device to sleep during a non-blocking write row. This action will reset the page latch buffer and the flash will be written with all zeroes.

Usage Requirements: Call the Configure Clock API before calling this function. The Configure Clock API ensures that the charge pump clock (clk_pump) and the HFCLK (clk_hf) are set to IMO at 48 MHz. Call the Load Flash Bytes function before calling this function to load the data bytes that will be used for programming the row. In addition, the non-blocking write row function can be called only from the SRAM. This is because the CM0+ CPU cannot execute code from flash while doing the flash erase program operations. If this function is called from the flash memory, the result is undefined, and may return a bus error and trigger a hard fault when the flash fetch operation is being done.

Parameters

Address	**Value to be written**	**Description**
SRAM Address 32'hYY (32-bit wide, word-aligned SRAM address)		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xDA	Key2
Bits [31:16]	Row ID	Row number to write 0x0000 – Row 0
CPUSS_SYSARG register		
Bits [31:0]	32'hYY	32-bit word-aligned address of the SRAM that stores the first function parameter (key1)
CPUSS_SYSREQ register		
Bits [15:0]	0x0007	Non-Blocking Write Row opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	**Return value**	**Description**
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:0]	0xXXXXXXX	Not used (don't care)

19.5.10 Non-blocking program row

This function is used when a flash row needs to be programmed by the CM0+ CPU in a non-blocking manner, so that the CPU can execute code from the SRAM when the program operation is being done. The explanation of non-blocking system calls is explained in **"Blocking and non-blocking system calls"** on page 193. While the program operation is being done, the CPU can execute code from the SRAM. When the non-blocking program row system call is called, the user cannot call any other system call function other than the Resume Non-Blocking function, which is required for the completion of the non-blocking write operation.

Unlike the Non-Blocking Write Row system call, the Program system call only has a single phase. Therefore, the Resume Non-Blocking function only needs to be called once from the SPC interrupt when using the Non-Blocking Program Row system call.

Usage Requirements: Call the Configure Clock API before calling this function. The Configure Clock API ensures that the charge pump clock (clk_pump) and the HFCLK (clk_hf) are set to IMO at 48 MHz. Call the Load Flash Bytes function before calling this function to load the data bytes that will be used for programming the row. In addition, the non-blocking program row function can be called only from SRAM. This is because the CM0+ CPU cannot execute code from flash while doing flash program operations. If this function is called from flash memory, the result is undefined, and may return a bus error and trigger a hard fault when the flash fetch operation is being done.

Parameters

Address	**Value to be written**	**Description**
SRAM Address 32'hYY (32-bit wide, word-aligned SRAM address)		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xDB	Key2
Bits [31:16]	Row ID	Row number to write 0x0000 – Row 0
CPUSS_SYSARG register		
Bits [31:0]	32'hYY	32-bit word-aligned address of the SRAM that stores the first function parameter (key1)
CPUSS_SYSREQ register		
Bits [15:0]	0x0008	Non-Blocking Program Row opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	**Return value**	**Description**
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:0]	0xXXXXXXX	Not used (don't care)

19.5.11 Resume non-blocking

This function completes the additional phases of erase and program that were started using the non-blocking write row and non-blocking program row system calls. This function must be called thrice following a call to Non-Blocking Write Row or once following a call to Non-Blocking Program Row from the SPC ISR. No other system calls can execute until all phases of the program or erase operation are complete. More details on the procedure of using the non-blocking functions are explained in **"Blocking and non-blocking system calls"** on page 193.

Parameters

Address	Value to be written	Description
SRAM Address 32'hYY (32-bit wide, word-aligned SRAM address)		
Bits [7:0]	0xB6	Key1
Bits [15:8]	0xDC	Key2
Bits [31:16]	0xXXXX	Don't care. Not used by SROM
CPUSS_SYSARG register		
Bits [31:0]	32'hYY	32-bit word-aligned address of the SRAM that stores the first function parameter (key1)
CPUSS_SYSREQ register		
Bits [15:0]	0x0009	Resume Non-Blocking opcode
Bits [31:16]	0x8000	Set SYSCALL_REQ bit

Return

Address	Return value	Description
CPUSS_SYSARG register		
Bits [31:28]	0xA	Success status code
Bits [27:0]	0xXXXXXXX	Not used (don't care)

19.6 System call status

At the end of every system call, a status code is written over the arguments in the CPUSS_SYSARG register. A success status is 0xAXXXXXXX, where X indicates don't care values or return data in the case of the system calls that return a value. A failure status is indicated by 0xF00000XX, where XX is the failure code.

Table 19-2. System call status codes

Status code (32-bit value in CPUSS_SYSARG register)	Description
AXXXXXXXh	Success – The "X" denotes a don't care value, which has a value of '0' returned by the SROM, unless the API returns parameters directly to the CPUSS_SYSARG register.
F0000001h	Invalid Chip Protection Mode – This API is not available during the current chip protection mode.
F0000003h	Invalid Page Latch Address – The address within the page latch buffer is either out of bounds or the size provided is too large for the page address.
F0000004h	Invalid Address – The row ID or byte address provided is outside of the available memory.
F0000005h	Row Protected – The row ID provided is a protected row.
F0000007h	Resume Completed – All non-blocking APIs have completed. The resume API cannot be called until the next non-blocking API.
F0000008h	Pending Resume – A non-blocking API was initiated and must be completed by calling the resume API, before any other APIs may be called.
F0000009h	System Call Still In Progress – A resume or non-blocking is still in progress. The SPC ISR must fire before attempting the next resume.
F000000Ah	Checksum Zero Failed – The calculated checksum was not zero.
F000000Bh	Invalid Opcode – The opcode is not a valid API opcode.
F000000Ch	Key Opcode Mismatch – The opcode provided does not match key1 and key2.
F000000Eh	Invalid Start Address – The start address is greater than the end address provided.
F0000012h	Invalid Pump Clock Frequency - IMO must be set to 48 MHz and HF clock source to the IMO clock source before flash write/erase operations.

19.7 Non-blocking system call pseudo code

This section contains pseudo code to demonstrate how to set up a non-blocking system call and execute code out of SRAM during the flash programming operations.

```
#define REG(addr)(*((volatile uint32 *) (addr)))
#define CM0_ISER_REG REG( 0xE000E100 )
#define CPUSS_CONFIG_REGREG( 0x40100000 )
#define CPUSS_SYSREQ_REG REG( 0x40100004 )
#define CPUSS_SYSARG_REG REG( 0x40100008 )

#define ROW_SIZE_    ()
#define ROW_SIZE        (ROW_SIZE_)

/*Variable to keep track of how many times SPC ISR is triggered */
__ram int iStatusInt = 0x00;

__flash int main(void)
{
       DoUserStuff();

       /*CM0+ interrupt enable bit for spc interrupt enable */
       CM0_ISER_REG |= 0x00000040;

       /*Set CPUSS_CONFIG.VECS_IN_RAM because SPC ISR should be in SRAM */
       CPUSS_CONFIG_REG |= 0x00000001;

       /*Call non-blocking write row API */
       NonBlockingWriteRow();

       /*End Program */
       while(1);
}
__sram void SpcIntHandler(void)
{
              /* Write key1, key2 parameters to SRAM */
       REG( 0x20000000 ) = 0x0000DCB6;

       /*Write the address of key1 to the CPUSS_SYSARG reg */
       CPUSS_SYSARG_REG = 0x20000000;

       /*Write the API opcode = 0x09 to the CPUSS_SYSREQ.COMMAND
       * register and assert the sysreq bit
       */
       CPUSS_SYSREQ_REG = 0x80000009;

       /* Number of times the ISR has triggered */
       iStatusInt ++;
}
__sram void NonBlockingWriteRow(void)
{
       int iter;

       /*Load the Flash page latch with data to write*/
```

```
        * Write key1, key2, byte address, and macro sel parameters to SRAM
        */
        REG( 0x20000000 ) = 0x0000D7B6;

        //Write load size param (128 bytes) to SRAM
        REG( 0x20000004 ) = 0x0000007F;


        for(i = 0; i < ROW_SIZE/4; i += 1)
        {
               REG( 0x20000008 + i*4 ) = 0xDADADADA;
        }

        /*Write the address of the key1 param to CPUSS_SYSARG reg*/
        CPUSS_SYSARG_REG = 0x20000000;

        /*Write the API opcode = 0x04 to CPUSS_SYSREQ.COMMAND
        * register and assert the sysreq bit
        */
        CPUSS_SYSREQ_REG = 0x80000004;

        /*Perform Non-Blocking Write Row on Row 200 as an example.
        * Write key1, key2, row id to SRAM row id = 0xC8 -> which is row 200
        */
        REG( 0x20000000 ) = 0x00C8DAB6;

        /*Write the address of the key1 param to CPUSS_SYSARG reg */
        CPUSS_SYSARG_REG = 0x20000000;

        /*Write the API opcode = 0x07 to CPUSS_SYSREQ.COMMAND
        * register and assert the sysreq bit
        */
        CPUSS_SYSREQ_REG = 0x80000007;

        /*Execute user code until iStatusInt equals 3 to signify
        * 3 SPC interrupts have happened. This should be 1 in case
        * of non-blocking program System Call
        */
        while( iStatusInt != 0x03 )
        {
               DoOtherUserStuff();
        }

        /* Get the success or failure status of System Call*/
        syscall_status = CPUSS_SYSARG_REG;
}
```

In the code, the CM0+ exception table is configured to be in SRAM by writing 0x01 to the CPUSS_CONFIG register. The SRAM exception table should have the vector address of the SPC interrupt as the address of the *SpcIntHandler()* function, which is also defined to be in SRAM. See the **"Interrupts"** on page 30 for details on configuring the CM0+ exception table to be in SRAM. The pseudo code for a non-blocking program system call is also similar, except that the function opcode and parameters will differ and the iStatusInt variable should be polled for 1 instead of 3. This is because the SPC ISR will be triggered only once for a non-blocking program system call.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-07-12

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2023 Infineon Technologies AG.
All Rights Reserved.

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-34474 Rev. *B

IMPORTANT NOTICE
The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”).

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

WARNINGS
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.