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

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

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

EZ-PD™ PMG1 MCU Family

EZ-PD™ PMG1-S2 MCU Architecture Technical Reference Manual (TRM)

Document No. 002-31772 Rev. *A

July 30, 2021

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

Copyrights

Content Overview

Contents

Section A: Overview

EZ-PD™ PMG1-S2 MCU Technical Reference Manual consists of two books. This EZ-PD™ PMG1-S2 MCU Architecture Technical Reference Manual is Book 1 of 2. For details on registers, refer to EZ-PD™ PMG1-S2 MCU Registers Technical Reference Manual (Book 2 of 2).

This section encompasses the following chapters:

- Introduction chapter on page 11
- Getting Started chapter on page 17
- Document Construction chapter on page 18

Document Revision History

Table 1-1. Document Revision History

Revision	Issue Date	Description of Change
**	November 26, 2020	New EZ-PD™ PMG1-S2 MCU TRM.
*A	July 30, 2021	Updated 20.7 Non-Blocking System Call Pseudo Code.

1. Introduction

EZ-PD™ Power Delivery Microcontroller Gen1 (PMG1) is a family of high-voltage USB-C power delivery (PD) microcontrollers (MCU). These chips include an Arm® Cortex®-M0/M0+ CPU and USB-C PD controller along with analog and digital peripherals. EZ-PD™ PMG1 MCU is targeted at any embedded system that powers from a high-voltage USB-C PD port and leverages the microcontroller to provide additional control capability. Figure 1-1 shows the EZ-PD™ PMG1 MCU family segmentation.

Figure 1-1. EZ-PD™ PMG1 MCU Family Segmentation

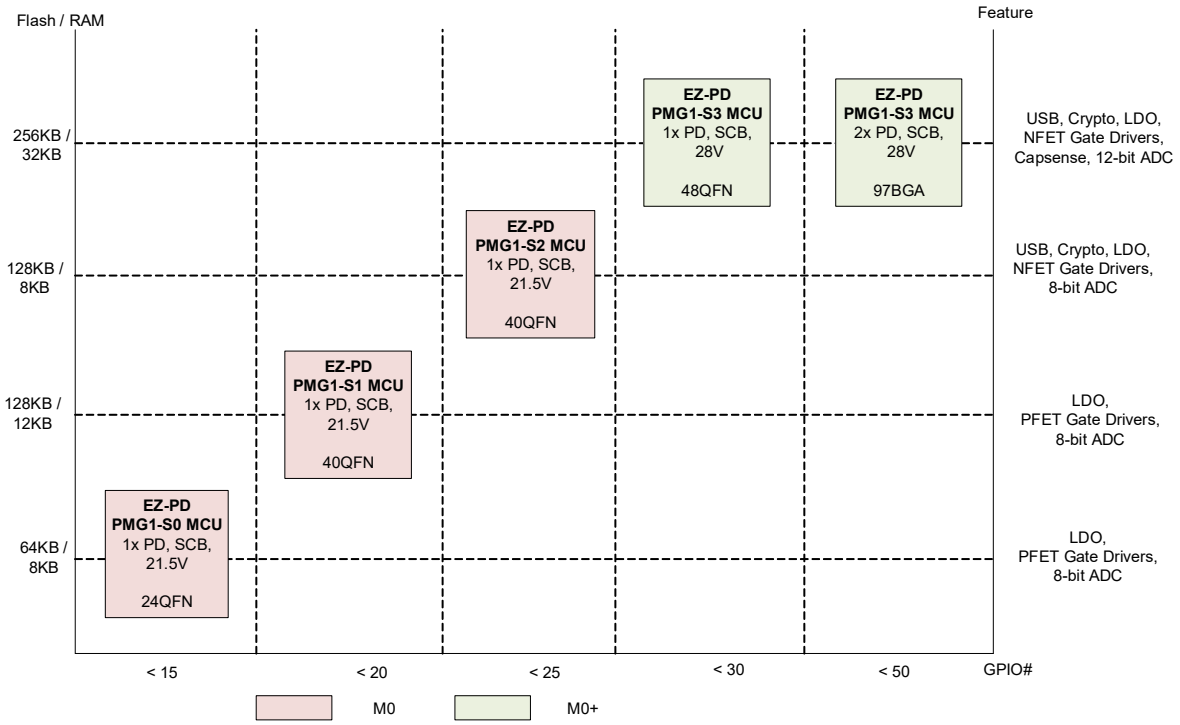


Table 1-1 shows the comparison of features of different MCUs of the EZ-PD™ PMG1 MCU family. The rest of the document discusses EZ-PD™ PMG1 MCU-S0 in detail.

Table 1-1. EZ-PD™ PMG1 MCU Family Feature Comparison

Subsystem or Range	Item	EZ-PD™ PMG1-S0 MCU	EZ-PD™ PMG1-S1 MCU	EZ-PD™ PMG1-S2 MCU	EZ-PD™ PMG1-S3 MCU
CPU & Memory Subsystem	Core	Arm Cortex-M0	Arm Cortex-M0	Arm Cortex-M0	Arm Cortex-M0+
	Max Freq (MHz)	48	48	48	48
	Flash (KB)	64	128	128	256
	SRAM (KB)	8	12	8	32
Power Delivery	Power Delivery Ports	1	1	1	2
	Role	Sink	DRP	DRP	DRP
	MOSFET Gate Drivers	2x PFET	2x PFET	2x PFET/NFET	2XNFET
	Fault Protections	VBUS OVP[a] and UVP[b]	VBUS OVP, UVP, and OCP[c]	VBUS OVP and OCP	VBUS OVP, OCP, SCP[d], and RCP[e]
USB	Integrated Full Speed USB 2.0 Device with Billboard Class support	No	No	Yes	Yes
Voltage Range	Supply (V)	VDDD (2.7 - 5.5) VBUS (4 - 21.5)	VSYS (2.75 - 5.5) VBUS (4 - 21.5)	VSYS (2.7 - 5.5) VBUS (4 - 21.5)	VSYS (2.8 - 5.5) VBUS (4 - 28)
	IO (V)	1.71 - 5.5	1.71 - 5.5	1.71 - 5.5	1.71 - 5.5
Digital	Serial communication blocks (configurable as I^2C/UART/SPI)	2	4	4	8
	TCPWM blocks (configurable as timer, counter or pulse width modulator)	4	2	4	8
	Hardware Authentication Block (Crypto)	No	No	Yes (AES[f], SHA[g], PRNG[h], CRC[i])	Yes (AES, SHA, TRNG[j], Vector Unit)
Analog	ADC	2x 8-bit SAR	1x 8-bit SAR	2x 8-bit SAR	2x 8-bit SAR 1x 12-bit SAR
	On-chip Temperature Sensor	Yes	Yes	Yes	Yes
DMA	DMA	No	No	No	Yes
GPIO	Max # of I/O	12 (10+2 OVT[k])	17 (15+2 OVT)	20 (18+2 OVT)	50
Charging Standards	Charging Standards	BC 1.2, Apple Charging (AC)	BC 1.2, AC	BC 1.2, AC	BC 1.2, AC, AFC and Quick Charge 3.0
ESD Protection	ESD Protection	Yes (Up to ± 8-kV contact discharge and up to ±15-kV air discharge)	Yes (Human Body Model Only)	Yes (Up to ± 8-kV contact discharge and up to ±15-kV air discharge)	Yes (Human Body model and Charged Device Model)
Packages	Package Options	24 QFN (4x4 mm, 0.5 mm pitch)	40 QFN (6×6 mm, 0.5 mm pitch)	40 QFN(6×6 mm, 0.5 mm pitch)	48QFN (6x6 mm, 0.5 mm pitch) 97BGA (6x6 mm, 0.5 mm, and 0.65 mm pitch)

a. Overvoltage Protection
b. Undervoltage Protection
c. Overcurrent Protection
d. Short Circuit Protection
e. Reverse Current Protection
f. Advanced Encryption Standard
g. Secure Hash Algorithm
h. Pseudo Random Number Generation
i. Cyclic Redundancy Check
j. True Random Number Generation
k. Over Voltage Tolerant

EZ-PD™ PMG1-S2 MCU has a 32-bit, 48-MHz Arm® Cortex® -M0 processor with 128-KB flash, 8-KB SRAM, 20 GPIOs, full-speed USB device controller, a Crypto engine for authentication, a 20 V-tolerant regulator, and a pair of 1W FETs to generate a 5 V (VCONN) supply, which powers cables. EZ-PD™ PMG1-S2 MCU also integrates two pairs of gate drivers to control external VBUS FETs and system level ESD protection.

EZ-PD™ PMG1-S2 MCU devices have these characteristics:

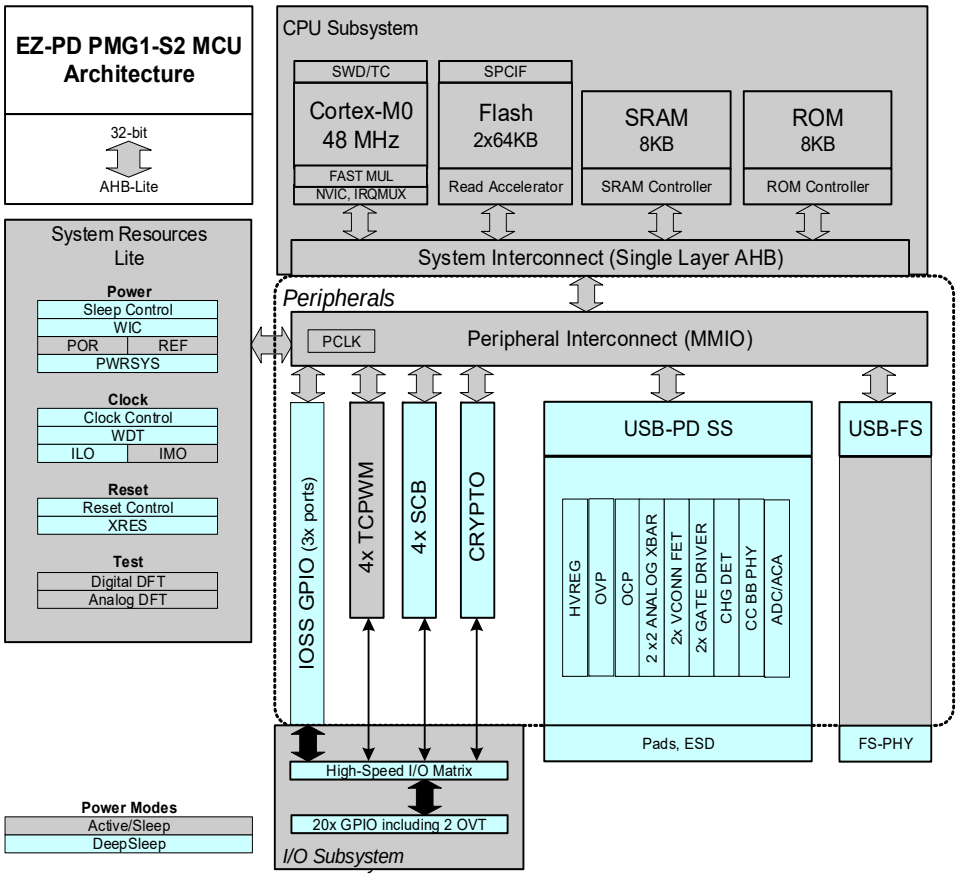
- High-performance, 32-bit 48 MHz Cortex-M0 CPU core
- Configurable Timer/Counter/PWM blocks
- Configurable Serial Communication blocks to support I^2C, SPI, UART
- Integrated USB-PD BMC transceiver
- Integrated VCONN FETs
- Configurable resistors R_P and R_D
- Hardware Crypto block enables Authentication
- Full-Speed USB Device Controller supporting Billboard Device Class

This document describes each function block of the EZ-PD™ PMG1-S2 MCU device in detail. This information will help designers to create system-level designs.

1.1 Top Level Architecture

Figure 1-2 shows the major components of the EZ-PD™ PMG1-S2 MCU architecture.

Figure 1-2. EZ-PD™ PMG1-S2 MCU Family Functional Block Diagram



1.2 Features

EZ-PD™ PMG1-S2 MCU family of USB Type-C cable controllers has these major components:

- 32-bit Cortex-M0 CPU at 48 MHz
- 128 KB flash and 8 KB SRAM
- Integrated USB-PD BMC transceiver
- Integrated VCONN FETs
- Hardware Crypto block to enable authentication
- Full-Speed USB device controller supporting billboard device class.
- Up to four dedicated timers and counters block to meet response times required by the USB-PD protocol
- Four independent run-time reconfigurable serial communication blocks (SCB) with reconfigurable I^2C, SPI, or UART functionality
- Supports one USB Type-C port
- Integrated oscillator eliminating the need of external clock
- Programming and debugging through serial wire debug (SWD)

1.3 CPU and Memory Subsystem

1.3.1 Processor

The Cortex M0 in EZ-PD™ PMG1-S2 MCU is a 32-bit MCU, which 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, which enables fully compatible binary upwards migration of code to higher performance processors such as the Cortex M3 and M4. The Cypress option includes a hardware multiplier, which provides a 32-bit result in one cycle. It includes an Interrupt Controller (the NVIC block) with 32 interrupt inputs and also includes a WIC (Wakeup Interrupt Controller), which can wake the processor up from Deep Sleep mode. The subsystem contains 128KB of Flash organized in two banks of 64KB. The two-bank flash allows the system to maintain two copies of the firmware and update one copy while continuing to execute from the other copy.

1.3.2 Interrupt Controller

The CPU subsystem of EZ-PD™ PMG1-S2 MCU includes a nested vectored interrupt controller (NVIC) with 32 interrupt inputs and a wakeup interrupt controller (WIC), which can wake the processor from Deep-Sleep mode. The Arm Cortex-M0 CPU provides a Non Maskable Interrupt (NMI) input, which is made available to the user when it is not in use for system functions requested by the user.

1.4 USB PD Subsystem

This subsystem provides the interface to the Type-C USB port. This subsystem comprises the USB-PD transceiver, the FS USB transceiver, the high voltage regulator, OVP, OCP and supply switch blocks. This subsystem also contains the analog switches, the VCONN FETs and the 8-bit ADC. This subsystem also includes all ESD required / supported on the Type-C port.

1.4.1 USB-PD physical layer

The USB-PD Physical Layer consists of a transmitter and receiver that communicate BMC encoded data over the CC channel per the PD 2.0 standard. All communication is half-duplex. The Physical Layer or PHY practices collision avoidance to minimize communication errors on the channel.

The USB-PD block includes all termination resistors (R_P and R_D) and their switches as required by the USB-PD specification. R_P and R_D resistors are required to implement connection detection, plug orientation detection, and for establishment of the

USB DFP/UFP roles. The R_P resistor is implemented as a current source. A dead battery R_D termination enables dead battery termination detection and charging while the EZ-PD™ PMG1-S2 MCU device is not powered

1.4.2 ADC

An 8-bit SAR ADC is available for general-purpose A-D conversion applications in the chip. The ADC can be accessed from all GPIOs and the DP/DM pins through an on-chip analog mux.

1.4.3 Analog Crossbar

EZ-PD™ PMG1-S2 MCU contains a set of analog switches to connect SBU1 and SBU2 pins of the Type-C connector to AUX_P and/or AUX_N of a display port connector. All four pins are provided with switchable pull-up and pull-down resistors as required by their respective specs.

1.4.4 Charger Detection

The charger detection block connected to the D+/D- pins allow EZ-PD™ PMG1-S2 MCU to detect conventional battery chargers conforming to BC 1.2 or other proprietary charger specifications.

1.5 System-Wide Resources

1.5.1 Clocking System

The clock system for the EZ-PD™ PMG1-S2 MCU controller consists of the internal main oscillator (IMO) and an internal low-speed oscillator (ILO) as internal clocks and has provision for an external clock.

The IMO is the primary source of internal clocking in EZ-PD™ PMG1-S2 MCU. It is trimmed during production to achieve the desired accuracy of ±2%. Trim values are stored in supervisory rows in the Flash memory. Additional trim settings from Flash can be used to compensate for changes. IMO default frequency for EZ-PD™ PMG1-S2 MCU is 24 MHz ±2%.

The ILO is a 32-kHz low-power, less accurate oscillator used to generate clocks for peripheral operation in Deep-Sleep power mode.

1.5.2 Power System

The power system provides assurance that voltage levels are as required for each respective mode and will either delay mode entry (on POR for instance) until voltage levels are as required for proper function or will generate Resets (BOD detection).

EZ-PD™ PMG1-S2 MCU operates from two possible external supply sources VSYS and VBUS. The VSYS supply supports operation over 2.7 - 5.5 V while the VBUS input supports operation over 4.0 - 21.5 V. The range of VSYS is valid for powered accessory and UFP applications not requiring a billboard device function. All DFP and DRP applications require VSYS to be over 3.0 V to ensure this device can support disconnect thresholds of 2.7 V on CC pins. All use cases requiring billboard function require VSYS or VBUS to be over 4.4 V.

EZ-PD™ PMG1-S2 MCU has three different power modes (Active, Sleep, and Deep Sleep) transitions between which are managed by the power system. A separate power domain VDDIO is provided for the GPIOs.

VDDD, the output of the VBUS regulator and VCCD, the output of the core (1.8 V) regulator, are brought out only for connecting external capacitors. These pins are not supported as a power supply sources.

1.5.3 GPIO

Twenty GPIOs are brought out on the 40-pin QFN package. Every GPIO in EZ-PD™ PMG1-S2 MCU has the following characteristics:

- Eight drive strength modes including strong push-pull, resistive pull-up and pull-down, weak (resistive) pull-up and pull-down, open drain and open source, input only, and disabled

- Input threshold select (CMOS or LVTTL)
- Individual control of input and output disables
- Hold mode for latching previous state (used for retaining I/O state in Deep Sleep mode)
- Selectable slew rates for dV/dt related noise control

The pins are organized in three logical entities called Ports. Port 0 contains the two OVT GPIOs. Ports 1, 2, and 3 are assigned 8, 7, and 7 GPIOs respectively. During power-on and reset, the blocks are forced to the Disable state so as not to crowbar any inputs and/or cause excess turn-on current. A multiplexing network known as 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 such as USB Type-C port are also fixed in order to reduce internal multiplexing complexity. Data Output Registers and Pin State Register store, respectively, the values to be driven on the pins and the states of the pins themselves. The configuration of the pins can be done by programming of registers through software for each digital I/O port.

Every I/O pin can generate an interrupt if so enabled and each I/O port has an Interrupt Request (IRQ) and Interrupt Service Routine (ISR) vector associated with it.

The I/O ports can retain their state during Deep Sleep mode or remain ON. If operation is restored using reset then the pins will go the High-Z state. If operation is restored by an interrupt event then the pin drivers will retain their state until firmware chooses to change it. The I/Os (on data bus) do not draw current on power down.

1.6 Fixed-Function Digital

1.6.1 Timer/Counter/PWM Block

The TCPWM block of EZ-PD™ PMG1-S2 MCU supports four timers. These timers are available for internal timer use by firmware or for providing PWM based functions on the GPIOs.

1.6.2 Serial Communication Block (SCB)

The EZ-PD™ PMG1-S2 MCU has four SCBs, which can each implement a serial communication interface as I^2C, universal asynchronous receiver/transmitter (UART), or serial peripheral interface (SPI).

The features of SCB includes:

- Standard I^2C multi-master and slave function
- Standard SPI master and slave function with Motorola, TI, and National (MicroWire) mode
- Standard UART transmitter and receiver function

1.6.3 Crypto

The EZ-PD™ PMG1-S2 MCU Crypto block provides cryptography functionality. It includes hardware acceleration blocks for AES (Advanced Encryption Standard) block cipher, SHA-1 (Secure Hash Algorithm) and SHA-2 hash, CRC (Cyclic Redundancy Check), and pseudo random number generation.

2. Getting Started

2.1 EZ-PD™ PMG1 MCU MCU Resources

This chapter provides the complete list of EZ-PD™ PMG1 MCU MCU resources that will help you get started with the device and design your applications with them. If you are new to EZ-PD™ PMG1 MCU, there is a wealth of data at www.cypress.com to help you to select the right EZ-PD™ PMG1 MCU device and quickly and effectively integrate it into your design.

The following is an abbreviated list of EZ-PD™ PMG1 MCU MCU resources:

- Overview: EZ-PD™ PMG1 MCU MCU webpage
- Datasheets describe and provide electrical specifications for each device family.
- Application Notes and Code Examples cover a broad range of topics, from basic to advanced level. Many of the application notes include code examples, which can be opened from ModusToolbox.
- Technical Reference Manuals (TRMs) provide detailed descriptions of the architecture and registers in each device family.
- Development Tools
  - ModusToolbox is a free integrated design environment (IDE). It enables you to design hardware and firmware systems concurrently with EZ-PD™ PMG1 MCU devices. In addition, ModusToolbox includes a device selection tool to select devices for ModusToolbox projects.
  - EZ-PD™ PMG1 MCU Prototyping Kits offer an easy-to-use, inexpensive platform to build EZ-PD™ PMG1 MCU-based systems.
- Additional Resources: Visit the EZ-PD™ PMG1 MCU MCU webpage for additional resources such as IBIS, BSDL models, CAD Library Files, and Programming Specifications.
- Technical Support
  - Forum: See if your question is already answered by fellow developers of the EZ-PD™ PMG1 MCU community.
  - Cypress support: Visit our support page or contact a local sales representative.

3. Document Construction

The following sections in this document include these topics:

- Section B: CPU System on page 23
- Section C: Memory System on page 38
- Section D: System-Wide Resources on page 41
- Section E: Digital System on page 64
- Section F: USB Power Delivery on page 159
- Section G: Program and Debug on page 162

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 about any particular area that inform the reader about the construction and organization of the product.
- Chapter – Presents the chapters specific to an individual aspect of the section topic. These are the detailed implementation and use information for some aspect of the integrated circuit.
- Glossary – Defines the specialized terminology used in this technical reference manual (TRM). Glossary terms are presented in bold, italic font throughout.
- Registers Technical Reference Manual – Supplies all device register details summarized in the technical reference manual. These are additional documents.

3.2 Documentation Conventions

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

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

3.2.1 Register Conventions

Register conventions are detailed in the EZ-PD™ PMG1-S2 MCU Registers TRM.

3.2.2 Numeric Naming

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

3.2.3 Units of Measure

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

Table 3-1. Units of Measure

Symbol	Unit of Measure
°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
σ	sigma: one standard deviation
V	volts

3.2.4 Acronyms

This table lists the acronyms used in this document

Table 3-2. Acronyms

Symbol	Unit of Measure
ABUS	analog output bus
AC	alternating current
ADC	analog-to-digital converter
AHB	AMBA (advanced microcontroller bus architecture) high-performance bus, an Arm data transfer bus

Table 3-2. Acronyms *(continued)*

Symbol	Unit of Measure
API	application programming interface
APOR	analog power-on reset
BMC	Bi-Phase Mark Coding
BR	bit rate
BRA	bus request acknowledge
BRQ	bus request
CAN	controller area network
CC	Configuration Channel
CI	carry in
CMP	compare
CO	carry out
CPU	central processing unit
CRC	cyclic redundancy check
CT	continuous time
DAC	digital-to-analog converter
DC	direct current
DFP	Downstream Facing Port
DI	digital or data input
DMA	direct memory access
DNL	differential nonlinearity
DO	digital or data output
DRP	Dual Role Port
DSI	digital signal interface
DSM	deep-sleep mode
ECO	external crystal oscillator
EEPROM	electrically erasable programmable read only memory
EMCA	Electronically Marked Cable Assembly
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
I^2C	inter-integrated circuit
IDE	integrated development environment
ILO	internal low-speed oscillator
IMO	internal main oscillator
INL	integral nonlinearity
I/O	input/output
IOR	I/O read
IOW	I/O write

Table 3-2. Acronyms *(continued)*

Symbol	Unit of Measure
IRES	initial power on reset
IRA	interrupt request acknowledge
IRQ	interrupt request
ISR	interrupt service routine
IVR	interrupt vector read
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
MSb	most significant bit
MSB	most significant byte
OVP	over voltage protection
PC	program counter
PCH	program counter high
PCL	program counter low
PD	power down
PGA	programmable gain amplifier
PM	power management
PMA	EZ-PD™ PMG1-S2 MCU memory arbiter
POR	power-on reset
PPOR	precision power-on reset
PRS	pseudo random sequence
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
RW	read/write
SAR	successive approximation register
SC	switched capacitor
SCB	serial communication block
SIE	serial interface engine
SIO	special I/O
SE0	single-ended zero
SNR	signal-to-noise ratio
SOF	start of frame
SOI	start of instruction

Table 3-2. Acronyms *(continued)*

Symbol	Unit of Measure
SOP	Start of Packet
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
TD	transaction descriptors
UART	universal asynchronous receiver/transmitter
UFP	Upstream Facing Port
USB	universal serial bus
USBIO	USB I/O
USB PD	USB Power Delivery
UVP	under voltage protection
WCO	watch crystal oscillator
WDT	watchdog timer
WDR	watchdog reset
XRES	external reset
XRES_N	external reset, active low

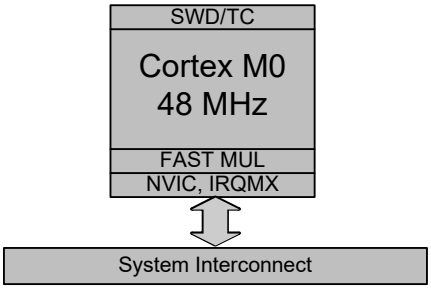
Section B: CPU System

This section encompasses the following chapters:

- Cortex-M0 CPU chapter on page 24
- Interrupts chapter on page 29

Top Level Architecture

CPU System Block Diagram



4. Cortex-M0 CPU

The EZ-PD™ PMG1-S2 MCU Arm Cortex-M0 core is a 32-bit CPU optimized for low-power operation. It has an efficient three-stage pipeline, a fixed 4-GB memory map, and supports the ARMv6-M Thumb instruction set. The Cortex-M0 also features a low-latency interrupt service routine (ISR) entry and exit.

The Cortex-M0 processor includes a number of other components that are tightly linked to the CPU core. These 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.

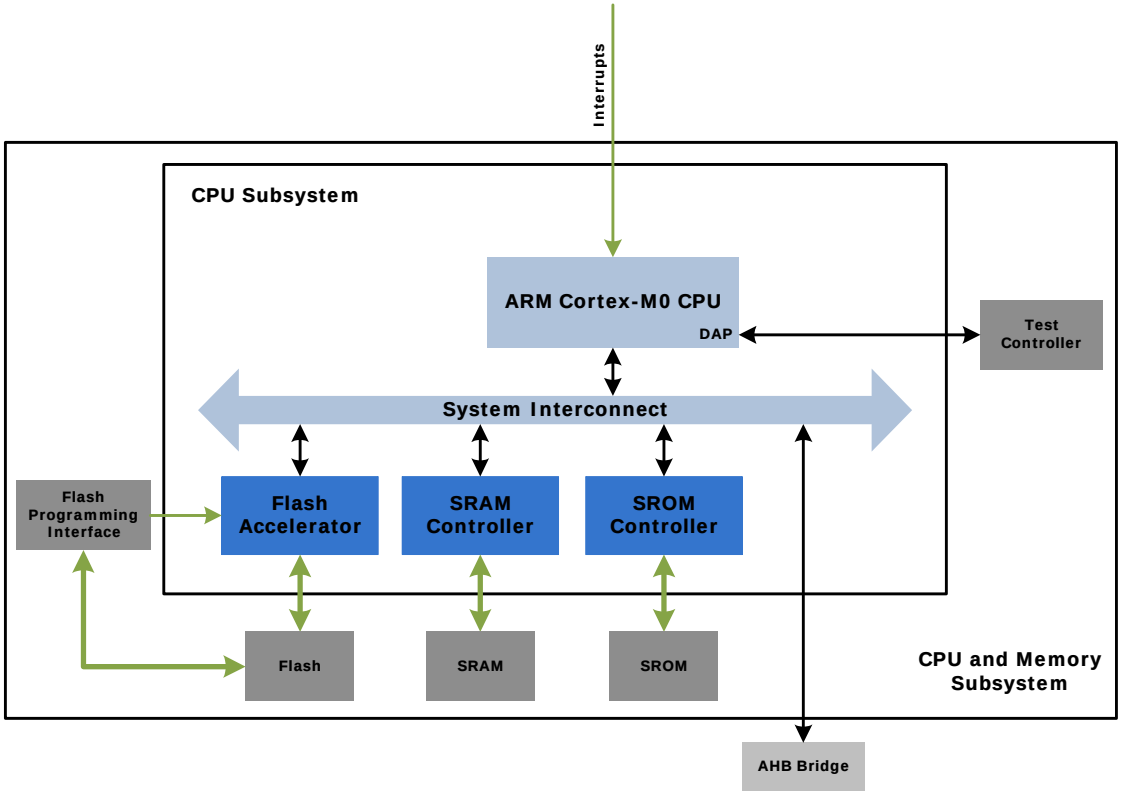
4.1 Features

The EZ-PD™ PMG1-S2 MCU Arm 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 0.9 DMIPS/MHz; this helps to increase execution speed or reduce power
- Supports Thumb instruction set for improved code density, ensuring efficient use of memory
- NVIC unit to support interrupts and exceptions for rapid and deterministic interrupt response
- Extensive debug support including:
  - Serial wire debug (SWD) port
  - Breakpoints
  - Watchpoints

4.2 Block Diagram

Figure 4-1. EZ-PD™ PMG1-S2 MCU 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.

4.3.1 Registers

The Cortex-M0 has 16 32-bit registers, as Table 4-1 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 4-1. Cortex-M0 Registers

Name	Type[a]	Reset Value	Description
R0-R12	RW	Unknown	R0-R12 are 32-bit general-purpose registers for data operations.
MSP	RW	[0x00000000]	The stack pointer (SP) is register R13. In thread mode, bit[1] of the CONTROL register indicates the stack pointer to use: 0 = Main stack pointer (MSP). This is the reset value. 1 = Process stack pointer (PSP). On reset, the processor loads the MSP with the value from address 0x00000000.
PSP			
LR	RW	Unknown	The link register (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions.
PC	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	Unknown[b]	The program status register (PSR) combines: Application Program Status Register (APSR). Execution Program Status Register (EPSR). Interrupt Program Status Register (IPSR).
APSR	RW	Unknown	The APSR contains the current state of the condition flags from previous instruction executions.
EPSR	RO	Unknown[b]	The EPSR contains the Thumb state bit.
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.
b. Bit[24] is the T-bit and is loaded from bit[0] of the reset vector.

Table 4-2 shows how the PSR bits are assigned.

Table 4-2. 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	APSE	V	Overflow flag
27–25	–	–	Reserved
24	EPSR	T	Thumb state bit. Must always be 1. Attempting to execute instructions when the T bit is 0 results in a HardFault exception.
23–6	–	–	Reserved
5–0	IPSR	N/A	Exception number of current ISR: 0 = thread mode 1 = reserved 2 = Non maskable interrupt (NMI) 3 = HardFault 4 – 10 = reserved 11 = SVCall 12, 13 = reserved 14 = PendSV 15 = SysTick 16 = IRQ0 … 47 = IRQ31

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 chapter on page 29 for a list of exceptions.

4.3.2 Operating Modes

The Cortex-M0 processor supports two operating modes:

- Thread Mode – used by all normal applications. In the thread 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 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.

4.3.3 Instruction Set

The Cortex-M0 implements a version of the Thumb instruction set. 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 4-3. Thumb Instruction Set

Mnemonic	Brief Description
ADCS	Add with Carry
ADD{S}	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}	Move

Table 4-3. Thumb Instruction Set

Mnemonic	Brief Description
MRS	Move to general register from special register
MSR	Move to special register from general register
MULS	Multiply, 32-bit result
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}	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

4.3.3.1 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.

4.3.3.2 Memory Endianness

The EZ-PD™ PMG1-S2 MCU Arm Cortex-M0 uses 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.

4.3.4 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 the Cortex-M0 internal clock as a source and can work in Sleep mode.

4.3.5 Debug

EZ-PD™ PMG1-S2 MCU Arm Cortex-M0 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 EZ-PD™ PMG1-S2 MCU 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. EZ-PD™ PMG1-S2 MCU provides a unified exception vector table for both interrupt handlers/ISR and exception handlers.

5.1 Features

EZ-PD™ PMG1-S2 MCU supports the following interrupt features:

- Supports 32 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

Figure 5-1. Interrupts Block Diagram

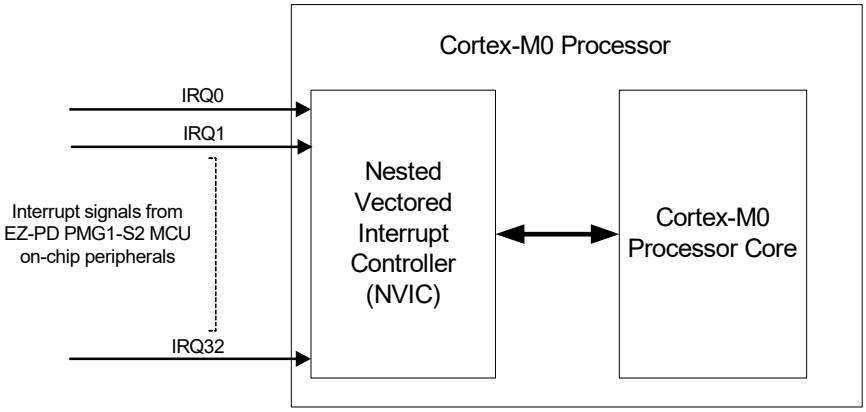


Figure 5-1 shows the interaction between interrupt signals and the Cortex-M0 CPU. EZ-PD™ PMG1-S2 MCU has 32 interrupts; these interrupt signals are processed by the NVIC. The NVIC takes care of enabling/disabling individual interrupts, priority resolution, and communication with the CPU core. The exceptions 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.

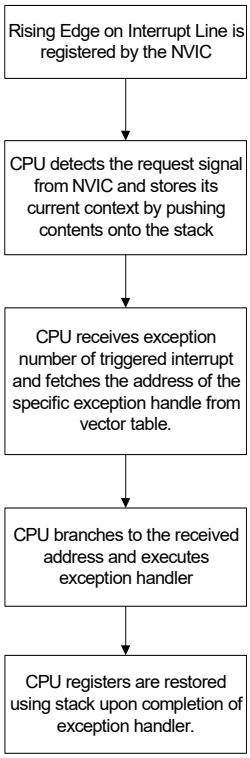
5.3 Interrupts and Exceptions - Operation

5.3.1 Interrupt/Exception Handling in EZ-PD™ PMG1-S2 MCU

The sequence of events that occur when an interrupt or exception event is triggered is:

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 in EZ-PD™ PMG1-S2 MCU 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.

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

EZ-PD™ PMG1-S2 MCU NVIC supports both level and pulse signals on the interrupt lines (IRQ0 to IRQ31). The classification of an interrupt as level or pulse is based on the interrupt source.

Figure 5-3. Level Interrupts

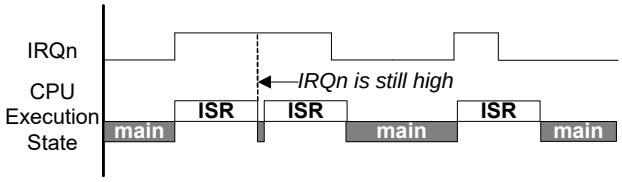


Figure 5-4. Pulse Interrupts

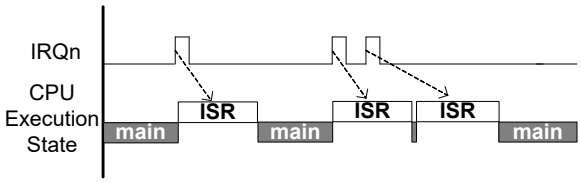


Figure 5-3 and Figure 5-4 show the working of level and pulse interrupts, respectively. Assuming the interrupt signal is initially inactive (logic low), the following sequence of events explains the handling of level and pulse interrupts:

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 in EZ-PD™ PMG1-S2 MCU. The CPU fetches the appropriate address based on the exception number.

Table 5-1. EZ-PD™ PMG1-S2 MCU Exception Vector Table

Exception Number	Exception	Exception Priority	Vector Address
–	Initial Stack Pointer Value	Not applicable (NA)	Base_Address - Can be 0x00000000 (start of flash memory) or 0x20000000 (start of SRAM)[a]
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)	…
47	External Interrupt (IRQ31)	Configurable (0–3)	Base_Address + 0xBC

a. Note that the reset exception address in SRAM vector table will never be used because the device comes out of reset with the flash vector table selected.

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.

In EZ-PD™ PMG1-S2 MCU, the vector table can be configured to be located either in flash memory (base address of 0x00000000) or SRAM (base address of 0x20000000). This configuration is done by writing to the VECT_IN_RAM bit field (bit 0) in the CPUSS_CONFIG register. When the VECT_IN_RAM bit field is '1', CPU fetches exception handler addresses from the SRAM vector table location. When this bit field is '0' (reset state), the vector table in flash memory is used for exception address fetches. You must set the VECT_IN_RAM bit field as part of the device boot code to configure the vector table to be in SRAM. 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.

The exception sources (exception numbers 1 to 15) are explained in 5.4 Exception Sources. The exceptions marked as Reserved in Table 5-1 are not used in EZ-PD™ PMG1-S2 MCU, though they have addresses reserved for them in the vector table. The interrupt sources (exception numbers 16 to 47) are explained in 5.5 Interrupt Sources.

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 EZ-PD™ PMG1-S2 MCU. 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 Cypress, 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 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 EZ-PD™ PMG1-S2 MCU:

- **NMI exception by setting NMIPENDSET bit (user NMI exception):** NMI exception can be triggered in software by setting the NMIPENDSET bit in the interrupt control state register (CM0_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 in EZ-PD™ PMG1-S2 MCU 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 EZ-PD™ PMG1-S2 MCU uses a privileged mode for the system call NMI exception, which is not related to the SVCall exception. (See the Chip Operational Modes chapter on page 57 for details on privileged mode.) There is no other privileged mode support for SVCall at the architecture level in EZ-PD™ PMG1-S2 MCU. 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, CM0_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, CM0_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 (CM0_SHPR3). See the ARMv6-M Architecture Reference Manual for more details.

5.4.6 SysTick Exception

CM0 CPU in EZ-PD™ PMG1-S2 MCU supports a system timer, referred to as SysTick, as part of its internal architecture. SysTick provides a simple, 24-bit decrementing counter for various time keeping 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 (CM0_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, CM0_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, CM0_ICSR.

5.5 Interrupt Sources

EZ-PD™ PMG1-S2 MCU supports 22 interrupts from peripherals. The source of each interrupt is listed in Table 5-2. EZ-PD™ PMG1-S2 MCU provides flexible sourcing options for each of the 22 interrupt lines. The interrupts include standard interrupts from on-chip peripherals such as TCPWM, serial communication block, and interrupts from ports. 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. These interrupts are usually level interrupts, which require the peripheral status register to 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.

Table 5-2. List of EZ-PD™ PMG1-S2 MCU Interrupt Sources

Interrupt No.	Interrupt Source
NMI	
0	GPIO Interrupt - Port0
1	GPIO Interrupt - Port1
2	GPIO Interrupt - Port2
3	GPIO Interrupt - Port3
4	GPIO Interrupt - Port4 USB Wakeup
5	GPIO All ports
6	Ganged USBPD Interrupt
7	WDT or Temp (WDT only in Deep Sleep)
8	SCB[0]
9	SCB[1]
10	SCB[2]
11	SCB[3]
12	SPC
13	Synchronous USBPD Interrupts

Table 5-2. List of EZ-PD™ PMG1-S2 MCU Interrupt Sources

Interrupt No.	Interrupt Source
14	TCPWM counter #0
15	TCPWM counter #1
16	TCPWM counter #2
17	TCPWM counter #3
18	USB Start of Frame
19	USB EP1-EP8 data
20	USB EP1-EP8 data
21	Crypto block

See the I/O System chapter on page 42 for details on GPIO interrupts.

5.6 Exception Priority

Exception priority is useful for exception arbitration when there are multiple exceptions that need to be serviced by the CPU. EZ-PD™ PMG1-S2 MCU provides flexibility in choosing priority values for different exceptions. All exceptions except 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 EZ-PD™ PMG1-S2 MCU, 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.

EZ-PD™ PMG1-S2 MCU 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 EZ-PD™ PMG1-S2 MCU 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 32.

The priority of the 32 interrupts (IRQ0–IRQ31) can be configured by writing to the Interrupt Priority registers (CM0_IPR). This is a group of four 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/Disabling Interrupts

The NVIC provides registers to individually enable and disable the 32 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 (CM0_ISER) and the Interrupt Clear-Enable Register (CM0_ICER) are used to enable and disable the interrupts respectively. These registers are 32-bit wide and each bit corresponds to the same numbered interrupt. 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 (CM0_ISER)	Write	1	To enable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled
Interrupt Clear Enable Register (CM0_ICER)	Write	1	To disable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled

The CM0_ISER and CM0_ICER registers are applicable only for the interrupts (IRQ0–IRQ31). 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 32.

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

The Interrupt Control State Register (CM0_ICSR) contains status bits describing the various exceptions states.

- The VECTACTIVE bits ([8:0]) in the CM0_ICSR store the exception number for the current executing exception. This value is zero if the CPU is not executing 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 Control State Register (ICSR), which is also used to store the active exception number.
- The VECTPENDING bits ([20:12]) in the CM0_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 CM0_ICSR indicates if a NVIC generated interrupt (IRQ0 to IRQ31) 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 32 interrupt lines by providing separate register bits for setting and clearing the pending states of the interrupts. The Interrupt Set-Pending Register (CM0_ISPR) and the Interrupt Clear-Pending Register (CM0_ICPR) are used to set and clear the pending status of the interrupt lines. These registers are 32 bits wide, and each bit corresponds to the same numbered interrupt. 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 (CM0_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 (CM0_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 CM0_ISER register.

Note that the CM0_ISPR and CM0_ICPR registers are used only for the 32 peripheral interrupts (exception numbers 16-47). 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 32.

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 chapter on page 24 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 done as part of the internal processor architecture (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419c/index.html).

5.10 Interrupts and Low-Power Modes

EZ-PD™ PMG1-S2 MCU 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 Supply and Modes chapter on page 55.

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, I^2C interrupts, USB PD, and GPIO interrupts)
- Fixed-function interrupt sources that are available only in the Active mode (all other fixed-function interrupts)

5.11 Exception - Initialization and Configuration

This section covers the different steps involved in initializing and configuring exceptions in EZ-PD™ PMG1-S2 MCU.

1. Configuring the Exception Vector Table Location: The first step in using exceptions is to configure the vector table location as required - either in flash memory or SRAM. This configuration is done by writing either a '1' (SRAM vector table) or '0' (flash vector table) to the VECT_IN_RAM bit field (bit 0) in the CPUSS_CONFIG register. 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 will need 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.
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 34.
   d. Enable the exception, as explained in Enabling/Disabling Interrupts on page 35.

5.12 Registers

Table 5-7. List of Registers

Register Name	Description
CM0_ISER	Interrupt Set-Enable Register
CM0_ICER	Interrupt Clear Enable Register
CM0_ISPR	Interrupt Set-Pending Register
CM0_ICPR	Interrupt Clear-Pending Register
CM0_IPR	Interrupt Priority Registers
CM0_ICSR	Interrupt Control State Register
CM0_AIRCR	Application Interrupt and Reset Control Register
CM0_SCR	System Control Register
CM0_CCR	Configuration and Control Register
CM0_SHPR2	System Handler Priority Register 2
CM0_SHPR3	System Handler Priority Register 3
CM0_SHCSR	System Handler Control and State Register
CM0_SYST_CSR	Systick Control and Status
CPUSS_CONFIG	CPU Subsystem Configuration
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.

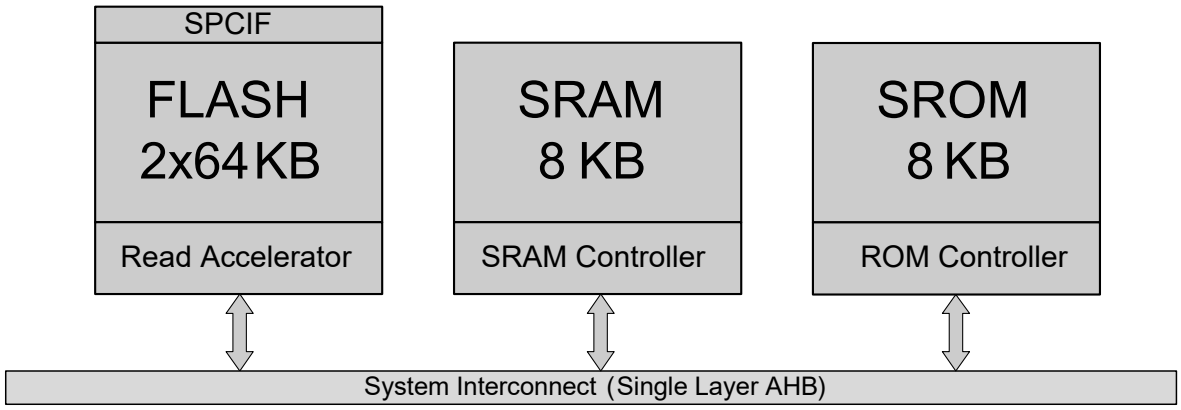
Section C: Memory System

This section presents the following chapter:

- Memory Map chapter on page 39

Top Level Architecture

Memory System Block Diagram



6. Memory Map

All EZ-PD™ PMG1-S2 MCU memory (flash, SRAM, and SROM) and all registers are accessible by the CPU and in most cases by the debug system. This chapter contains an overall map of the addresses of the memories and registers.

6.1 Features

The EZ-PD™ PMG1-S2 MCU memory system has the following features:

- 128K bytes flash, 8K bytes SRAM
- 8K byte SROM contains boot and configuration routines
- Arm Cortex-M0 32-bit linear address space, with regions for code, SRAM, peripherals, and CPU internal registers
- Flash is mapped to the Cortex-M0 code region
- SRAM is mapped to the Cortex-M0 SRAM region
- Peripheral registers are mapped to the Cortex-M0 peripheral region
- The Cortex-M0 Private Peripheral Bus (PPB) region includes registers implemented in the CPU core. These include registers for NVIC, SysTick timer, and serial communication block (SCB). For more information, see the Cortex-M0 CPU chapter on page 24.

6.2 How It Works

The EZ-PD™ PMG1-S2 MCU memory map is detailed in the following tables. For additional information, refer to the EZ-PD™ PMG1-S2 MCU Registers TRM.

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 6-1. Note that code can be executed from the code and SRAM regions.

Table 6-1. Cortex-M0 Address Map

Address Range	Name	Use
0x00000000–0x1FFFFFFF	Code	Executable region for program code. You can also put data here. Includes the exception vector table, which starts at address 0.
0x20000000–0x3FFFFFFF	SRAM	Executable region for data. You can also put code here.
0x40000000–0x5FFFFFFF	Peripheral	All peripheral registers. Code cannot be executed out of this region.
0x60000000–0xDFFFFFFF	–	Not used
0xE0000000–0xE00FFFFF	PPB	Peripheral registers within the CPU core.

Table 6-2 shows the EZ-PD™ PMG1-S2 MCU address map.

Table 6-2. EZ-PD™ PMG1-S2 MCU Address Map

Address Range	Use
0x00000000–0x0001FFFC	128 KB flash
0x0FFFF000–0x0FFFF3FF	1 KB supervisory flash
0x20000000–0x20001FFE	8 KB SRAM
0x40100000–0x4011FFFF	CPU subsystem registers
0x40020000–0x40023FFF	I/O port control (high-speed I/O matrix) registers
0x40040000–0x40043FFF	I/O port registers
0x40070000–0x4007FFFF	TCPWM registers
0x40050000–0x4005FFFF	SCB registers
0x40080000–0x4008FFFF	USB Power Delivery Controller registers
0x40030000–0x4003FFFF	Power, clock, reset control registers
0xE0000000–0xE00FFFFF	Cortex-M0 PPB registers
0xF0000000–0xF0000FFF	CoreSight ROM

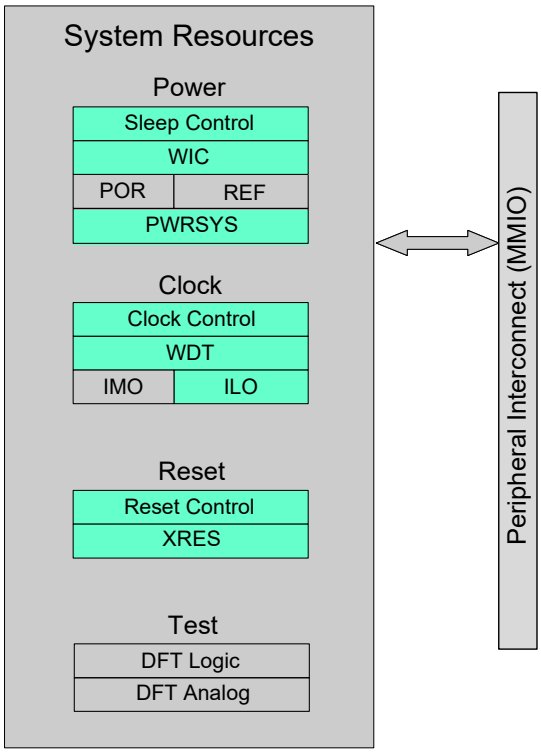
Section D: System-Wide Resources

This section encompasses the following chapters:

- I/O System chapter on page 42
- Clocking System chapter on page 50
- Power Supply and Modes chapter on page 55
- Chip Operational Modes chapter on page 57
- Watchdog Timer chapter on page 58
- Reset System chapter on page 61
- Device Security chapter on page 63

Top Level Architecture

System-Wide Resources Block Diagram



7. I/O System

This chapter explains the EZ-PD™ PMG1-S2 MCU I/O system, its features, architecture, operating modes, and interrupts. The general-purpose I/O (GPIOs) pins in EZ-PD™ PMG1-S2 MCU are grouped into ports; a port can have a maximum of eight GPIOs. The EZ-PD™ PMG1-S2 MCU die has a maximum of 20 GPIOs arranged in four ports.

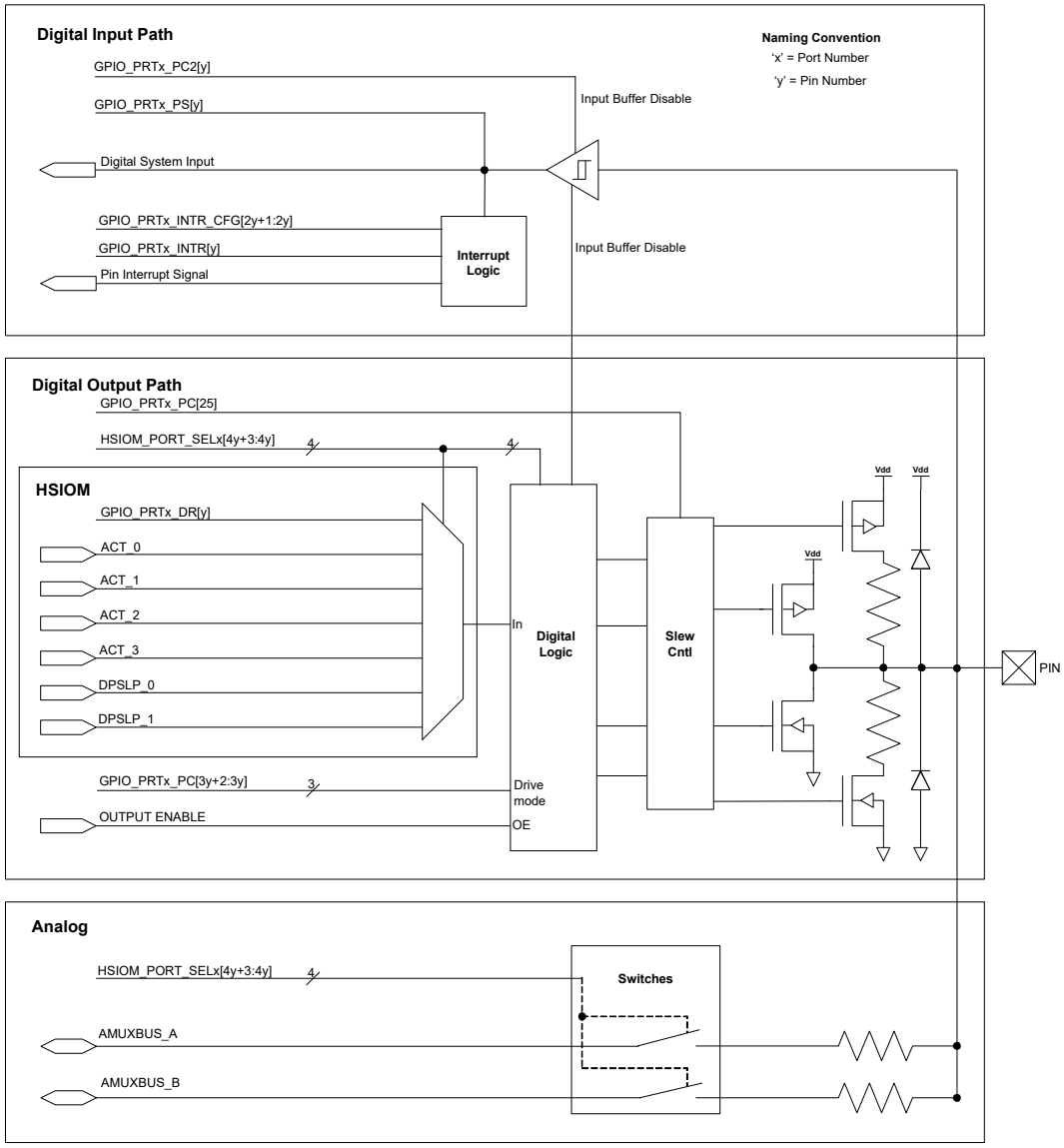
7.1 Features

The EZ-PD™ PMG1-S2 MCU GPIOs have these features:

- Analog and digital input and output capabilities
- 10-mA sink and 4-mA source current in digital mode
- Separate port read (PS) and write (DR) data registers to avoid read-modify-write errors
- Edge-triggered interrupts on rising edge, falling edge, or on both the edges, on pin basis
- Slew rate control
- Selectable CMOS and low-voltage LVTTL input buffer mode
- Two overvoltage tolerant GPIOs (I^2C pins from SCB0)

7.2 Block Diagram

Figure 7-1. GPIO Block Diagram



Note The GPIO features shown in this image may not be available on all the pins.

7.3 GPIO Drive Modes

Each I/O is individually configurable into one of the eight drive modes listed in Table 7-1. Figure 7-2 is a simplified pin diagram that shows the pin view based on each of the eight drive modes.

Two port configuration registers are used to configure GPIOs in EZ-PD™ PMG1-S2 MCU: Port Configuration Register (GPIO_PRTx_PC) and Port Secondary Configuration Register (GPIO_PRTx_PC2). All EZ-PD™ PMG1-S2 MCU devices have ports with dedicated GPIO_PRTx_PC and GPIO_PRTx_PC2 registers.

GPIO_PRTx_PC is used to configure the following properties of a port:

- Output drive mode of each pin (three bits select a particular drive mode for a pin)
- Slew rate of the whole port (see Slew Rate Control on page 46)
- Input threshold selection of the whole port (see CMOS LVTTL Level Control on page 46)

GPIO_PRTx_PC2 is used to enable/disable the input buffer of each pin on the port, irrespective of the drive mode configured in GPIO_PRTx_PC. When analog signals are present on the pin, input buffer should be disabled by setting the bit to '1'.

Figure 7-2. I/O Drive Mode Block Diagram

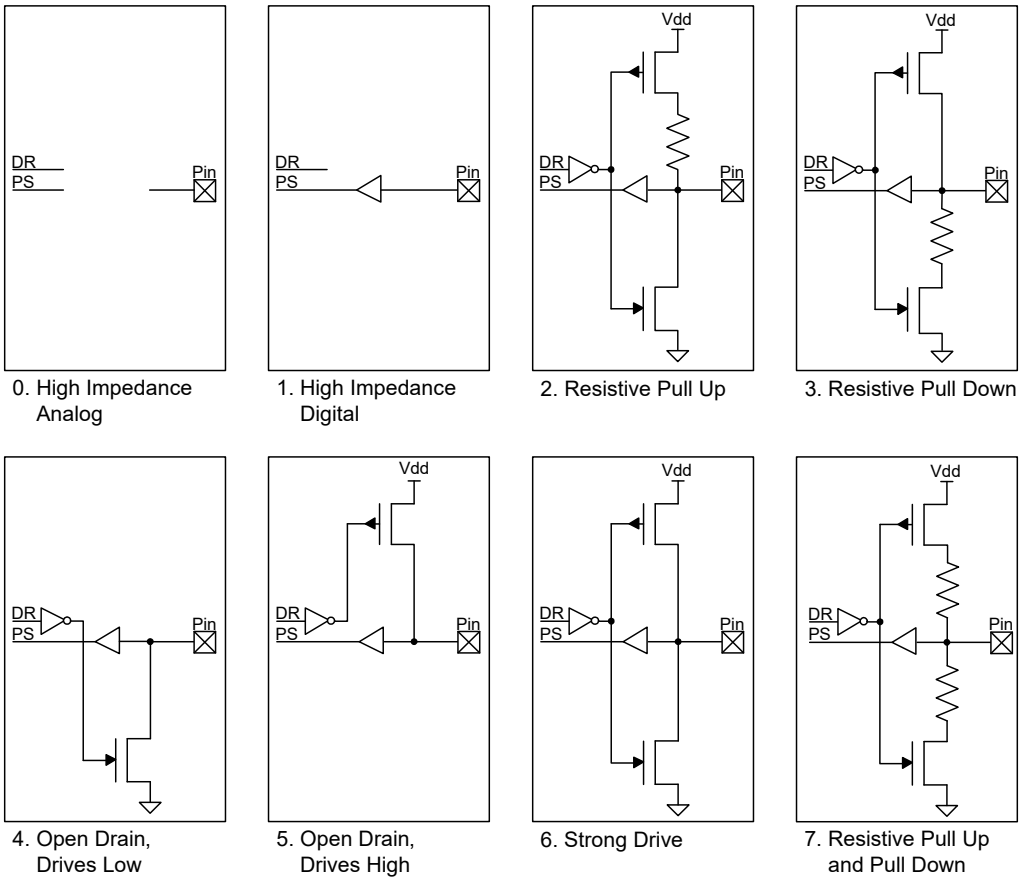


Table 7-1. Drive Mode Settings

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' ($0 \le y \le 7$)		
	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

Table 7-2. Input Buffer Disable (Port Configuration 2)

GPIO_PRTx_PC2 ('x' denotes port number and 'y' denotes pin number)		
Bits	**Name**	**Description**
7:0	INP_DIS	Disables the input buffer independent of the port control drive mode. This bit should be set when analog signals are present on the pin.

7.3.1 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.

7.3.2 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.

7.3.3 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 EZ-PD™ PMG1-S2 MCU 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.

7.3.4 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^2C bus signal lines.

7.3.5 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 digi-

tal output signals or to drive external transistors.

7.3.6 Resistive Pull-Up and Resistive Pull-Down

This mode is similar to the drive modes explained in 7.3.3 Resistive Pull-Up or 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.

7.4 Slew Rate Control

GPIO pins have fast and slow output slew rate options in strong drive mode; this can be configured using the GPIO_PRTx_PC[25] bit. 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. The fast slew rate is recommended for signals higher than 1 MHz. Slower slew rate results in reduced EMI and crosstalk; hence, the slow option is recommended for signals that are not speed critical – generally less than 1 MHz.

7.5 CMOS LVTTL Level Control

I/O pins can work at two voltage levels. These levels can be selected by writing to the GPIO_PRTx_PC[24] bit.

Input level is individually configurable for each port. This bit is cleared by default and the port works in CMOS mode. This bit can be set to reconfigure the port to LVTTL mode.

CMOS mode can be used in most cases, whereas LVTTL can be used for custom interface requirements, which works at lower voltage levels. See the device datasheet for the input voltage thresholds (VIH and VIL) for the modes.

7.6 GPIO-OVT

EZ-PD™ PMG1-S2 MCU device has two over-voltage tolerant (OVT) pins – I^2C pins from SCB0. It is similar to regular GPIOs with the following additional features:

- Over-voltage tolerant
- Provides better pull-down drive strength
- Serial Communication Block (SCB) when configured as I^2C and its lines routed to GPIO-OVT pins; it meets the following I^2C specifications:
  - Fast Mode hot-swap
  - Fast Mode Plus IOL Specification
  - Fast Mode and Fast Mode Plus Hysteresis and minimum fall time specifications

See the EZ-PD™ PMG1-S2 MCU datasheet for specifications.

7.7 High-Speed I/O Matrix

High-speed I/O matrix (HSIOM) is a group of high-speed switches that routes GPIOs to the resources inside EZ-PD™ PMG1-S2 MCU. These resources include TCPWMs, SCB, and the USB PD. The HSIOM selects Active and Deep-Sleep power domain sources for a pin. HSIOM_PORT_SELx are 32-bit wide registers that control the routing of GPIOs. Each register controls one port; four dedicated bits are assigned to each GPIO in the port. This provides up to 16 different options for GPIO routing. This selection provides different pin functions, as listed in Table 7-3.

Table 7-3. 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 ≤ y ≤ 7)**
4y+3 : 4y	DR	0	Pin is regular firmware-controlled GPIO.
	AMUXA	6	Pin is connected to AMUXBUS-A.
	AMUXB	7	Pin is connected to AMUXBUS-B. This mode is also used for GPIO pre-charging of tank capacitors.
	ACT_0	8	Pin-specific Active source # 0 (TCPWM, EXT CLOCK)
	ACT_1	9	Pin-specific Active source #1.
	ACT_2	10	Pin-specific Active source #2.
	ACT_3	11	Pin-specific Active source #3.
	DPSLP_0	14	Pin-specific Deep-Sleep source #0 (SCB - I^2C)
	DPSLP_1	15	Pin-specific Deep-Sleep source #1 (SWD)

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.

7.8 Firmware Controlled GPIO

See Table 7-3 to know the HSIOM settings for a firmware controlled GPIO. 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, two other registers - GPIO_PRTx_DR_SET and GPIO_PRTx_DR_CLR - are provided to set or clear the output data of specific GPIOs in port. Additionally, the GPIO_PRTx_DR_INV register can be used to invert the output data of a specific GPIO.

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

See the EZ-PD™ PMG1-S2 MCU Registers TRM for the details of these registers.

7.9 I/O Port Reconfiguration

Drive mode and GPIO can be reconfigured in runtime by changing the value of the GPIO_PRTx_PC and HSIOM-_PORT_SELx registers. Take care to retain the pin state during reconfiguration of pins when they are connected directly to a digital peripheral. If the ports are driven by the data registers, state maintenance is automatic. During port configuration, the current configuration should be saved as follows:

1. Read the GPIO pin state - GPIO_PRTx_PS in software.
2. Write the GPIO_PRTx_PS value into the data registers - GPIO_PRTx_DR.
3. Change the corresponding field in HSIOM_PORT_SELx to drive the pin by the data register - GPIO_PRTx_DR.

7.10 I/O State on Power Up

By default, during power up all GPIOs are in high-impedance analog state and input buffers are disabled. When the chip is powered, its GPIOs can be configured according to the required application, by writing to the associated registers.

7.11 Behavior in Low-Power Modes

The GPIOs maintain the current pin state during Sleep mode. In Sleep mode, all the GPIOs are active and can be driven by active peripherals, such as USB PD, TCPWM, and SCB.

In Deep-Sleep mode, all the pin states are latched and the pin signals are retained, except the SCB pins, which remain func-

tional and can wake up the processor on I^2C address matching event. GPIO interrupts are also available in Deep-Sleep mode with wake up ability.

For more details, see the Power Supply and Modes chapter on page 55, Interrupts chapter on page 29, and Inter Integrated Circuit (I2C) chapter on page 88.

To achieve the lowest device current in low-power modes, unused I/Os must be configured in the high-impedance analog mode.

7.12 GPIO Interrupt

This section describes the interrupt functionality of the EZ-PD™ PMG1-S2 MCU GPIOs.

7.12.1 Features

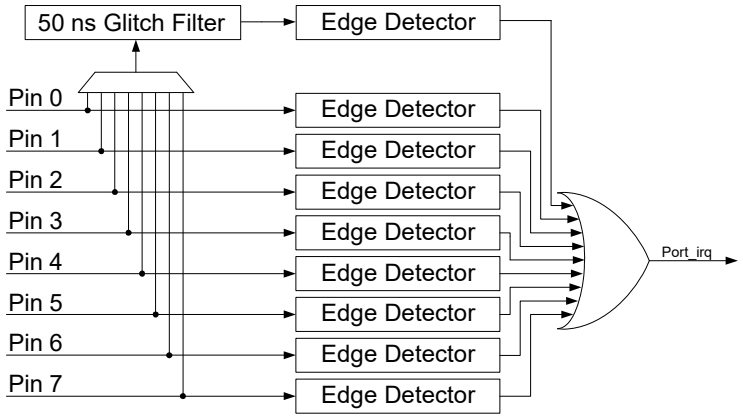
The features of the GPIO interrupt are:

- All eight pins in each port interface have an interrupt and an associated interrupt vector
- Pin status bits provide easy determination of interrupt source down to the pin level
- Rising, falling, or both edge-triggered interrupts are handled
- Pin interrupts can be individually enabled or disabled
- AHB interfaces for read and write into its registers
- Sends out a single port interrupt request (PIRQ) signal, derived from all GPIOs in a port, to the interrupt controller

7.12.2 Interrupt Controller Block Diagram

Each port has its own individual interrupt request and associated interrupt request (IRQ) vector and interrupt service routine (ISR). Additionally, one pin can be selected on each port that is routed through a 50-ns glitch filter to form a glitch-tolerant interrupt for the port. The details are shown in Figure 7-3.

Figure 7-3. Interrupt Generator



7.12.3 Function and Configuration

Each pin of the port can be configured independently to generate an interrupt on the rising edge, falling edge, or on both edges by writing to the GPIO_PRTx_INTR_CFG register. Level-sensitive interrupts are not supported. GPIO_PRTx_INTR_CFG is also used to route a specific channel to the glitch filter and generate a ninth glitch-tolerant interrupt.

When a GPIO interrupt is triggered by a signal on an interrupt-enabled port pin, the GPIO_PRTx_INTR register (Port Interrupt Status Register) is updated. The firmware can read this register to determine which GPIO triggered the interrupt. Firmware can then clear the IRQ bit by writing a '1' to its corresponding bit.

Additionally, when the Port Interrupt Control Status Register is read at the same time 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.

See GPIO_PRTx_INTR_CFG and GPIO_PRTx_INTR in the EZ-PD™ PMG1-S2 MCU Registers TRM for details.

7.13 Registers

Table 7-4. I/O 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 - Used to read 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

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

8. Clocking System

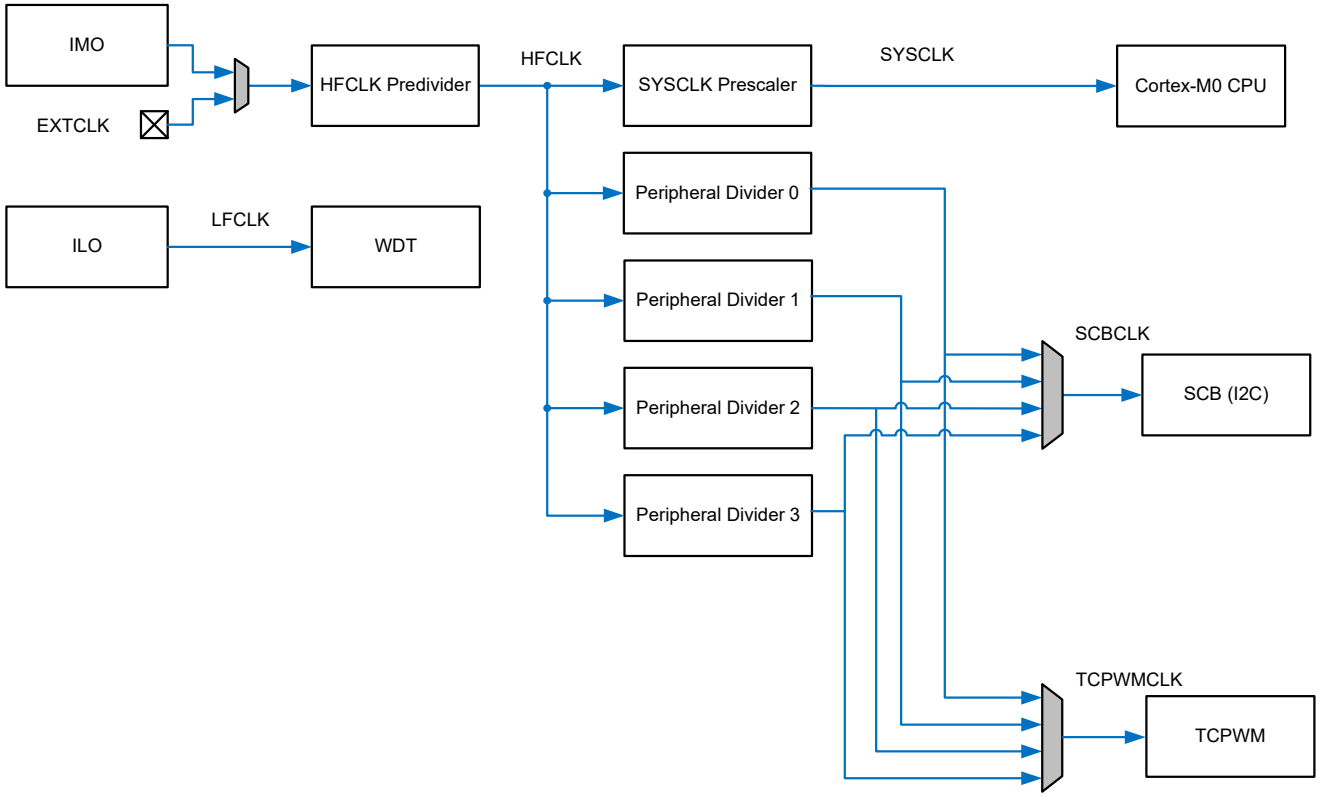
The EZ-PD™ PMG1-S2 MCU clock system includes these clock resources:

- Two internal clock sources:
  - 24–48 MHz internal main oscillator (IMO) ±2 percent across all frequencies with trim.
  - 32-kHz internal low-speed oscillator (ILO)
- External clock (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
  - Dedicated prescaler for HFCLK
- Low-frequency clock (LFCLK) sourced by ILO
- Dedicated prescaler for system clock (SYSCLK) of up to 48 MHz sourced by HFCLK
- Eight peripheral clocks, four clocks with an 8-bit divider and the remaining four clocks with 16-bit dividers

8.1 Block Diagram

Figure 8-1 gives a generic view of the clocking system in EZ-PD™ PMG1-S2 MCU devices.

Figure 8-1. Clocking System Block Diagram



The three clock sources in the device are shown in Figure 8-1, on the left. The HFCLK mux selects the HFCLK source from an external clock source or the IMO. The HFCLK prescaler divides the HFCLK input. The SYSCLK prescaler generates the SYSCLK and the peripheral dividers generate the individual peripheral clocks. The ILO sources the LFCLK.

8.2 Clock Sources

8.2.1 Internal Main Oscillator

The internal main oscillator operates with no external components and outputs a stable clock at frequencies spanning 24-48 MHz in 4-MHz increments. Frequencies are selected by setting the frequency range in the CLK_IMO_TRIM2 register and setting the IMO trim in the CLK_IMO_TRIM1 register Each device has IMO trim measured during manufacturing to meet datasheet specifications; the trim is stored in manufacturing configuration data in SFlash. These values may be loaded at startup to achieve the desired configuration. Firmware can retrieve these trim values and reconfigure the device to change the frequency at run-time.

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. The HFCLK predivider is initially set to a divide value of 4 to reduce current consumption at startup.

8.2.2 Internal Low-speed Oscillator

The internal low-speed oscillator operates with no external components and outputs a stable clock at 32-kHz nominal. The ILO is relatively low power and low accuracy. It is available in all power modes. The ILO is always used as the system low-frequency clock LFCLK in EZ-PD™ PMG1-S2 MCU. The ILO is recommended to be always on, because it is the source of the WDT, which is required for reliable system operation. The ILO can be disabled by clearing the ENABLE bit in the CLK_ILO_CONFIG register. The WDT reset must be disabled before disabling the ILO. Otherwise, any register write to disable the ILO will be ignored. Enabling the WDT reset will automatically enable the ILO.

Note Disabling the ILO reset is not recommended if:

- WDT protection is required against firmware crashes
- WDT protection is required against the power supply events that produce sudden brownout events that may in turn compromise the CPU functionality.

See the Watchdog Timer chapter on page 58 for details.

8.2.3 External Clock

The external clock is a MHz range clock that can be generated from a signal on a designated EZ-PD™ PMG1-S2 MCU pin. This clock may be used in place of the IMO as the source of the system high-frequency clock HFCLK. The allowable range of external clock frequencies is 0–48 MHz. EZ-PD™ PMG1-S2 MCU always starts up using the IMO, and the external clock must be enabled in user mode, so the device cannot be started from a reset clocked by the external clock.

8.3 Clock Distribution

EZ-PD™ PMG1-S2 MCU clocks are developed and distributed throughout the device, as shown in Figure 8-1. The distribution configuration options are as follows:

- HFCLK input selection
- HFCLK predivider configuration
- SYSCLK prescaler configuration
- Peripheral divider configuration

8.3.1 HFCLK Input Selection

HFCLK in EZ-PD™ PMG1-S2 MCU has two input options: IMO and EXTCLK. The HFCLK input is selected using the CLK_SELECT register’s DIRECT_SEL bits, as described in Table 8-1.

Table 8-1. HFCLK Input Selection Bits DIRECT_SEL

Name	Description
DIRECT_SEL[2:0]	HFCLK input clock selection 0: IMO. Uses the IMO as the source of the HFCLK 1: EXTCLK. Uses the EXTCLK as the source of the HFCLK 2–7: Reserved. Do not use

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 chapter on page 42 for more details.

8.3.2 HFCLK Predivider Configuration

The HFCLK predivider allows the device to divide the HFCLK selection mux input before use as HFCLK. The predivider is capable of dividing the HFCLK by powers of 2 between 1 and 8. The predivider value is set using register CLK_SELECT bits HFCLK_DIV, as described in Table 8-2. The HFCLK predivider is set to a divide value of 4 during boot to reduce current consumption.

Note HFCLK's frequency cannot exceed 16 MHz.

Table 8-2. HFCLK Predivider Value Bits HFCLK_DIV

Name	Description
HFCLK_DIV[1:0]	HFCLK predivider value 0: 1. no divider on HFCLK 1: 2. divides HFCLK by 2 2: 4. divides HFCLK by 4 3: 8. divides HFCLK by 8

8.3.3 SYSCLK Prescaler Configuration

The SYSCLK prescaler allows the device to divide the predivided 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 1 and 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[1:0]	SYSCLK prescaler divide value 0: 1. SYSCLK = HFCLK 1: 2. SYSCLK = HFCLK/2 2: 4. SYSCLK = HFCLK/4 3: 8. SYSCLK = HFCLK/8

8.3.4 Peripheral Clock Divider Configuration

The eight peripheral clocks are derived from the HFCLK using the 8-bit or 16-bit peripheral clock dividers. Each of the eight dividers is controlled by a PERI_DIV_8_CTL or PERI_DIV_16_CTL register, whose mapping is explained in Table 8-4 and Table 8-5.

Table 8-4. Peripheral Clock Divider Control Register PERI_DIV_8_CTLx

Bits	Name	Description
0	EN	Enables or disables the divider 0: Divider disabled 1: Divider enabled
15:8	INT8_DIV	Divide value of the divider. Output = Input/(INT8_DIV + 1)

Table 8-5. Peripheral Clock Divider Control Register PERI_DIV_16_CTLx

Bits	Name	Description
0	EN	Enables or disables the divider 0: Divider disabled 1: Divider enabled
23:8	INT16_DIV	Divide value for the divider. Output = Input/(INT16_DIV + 1) Acceptable divide values range from 0 to 65,536.

The PERI_DIV_CMD register can be used to enable, disable, and select the type of clock dividers for all peripheral clock dividers. See the PERI_DIV_CMD in the EZ-PD™ PMG1-S2 MCU Registers TRM for more details.

Input clocks to the peripherals are selected byPERI_PCLK_CTLx registers. Table 8-6 shows the peripheral clocks and their respective registers. See the EZ-PD™ PMG1-S2 MCU Registers TRM for more details.

Table 8-6. Selecting Peripheral Clocks

Clock	Register
SCB0	PERI_PCLK_CTL0
SCB1	PERI_PCLK_CTL1
TCPWM	PERI_PCLK_CTLx (x = 2–7)
USBPD (RX)	PERI_PCLK_CTL8
USBPD (TX)	PERI_PCLK_CTL9
USBPD (SAR)	PERI_PCLK_CTL10

8.4 Low-Power Mode Operation

EZ-PD™ PMG1-S2 MCU clock behavior is different in different power modes. The MHz 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

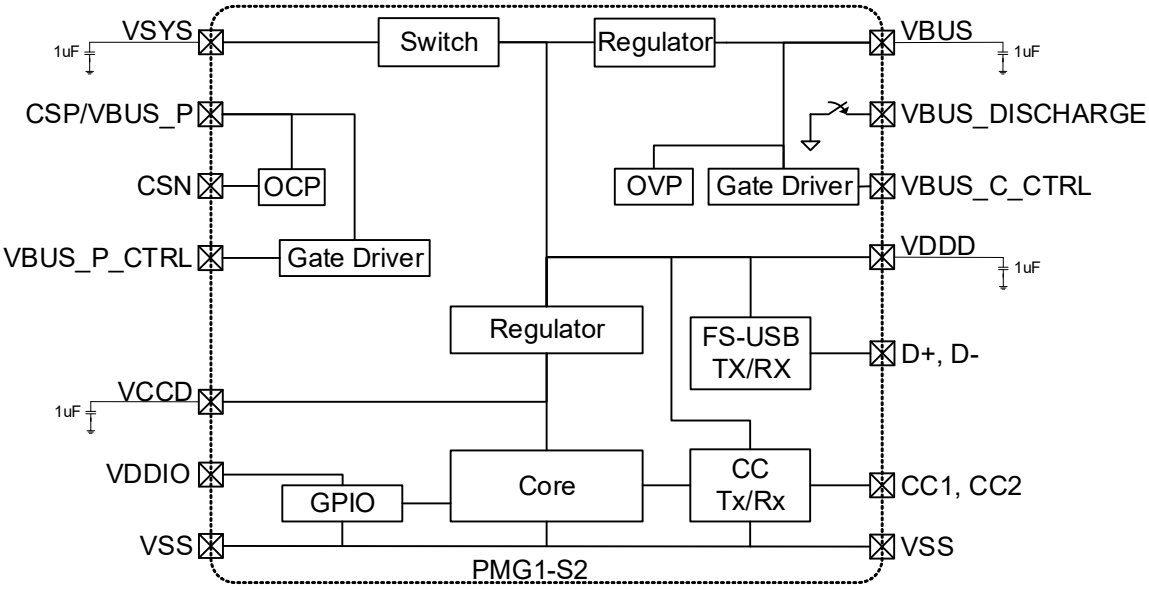
Table 8-7. Clocking System Register List

Register Name	Description
CLK_IMO_TRIM1	IMO Trim Register - This register contains IMO trim, allowing fine manipulation of its frequency.
CLK_IMO_TRIM2	IMO Frequency Selection Register - This register controls the frequency range of the IMO, allowing gross manipulation of its frequency.
CLK_ILO_CONFIG	ILO Configuration Register - This register controls the ILO configuration.
CLK_SELECT	Clock Select - This register controls clock tree configuration, selecting different sources for the system clocks.
PERI_DIV_16_CTLx	Peripheral Clock Divider Control Registers - These registers configure each of the peripheral clock dividers, enabling or disabling the divider and setting the integer divide value.
PERI_PCLK_CTLx	Programmable clock control registers - These registers are used to select the input clocks to peripherals.

9. Power Supply and Modes

9.1 Block Diagram

Figure 9-1. Power System Requirement Block Diagram



9.2 Power System Requirement Overview

Figure 9-1 shows an overview of the EZ-PD™ PMG1-S2 MCU power system requirement. EZ-PD™ PMG1-S2 MCU operates from two possible external supply sources VBUS (4.0 - 21.5 V) or VSYS (2.7 - 5.5 V). The VBUS supply is regulated inside the chip with an LDO. The chip's internal VDDD rail is intelligently switched between the output of the VBUS regulator and unregulated VSYS. The switched supply, VDDD is either used directly inside some analog blocks or further regulated down to VCCD, which powers majority of the core using the regulators inside the SRSS-Lite block. EZ-PD™ PMG1-S2 MCU has three power modes: Active, Sleep, and Deep Sleep, transitions between which are managed by the power system. A separate power domain VDDIO is provided for the GPIOs. The VDDD and VCCD pins, both the output of regulators are brought out for connecting a 1-uF capacitor for the regulator stability only. These pins are not supported as a power supplies.

9.3 Power Modes

The power modes of the device accessible and observable by the user are listed in the following table.

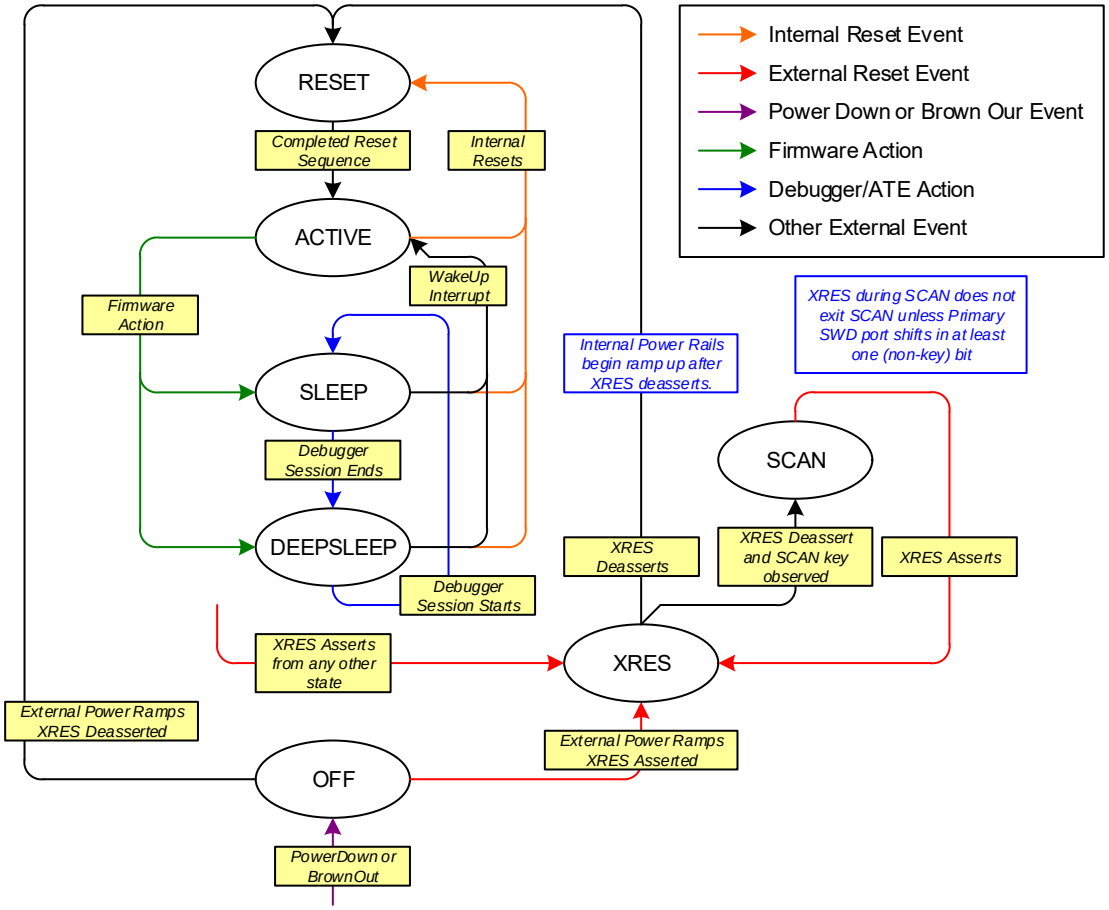
Table 9-1. EZ-PD™ PMG1-S2 MCU Power Modes

Mode	Description
RESET	Power is Valid and XRES is not asserted. An internal reset source is asserted or Sleep Controller sequences the system out of reset
ACTIVE	Power is Valid and CPU is executing instructions. This mode includes the critical Type-C power specification requirements.
SLEEP	Power is Valid and CPU is not executing instructions. All logic that is not operating is clock gated to save power.
DEEP SLEEP	Main regulator and most of the blocks in the device are shut off. Deep Sleep regulator powers logic, but only low-frequency clock is available
SCAN	System is in SCAN mode. Scan mode is entered by applying DFT key during XRES and exited by applying something other than the DFT key (at least one bit).

9.4 Mode Transitions

EZ-PD™ PMG1-S2 MCU follows the mode transitions supported for SRSS-Lite system resource block as described in SRSS-Lite on page 144. The primary mode transition diagram is shown in Figure 9-2.

Figure 9-2. EZ-PD™ PMG1-S2 MCU Primary Mode Transition Diagram



10. Chip Operational Modes

EZ-PD™ PMG1-S2 MCU 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.

EZ-PD™ PMG1-S2 MCU's operational modes are:

- Boot
- User
- Privileged
- Debug

10.1 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 the power-up phase, hardware trim settings are loaded from nonvolatile (NV) latches to guarantee proper operation during power-up. When boot concludes, the device enters user mode and code execution from flash begins.

10.2 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 IDE 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.

10.3 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 171. Exit from this mode returns the device to user mode.

10.4 Debug

Debug mode is an operational mode that allows observation of the EZ-PD™ PMG1-S2 MCU 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 Eclipse and Arm MDK 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 chapter on page 163.

For more details on protection modes, see the Device Security chapter on page 63.

11. 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 the LFCLK (32-kHz clock), generated by the ILO. The timer 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.

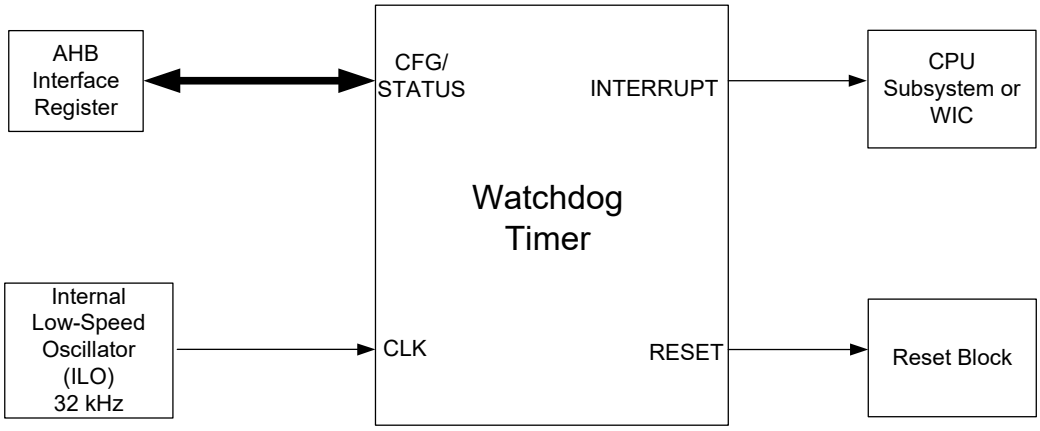
11.1 Features

The WDT has these features:

- Configurable timer period
- Can generate an interrupt in Sleep and Deep-Sleep power modes to wake up the device
- Can generate an interrupt in Active mode after a specified interval

11.2 Block Diagram

Figure 11-1. Watchdog Timer Block Diagram



11.3 How It Works

The WDT asserts a hardware reset to the device after three WDT interrupts each of programmable interval of up to 2048 ms, unless it is periodically serviced in firmware. The WDT is a 16-bit free-running wraparound up-counter.

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 and rolls back to 0. When the count value again reaches the match value, another interrupt is generated.

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 feed the watchdog timer. If the firmware does not feed the WDT for two consecutive interrupts, the third match event will generate a hardware reset.

For details, see the WDT_COUNTER, WDT_MATCH, and SRSS_INTR registers in the EZ-PD™ PMG1-S2 MCU Registers TRM.

When the WDT is used to protect against system crashes, clearing the WDT interrupt bit to feed 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:

- Feed the watchdog by clearing the interrupt bit regularly in the main body of the firmware code.
- Guarantee that the interrupt is cleared at least once every WDT period.
- Use the WDT interrupt service routine (ISR) only as a timer to trigger certain actions and to change the next WDT_-MATCH value. Do not feed WDT in this ISR.

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

1. Write the desired match value to the WDT_MATCH register.
2. Clear the WDT_MATCH bit in SRSS_INTR to clear any pending WDT interrupt.
3. Enable the WDT interrupt by setting the WDT_MATCH bit in SRSS_INTR_MASK.
4. 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.
5. 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 needs to be discarded. For example, if the IGNORE_BITS value is 3, then WDT counter becomes a 13-bit counter.

11.3.1 Enabling and Disabling WDT

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

11.3.2 WDT Interrupts and Low-Power Modes

The WDT counter sends the 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 sends 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, see the Power Supply and Modes chapter on page 55.

11.3.3 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 chapter on page 61.

11.4 Register List

Table 11-1. 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	Stores the match value of the WDT
SRSS_INTR	Feeds the WDT to avoid reset

12. Reset System

EZ-PD™ PMG1-S2 MCU 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. EZ-PD™ PMG1-S2 MCU 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 electrical signal external to EZ-PD™ PMG1-S2 MCU
- Protection fault reset (PROT_FAULT) to reset the device if unauthorized operating conditions occur

12.1 Reset Sources

The following sections provide a description of the reset sources available in EZ-PD™ PMG1-S2 MCU.

12.1.1 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.

12.1.2 Brownout Reset

Brownout reset monitors the 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.

BOD events do not set a reset cause status bit, but in some cases they can be detected. In some BOD events, V_{CCD} will fall below the minimum logic operating voltage, but remain above the minimum logic retention voltage. Thus, some BOD events may be distinguished from POR events by checking for logic retention.

12.1.3 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 setting the WDT_ENABLEx bit in the WDT_CONTROL 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 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 chapter on page 58

12.1.4 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 (CM0_AIRCR) forces a device reset when a ‘1’ is written into the SYSRESETREQ bit. CM0_AIRCR requires a value of A05F written to the top two bytes for writes. Therefore, write A05F0004 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 or BOD reset; for example, in the case of a device power cycle. All other resets leave this bit untouched.

12.1.5 External Reset

External reset (XRES) is a user-supplied reset that causes immediate system reset when asserted. The XRES_N pin is ***active low*** – a high voltage on the pin causes no behavior and a low voltage causes a reset. The pin is pulled high inside the device. XRES_N 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.

12.1.6 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 57.

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 or BOD reset; for example, in the case of a device power cycle. All other resets leave this bit untouched.

12.2 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, 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 or brownout below the logic retention voltage.

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, or XRES. These resets cannot be distinguished using on-chip resources.

12.3 Register List

Table 12-1. Reset System Register List

Register Name	Description
WDT_CONTROL	Watchdog Timer Control Register - This register allows configuration of the device watchdog timer.
CM0_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.

13. Device Security

EZ-PD™ PMG1-S2 MCU offers a number of options for protecting user designs from unauthorized access or copying. Disabling debug features and robust 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 of flash row and chip protection, see the EZ-PD™ PMG1 MCU Programming Specifications.

Note Because all programming, debug, and test interfaces are disabled when maximum device security is enabled, EZ-PD™ PMG1-S2 MCU devices with full device security enabled may not be returned for failure analysis.

13.1 Features

The EZ-PD™ PMG1-S2 MCU 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.

13.2 How It Works

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 disables all debug access to user code or memory. Access to most registers is still available; 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 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.

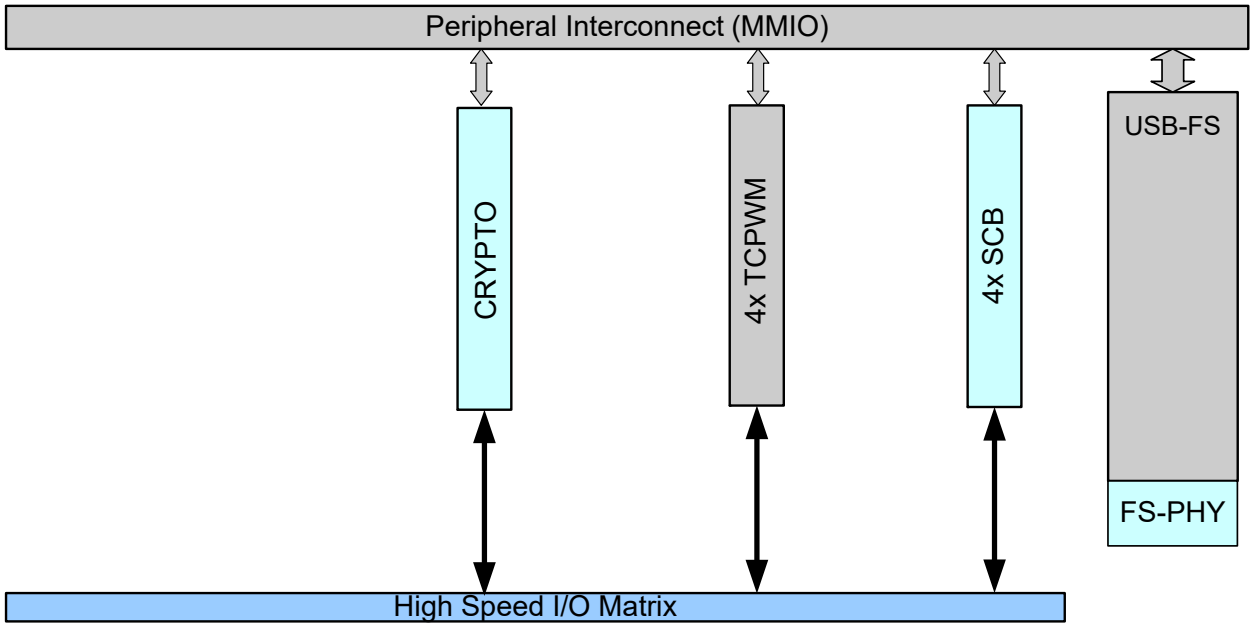
Section E: Digital System

This section encompasses the following chapters:

- Serial Communications (SCB) chapter on page 65
- Timer, Counter, and PWM chapter on page 105
- Cryptography Block chapter on page 127
- USB Full Speed (USB FS) chapter on page 146

Top Level Architecture

Digital System Block Diagram



14. Serial Communications (SCB)

The Serial Communications Block (SCB) of EZ-PD™ PMG1-S2 MCU supports three serial interface protocols: SPI, UART, and I^2C. Only one of the protocols is supported by an SCB at any given time. EZ-PD™ PMG1-S2 MCU devices have four SCBs.

14.1 Features

This block 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, and IrDA protocols
- Standard I^2C master and slave functionality
- SPI and I^2C EZ mode, which allows for operation without CPU intervention
- Low-power (Deep-Sleep) mode of operation for SPI and I^2C protocols (using external clocking)

Each of the three protocols is explained in the following sections.

14.2 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.

You can use the SPI master mode when the EZ-PD™ PMG1-S2 MCU has to communicate with one or more SPI slave devices. The SPI slave mode can be used when the EZ-PD™ PMG1-S2 MCU has to communicate with an SPI master device.

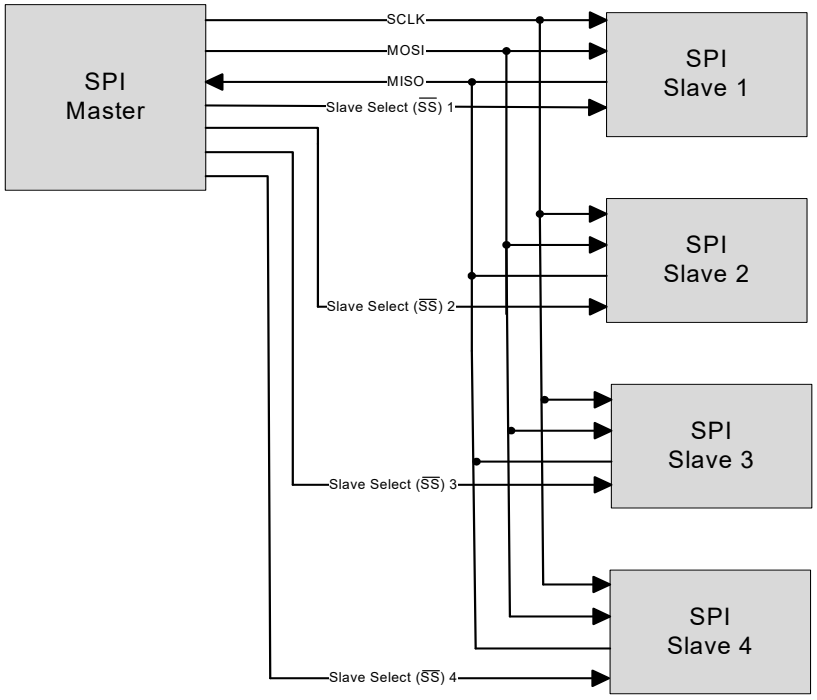
14.2.1 Features

- Supports master and slave functionality
- Supports three types of SPI protocols:
  - Motorola SPI – modes 0, 1, 2, and 3
  - TI SPI, with coinciding and preceding data frame indicator for mode 1
  - National (MicroWire) SPI for mode 0
- Data frame size programmable from 4 bits to 16 bits
- Interrupts or polling CPU interface
- Programmable oversampling
- Supports EZ mode of operation (Easy SPI (EZSPI) Protocol (applicable only for SPI slave functionality))
- Supports externally clocked slave operation:
  - In this mode, the slave operates in Active, Sleep, and Deep-Sleep system power modes
  - EZSPI mode allows for operation without CPU intervention

14.2.2 General Description

Figure 14-1 illustrates an example of SPI master with four slaves.

Figure 14-1. SPI Example



A standard SPI interface consists of four signals as follows.

- SCLK: Serial clock (clock output from the master, input to the slave).
- MOSI: Master-out-slave-in (data output from the master, input to the slave).
- MISO: Master-in-slave-out (data input to the master, output from the slave).
- Slave Select ($\overline{SS}$): Typically an active low signal (output from the master, input to the slave).

A simple SPI data transfer involves the following: the master selects a slave by driving its SS line, then it drives data on the MOSI line and a clock on the SCLK line, The slave uses the edges of SCLK 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 $\overline{SS}$ line.
- National Semiconductors SPI: A half duplex variation of the original SPI protocol.

14.2.3 SPI Modes of Operation

14.2.3.1 Motorola SPI

The original SPI protocol was defined by Motorola. It is a full duplex protocol. Multiple data transfers may happen with the SS line held at '0'. As a result, slave devices must keep track of the progress of data transfers to separate individual data frames. When not transmitting data, the SS line is held at '1' and SCLK is typically off.

Motorola SPI Modes

The Motorola SPI protocol has four different 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 14-2 illustrates driving and capturing of MOSI/MISO data as a function of CPOL and CPHA.

Figure 14-2. SPI Motorola, Four Modes

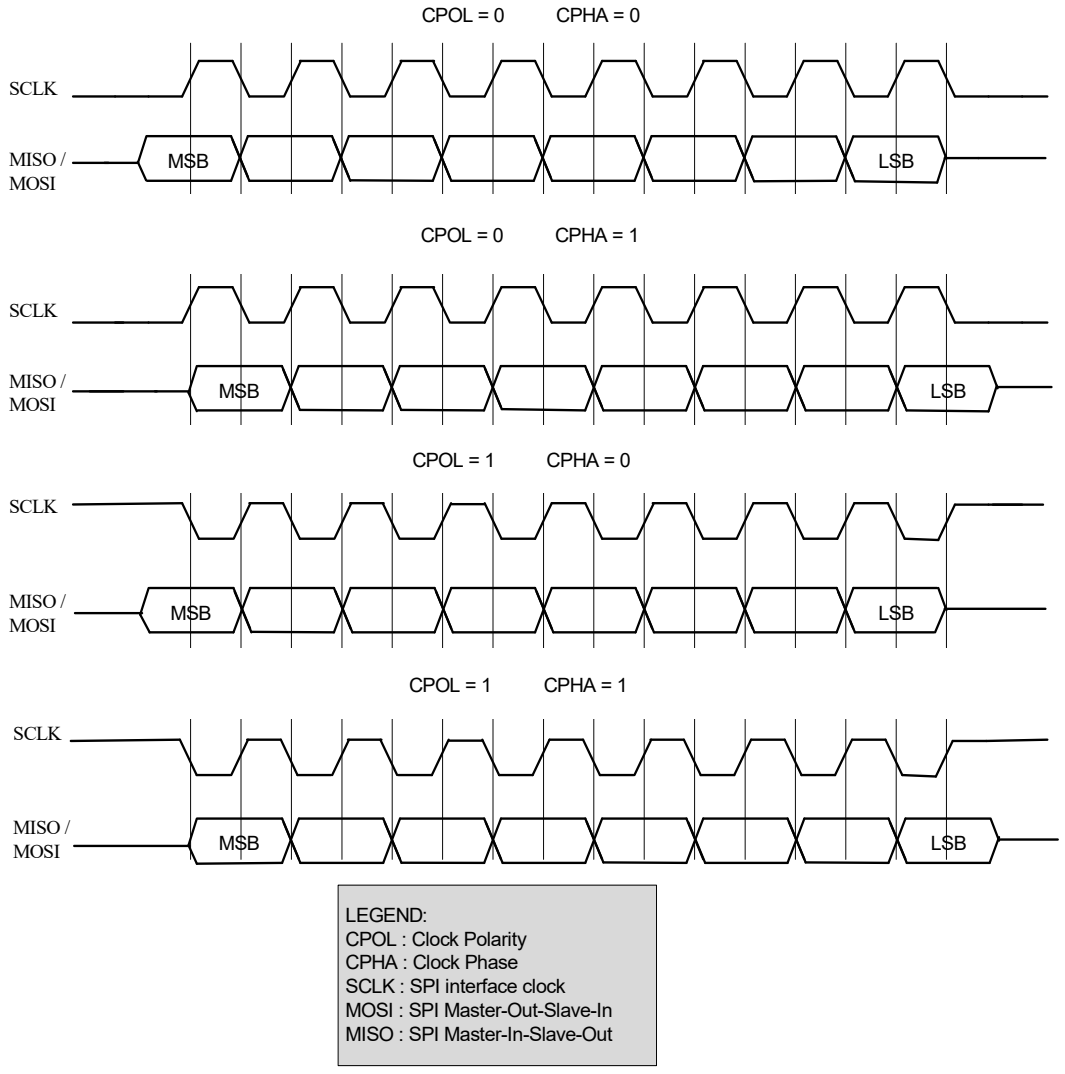
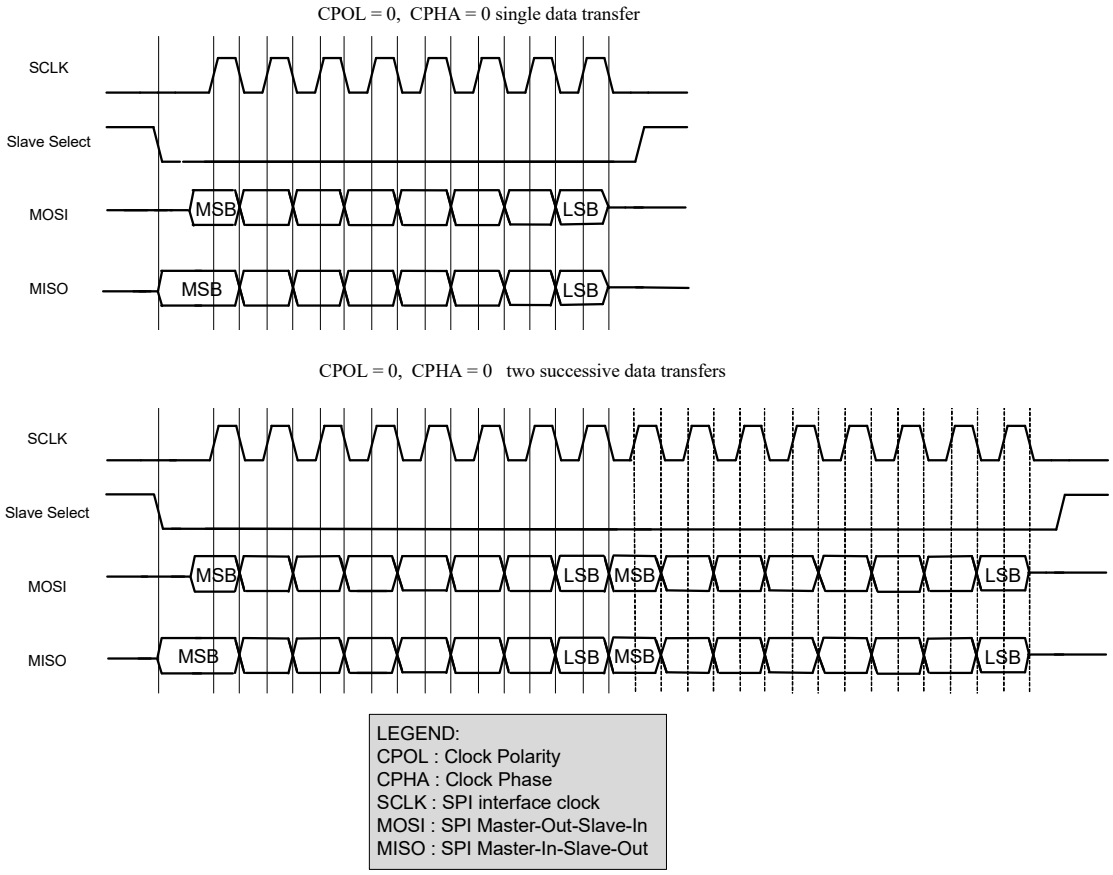


Figure 14-3 illustrates a single 8-bit data transfer and two successive 8-bit data transfers in mode 0 (CPOL is '0', CPHA is '0').

Figure 14-3. SPI Motorola Data Transfer Example



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 SCB_MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI Motorola mode by writing '00' to the SCB_MODE (bits [25:24]) of the SCB_SPI_CTRL register.
3. Select the mode of operation in Motorola by writing to the SCB_CPHA and SCB_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 75.

For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

14.2.3.2 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. As a result, slave devices need not keep track of the progress of data transfers to separate individual data frames. The start of a transfer is indicated by a high active pulse on slave select signal for a single bit transfer period. This pulse can be configured to occur one cycle before the transmission of the first data bit, or 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 14-4 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SELECT pulse precedes the first data bit. Note how the SELECT pulse of the second data transfer coincides with the last data bit of the first data transfer.

Figure 14-4. SPI TI Data Transfer Example

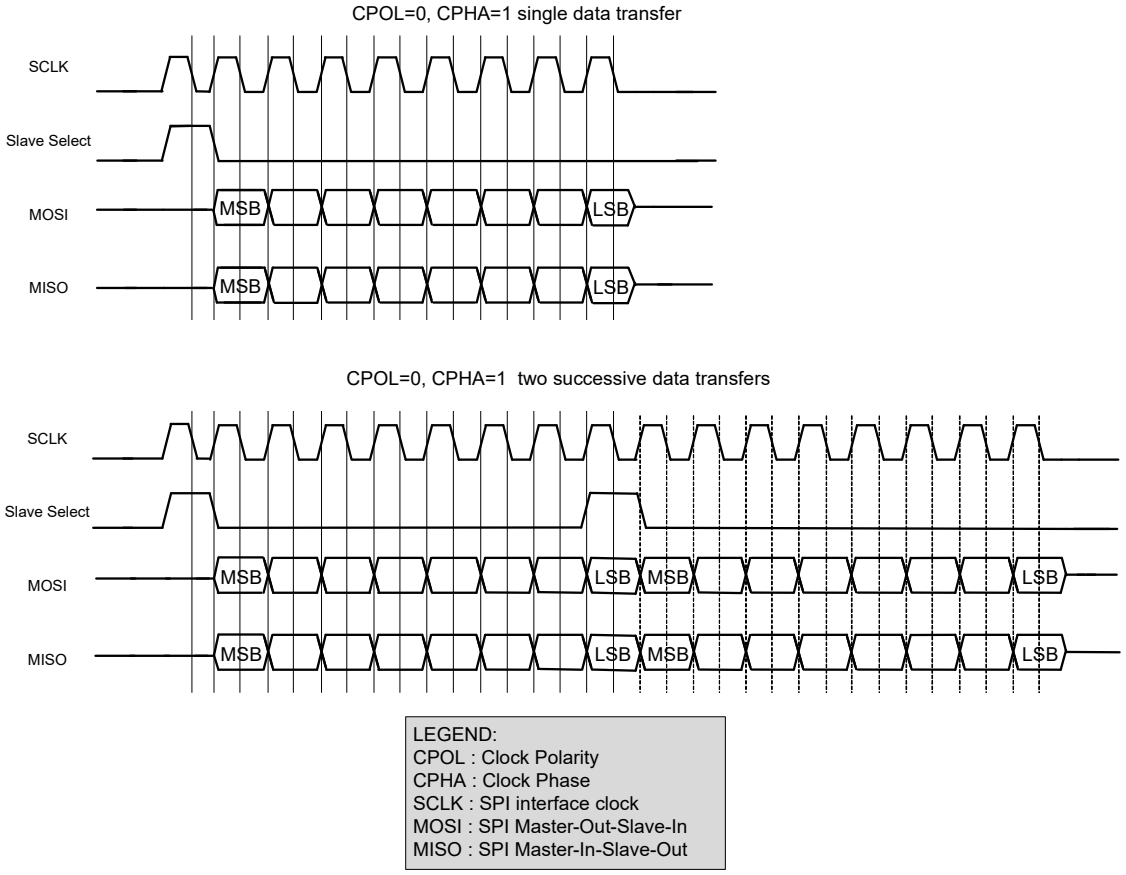
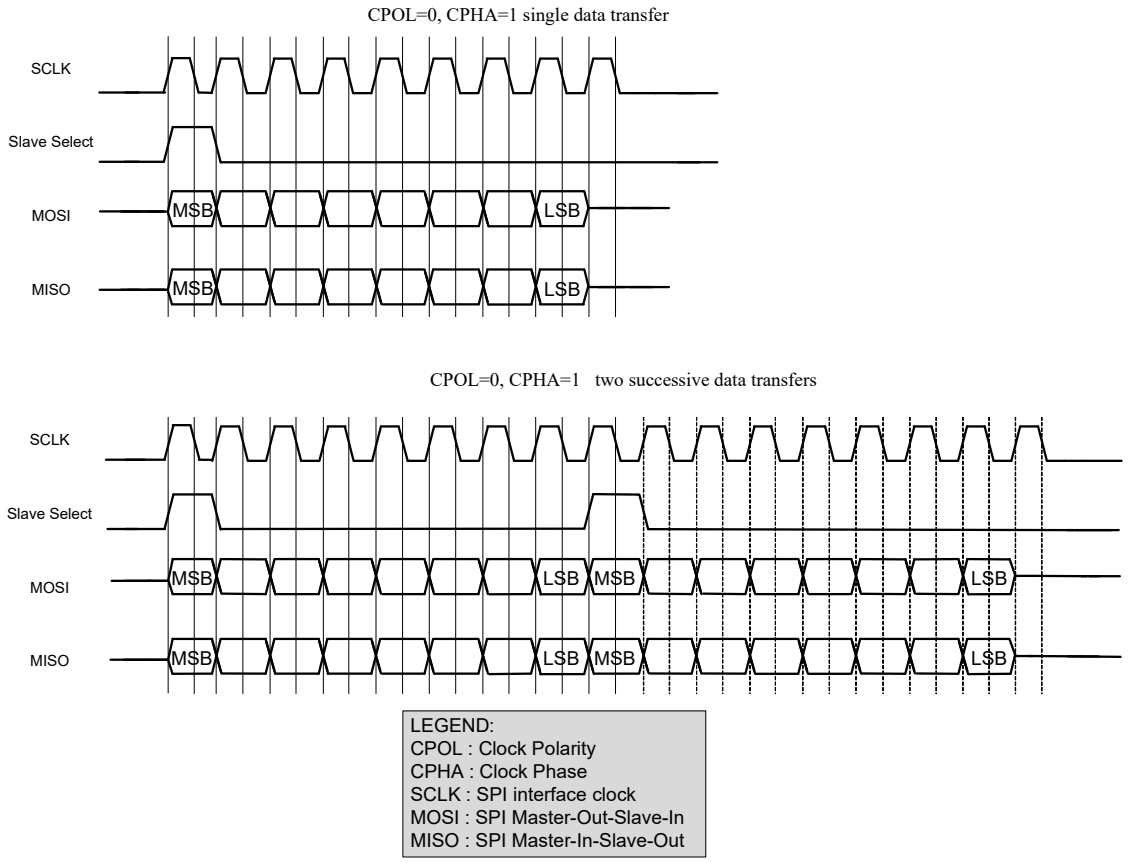


Figure 14-5 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SELECT pulse coincides with the first data bit of a frame.

Figure 14-5. SPI TI Data Transfer Example



Configuring the 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 SCB_MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI TI mode by writing '01' to the SCB_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. Follow steps 2 to 4 mentioned in Enabling and Initializing SPI on page 75.

For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

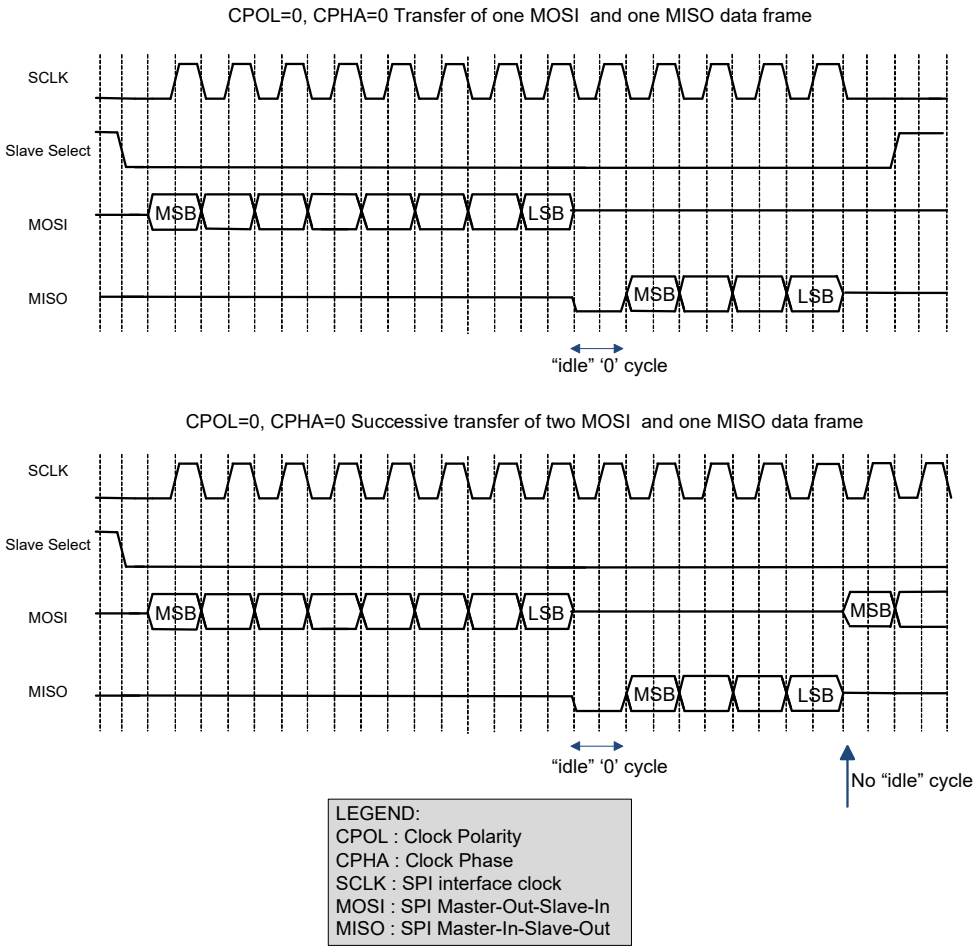
14.2.3.3 National Semiconductors SPI

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 transmission from reception. However, the successive data transfers are NOT separated by an "idle" bit transfer period.

The National Semiconductors SPI protocol only supports mode 0 (CPOL is ‘0’ and CPHA is ‘0’): data is driven on a falling edge of SCLK and data is captured on a rising edge of SCLK.

Figure 14-6 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 14-6. SPI NS Data Transfer Example



Configuring the 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 SCB_MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI NS mode by writing '10' to the SCB_MODE (bits [25:24]) of the SCB_SPI_CTRL register.
3. Follow steps 2 to 4 mentioned in Enabling and Initializing SPI on page 75.

For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

14.2.4 Easy SPI (EZSPI) Protocol

The easy SPI (EZSPI) protocol is based on the Motorola SPI operating in mode 0 (CPOL is '0' and CPHA is '0'). EZSPI mode is applicable only for slave functionality in a single slave topology. It allows communication between a master and a single slave without the need for CPU intervention at the level of individual frames.

The EZSPI protocol defines an 8-bit EZ address that indexes a memory array (32-entry array of eight bit per entry is supported) located on the slave device. To address these 32 locations, the lower five bits of the EZ address are used. All EZSPI data transfers have 8-bit data frames.

Note The SCB has a FIFO memory, which is a 16 word by 16-bit SRAM, with byte write enable. The access methods for EZ and non-EZ functions are different. In non-EZ mode, the FIFO is split into TXFIFO and RXFIFO. Each has eight entries of 16 bits per entry. The 16-bit width per entry is used to accommodate configurable data width. In EZ mode, it is used as a single 32x8 bit EZFIFO because only a fixed 8-bit width data is used in EZ mode.

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.

14.2.4.1 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 observed (0xFE) or not (0xFF). The second byte on the MOSI line is the EZ address.

14.2.4.2 Memory Array Write

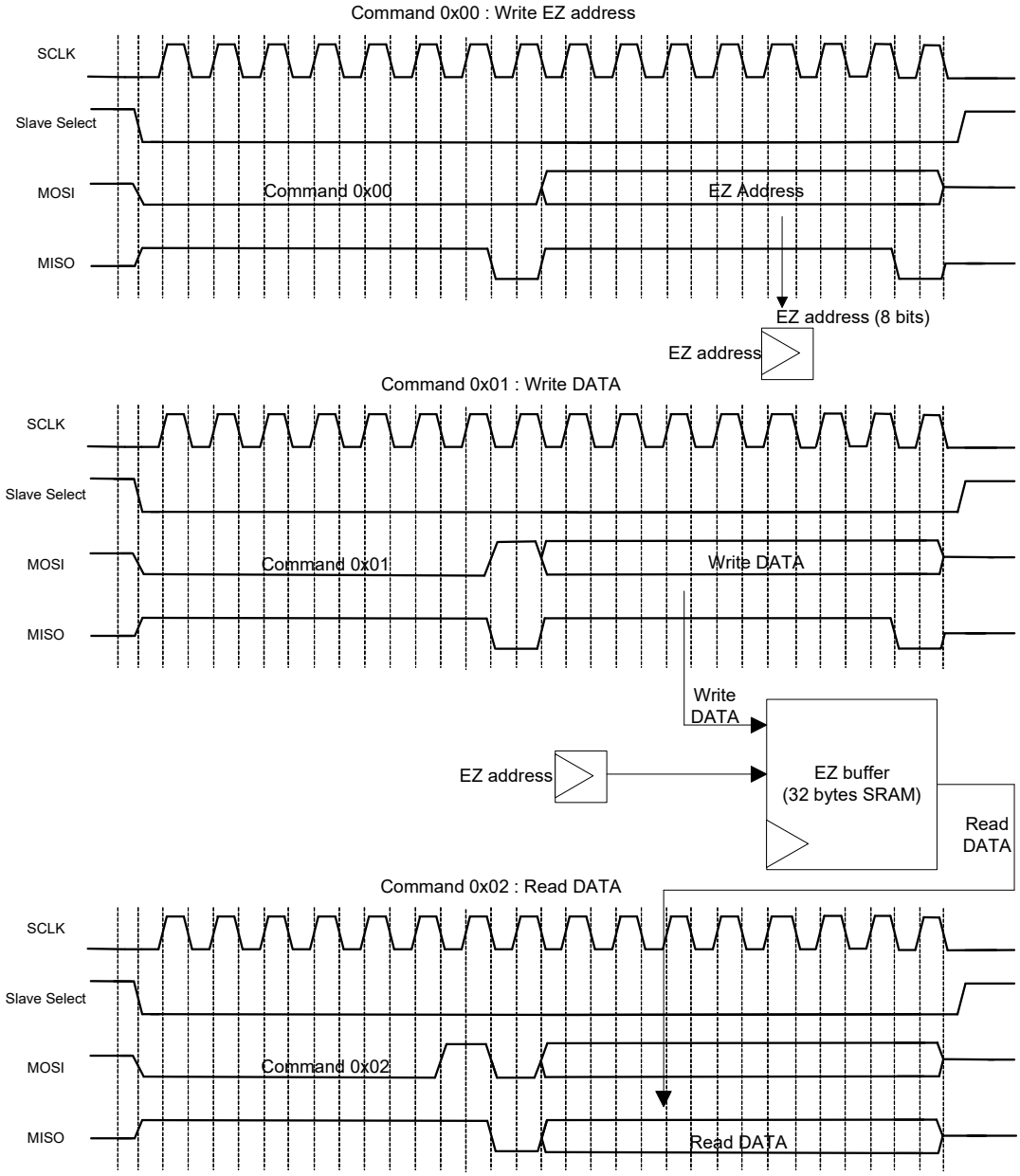
A write to a memory array index starts with a command byte (0x01) on the MOSI line indicating the master's intent to write to the memory array. The slave then drives a reply byte on the MISO line to indicate that the command was observed (0xFE) or not (0xFF). Any additional write data bytes on the MOSI 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 wraps around to 0.

14.2.4.3 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 observed (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 wraps around to 0.

Figure 14-7 illustrates the write of EZ address, write to a memory array and read from a memory array operations in the EZSPI protocol.

Figure 14-7. EZSPI Example



14.2.4.4 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 various 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. Follow steps 2 to 4 mentioned in Enabling and Initializing SPI on page 75.
3. Use continuous transmission mode for transmitter by writing '1' to the SCB_CONTINUOUS bit of SCB_SPI_CTRL register.
4. EZSPI mode is applicable only for slave functionality (write '0' to the SCB_MASTER_MODE field, bit 31 of the SCB_SPI_CTRL register).
5. Set the data frame width eight bits long (write ‘0111’ to the SCB_DATA_WIDTH field, bits [3:0] of the SCB_TX_CTRL and SCB_RX_CTRL registers).
6. Set the shift direction as MSB first (write '1' to the SCB_MSB_FIRST field, bit 8 of the SCB_TX_CTRL and SCB_RX_CTRL registers).

For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

14.2.5 SPI Registers

The SPI interface is controlled using a set of 32-bit control and status registers listed in Table 14-1. For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

Table 14-1. SPI Registers

Register Name	Operation
SCB_CTRL	Enables the SCB, selects the type of serial interface (SPI, UART, I^2C), and selects internally and externally clocked operation, 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 SELECT 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	Enables the transmitter, 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.
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 read from the receiver FIFO. Reading a data frame removes the data frame from the FIFO - behavior is similar to that of a POP operation. This register has a side effect when read by software: a data frame is removed from the FIFO.
SCB_RX_FIFO_RD_SILENT	Holds the data read from the receiver FIFO. Reading a data frame does not remove the data frame from the FIFO; behavior is similar to that of a PEEK operation.
SCB_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_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

14.2.6 SPI Interrupts

The SPI supports both internal and external interrupt requests. The internal interrupt events are listed here.

The SPI predefined interrupts can be classified as TX interrupts and RX interrupts. 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 RX interrupt output is the logical OR of the group of all possible RX interrupt sources. This signal goes high when any of the enabled RX interrupt sources are true. Various interrupt registers are used to determine the actual source of the interrupt.

The SPI supports interrupts on the following events:

- SPI transfer done
- SPI is Idle
- TX FIFO is not full
- TX FIFO is empty
- SPI Byte/Word transfer complete
- RX FIFO is empty
- RX FIFO is not empty
- Attempt to write to a full RX FIFO.
- RX FIFO is Full

14.2.7 Enabling and Initializing SPI

The SPI must be programmed in the following order:

1. Program protocol specific information using the SCB_SPI_CTRL register, according to Table 14-2. This includes selecting the submodes of the protocol and selecting master-slave functionality.
2. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers, as shown in Table 14-3:
   a. Specify the data frame width.
   b. Specify whether MSB or LSB is the first bit to be transmitted/received.
   c. Enable the transmitter and receiver.
3. Program the transmitter and receiver FIFOs using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers respectively, as shown in Table 14-4:
   a. Set the trigger level.
   b. Clear the transmitter and receiver FIFO and Shift registers.
   c. Freeze the TX and RX FIFO.
4. Program SCB_CTRL register to enable the SCB block. Also select the mode of operation. These register bits are shown in Table 14-5.

Table 14-2. SCB_SPI_CTRL Register

Bits	Name	Value	Description
[25:24]	MODE	00	SPI Motorola submode
		01	SPI Texas Instruments submode
		10	SPI National Semiconductors submode
		11	Reserved
31	MASTER_MODE	0	Master mode
		1	Slave mode

Table 14-3. SCB_TX_CTRL/SCB_RX_CTRL Registers

Bits	Name	Description
[3:0]	DATA_ WIDTH	'DATA_WIDTH + 1' is the number of bits in the transmitted or received data frame. The valid range is [3, 15]. This does not include start, stop, and parity bits.
8	MSB_FIRST	1= MSB first 0= LSB first
31	ENABLED	Transmitter enable bit for SCB_TX_CTRL and receiver enable bit for SCB_RX_CTRL registers. They must be enabled for all the protocols. Otherwise, the block may not function or the data may get lost.

Table 14-4. SCB_TX_FIFO_CTRL/SCB_RX_FIFO_CTRL Registers

Bits	Name	Description
[2:0]	TRIGGER_LEVEL	Trigger level. When the transmitter FIFO has less entries or receiver FIFO has more entries than the value of this field, a transmitter or receiver trigger event is generated in the respective case.
16	CLEAR	When '1', the transmitter or receiver FIFO and the shift registers are cleared.
17	FREEZE	When '1', hardware reads/writes to the transmitter or receiver FIFO have no effect. Freeze does not advance the TX or RX FIFO read/write pointer.

Table 14-5. SCB_CTRL Register

Bits	Name	Value	Description
[25:24]	MODE	00	I^2C mode
		01	SPI mode
		10	UART mode
		11	Reserved
31	ENABLED	0	SCB block enabled
		1	SCB block disabled

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 to internally clocked operation (explained in Internally and Externally Clocked SPI Operations on page 76). 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).

The last step of initialization should always be to enable the block (write a '1' to the ENABLED bit of the SCB_CTRL register).

14.2.8 Internally and Externally Clocked SPI Operations

The SCB supports both internally and externally clocked operations for SPI and I^2C functions. An internally clocked operation uses a clock provided by the chip. An externally clocked operation uses a clock provided by the serial interface. Externally clocked operation enables operation in the Deep-Sleep system power mode, in which a no-chip internal clock is provided to the block.

Internally clocked operation uses the high-frequency clock of the system. For more information on system clocking, see the Clocking System chapter on page 50. It also supports oversampling. Oversampling is implemented with respect to the high-frequency clock. The SCB_OVS (bits [3:0]) of the SCB_CTRL register specify the oversampling.

In SPI master mode, the valid range for oversampling is 4 to 16. Hence, the maximum bit rate is 12 Mbps. However, if you consider the I/O cell and routing delays, the effective oversampling range becomes 6 to 16. So, the maximum bit rate is 8 Mbps. **Note** LATE_MISO_SAMPLE must be set to '1' in SPIM mode.

In SPI slave mode, the oversampling field (bits [3:0]) of SCB_CTRL register is not used. However, there is a frequency requirement for the SCB clock with respect to the interface clock (SCLK). This requirement is expressed in terms of the ratio (SCB clock/SCLK). This ratio is dependent on two fields: MEDIAN of SCB_RX_CTRL register and LATE_MISO_SAMPLE of SCB_CTRL register. With the MEDIAN bit set to ‘0’ and LATE_MISO_SAMPLE bit set to ‘1’, the SCB can achieve a maximum

bit rate of 16 Mbps. However, if you consider the I/O cell and routing delays, the maximum data rate that can be achieved becomes 8 Mbps. Based on these bits, the maximum bit rates are given in Table 14-6.

Table 14-6. SPI Slave Maximum Data Rates

Median of SCB_RX_CTRL	LATE_MISO_SAMPLE of SCB_CTRL	Ratio Requirement	Maximum Bit Rate at Peripheral Clock of 48 MHz
0	0	≥12	4 Mbps
0	1	≥6	8 Mbps
1	0	≥16	3 Mbps
1	1	≥8	6 Mbps

Externally clocked operation is limited to:

- Slave functionality.
- EZ functionality. EZ functionality uses the block's SRAM as a memory structure. Non-EZ functionality uses the block's SRAM as TX and RX FIFOs; FIFO support is not available in externally clocked operation.
- Motorola mode 0 (in the case of SPI slave functionality).

Externally clocked EZ mode of operation can support a data rate of 48 Mbps (at a peripheral clock of 48 MHz).

Internally and externally clocked operation is determined by two register fields of the SCB_CTRL register:

- **EC_AM_MODE:** Indicates whether SPI slave selection is internally ('0') or externally ('1') clocked. SPI slave selection comprises the first part of the protocol.
- EC_OP_MODE: Indicates whether the rest of the protocol operation (besides SPI slave selection) is internally ('0') or externally ('1') clocked. As mentioned earlier, externally clocked operation does NOT support non-EZ functionality.

These two register fields determine the functional behavior of SPI. The register fields should be set based on the required behavior in Active, Sleep, and Deep-Sleep system power mode. Improper setting may result in faulty behavior in certain system power modes. Table 14-7 and Table 14-8 describe the settings for SPI (in EZ and non-EZ mode).

14.2.8.1 Non-EZ Mode of Operation

In non-EZ mode there are two possible settings. As externally clocked operation is not supported for non-EZ functionality (no FIFO support), EC_OP_MODE should always be set to '0'. However, EC_AM_MODE can be set to '0' or '1'. Table 14-7 gives an overview of the possibilities. The combination EC_AM_MODE=0 and EC_OP_MODE=1 is invalid and the block will not respond.

Table 14-7. SPI Non-EZ Mode

SPI, Standard (non-EZ) Mode				
	EC_OP_MODE = 0		EC_OP_MODE = 1	
System Power Mode	EC_AM_MODE = 0	EC_AM_MODE = 1	EC_AM_MODE = 1	EC_AM_MODE=0
Active and Sleep	Selection using internal clock. Operation using internal clock.	Selection using external clock: Wakeup interrupt cause is disabled in Active mode (MASK = 0) and in the Sleep mode, the MASK bit can be configured by the user. After that, selection using internal clock. Operation using internal clock.	Not supported	Invalid configuration
Deep-Sleep	Not supported	Selection using external clock: Wakeup interrupt cause in enabled (MASK = 1). Generate 0xff bytes.	Not supported	

EC_OP_MODE is '0' and EC_AM_MODE is '0': This setting only works in Active and Sleep system power modes. The entire block's functionality is provided in the internally clocked domain.

EC_OP_MODE is '0' and EC_AM_MODE is '1': This setting works in Active and Sleep system power mode and provides limited (wake up) functionality in Deep-Sleep system power mode. SPI slave selection is performed by both the internally and externally clocked logic: in Active system power mode both are active and in Deep-Sleep system power mode only the externally clocked logic is active. When the externally clocked logic detects slave selection, it sets a wakeup interrupt cause bit, which can be used to generate an interrupt to wake up the CPU.

- In Active system power mode, the CPU and the block's internally clocked slave selection logic are active and the wakeup interrupt cause is disabled (associated MASK bit is '0'). But in the Sleep mode, wakeup interrupt cause can be either enabled or disabled (MASK bit can be either '1' or '0') based on the application. The remaining operations in the Sleep mode are same as that of the Active mode. The internally clocked logic takes care of the ongoing SPI transfer.
- In Deep-Sleep system power mode, the CPU needs to be woken up and the wakeup interrupt cause is enabled (MASK bit is '1'). Waking up takes time, so the ongoing SPI transfer is negatively acknowledged ('1' bits or "0xFF" bytes are send out on the MISO line) and the internally clocked logic takes care of the next SPI transfer when it is woken up.

14.2.8.2 EZ Mode of Operation

EZ mode has three possible settings. EC_AM_MODE can be set to '0' or '1' when EC_OP_MODE is '0' and EC_AM_MODE must be set to '1' when EC_OP_MODE is '1'. Table 14-8 gives an overview of the possibilities. The grey cells indicate a possible, yet not recommended, setting because it involves a switch from the externally clocked logic (slave selection) to the internally clocked logic (rest of the operation). The combination EC_AM_MODE=0 and EC_OP_MODE=1 is invalid and the block will not respond.

Table 14-8. SPI EZ Mode

SPI, EZ Mode				
	EC_OP_MODE = 0		EC_OP_MODE = 1	
System Power Mode	**EC_AM_MODE = 0**	**EC_AM_MODE = 1**	**EC_AM_MODE = 1**	**EC_AM_MODE=0**
Active and Sleep	Selection using internal clock. Operation using internal clock.	Selection using external clock: Wakeup interrupt cause is disabled in Active mode (MASK = 0) and in Sleep mode, the mask bit can be configured by the user. After that, selection using internal clock. Operation using internal clock.	Selection using external clock. Operation using external clock.	Invalid configuration
Deep-Sleep	Not supported	Selection using external clock: Wakeup interrupt cause is enabled (MASK = 1). Generate 0xff bytes.	Selection using external clock. Operation using external clock.	

EC_OP_MODE is '0' and EC_AM_MODE is '0': This setting only works in Active system power mode. The entire block's functionality is provided in the internally clocked domain.

EC_OP_MODE is '0' and EC_AM_MODE is '1': This setting works in Active system power mode and provides limited (wake up) functionality in Deep-Sleep system power mode. SPI slave selection is performed by both the internally and externally clocked logic: in Active system power mode both are active and in Deep-Sleep system power mode only the externally clocked logic is active. When the externally clocked logic detects slave selection, it sets a wakeup interrupt cause bit, which can be used to generate an interrupt to wake up the CPU.

- In Active system power mode, the CPU and the block's internally clocked slave selection logic are active and the wakeup interrupt cause is disabled (associated MASK bit is '0'). But in Sleep mode, wakeup interrupt cause can be either enabled or disabled (MASK bit can be either '1' or '0') based on the application. The remaining operations in the Sleep mode are same as that of the Active mode. The internally clocked logic takes care of the ongoing SPI transfer.

- In Deep-Sleep system power mode, the CPU needs to be woken up and the wakeup interrupt cause is enabled (MASK bit is '1'). Waking up takes time, so the ongoing SPI transfer is negatively acknowledged ('1' bits or "0xFF" bytes are send out on the MISO line) and the internally clocked logic takes care of the next SPI transfer when it is woken up.

EC_OP_MODE is '1' and EC_AM_MODE is '1': This setting works in Active system power mode and Deep-Sleep system power mode. The SCB functionality is provided in the externally clocked domain. Note that this setting results in externally clocked accesses to the block's SRAM. These accesses may conflict with internally clocked accesses from the device. This may cause wait states or bus errors. The field FIFO_BLOCK of the SCB_CTRL register determines whether wait states ('1') or bus errors ('0') are generated.

14.3 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

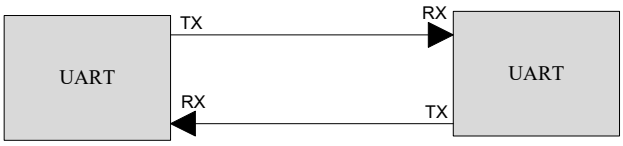
14.3.1 Features

- Asynchronous transmitter and receiver functionality
- Supports a maximum data rate of 1 Mbps
- Supports UART protocol
  - Standard UART
  - SmartCard (ISO7816) reader.
  - IrDA
- Supports Local Interconnect Network (LIN)
  - Break detection
  - Baud rate detection
  - Collision detection (ability to detect that a driven bit value is not reflected on the bus, indicating that another component is driving the same bus).
- Multi-processor mode
- Data frame size programmable from 4 bits to 16 bits.
- Programmable number of STOP bits, which can be set to 1, 1.5, or 2 data bits
- Parity support (odd and even parity)
- Interrupt or polling CPU interface
- Programmable oversampling

14.3.2 General Description

Figure 14-8 illustrates a standard UART TX and RX.

Figure 14-8. 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 be the receiver to detect single bit errors. As the interface does not have a clock (asynchronous), the transmitter and receiver use their own clocks; also, they need to agree upon the period of a bit transfer.

Three different serial interface protocols are supported:

- Standard UART protocol
  - Multi-Processor Mode
  - Local Interconnect Network (LIN)
- SmartCard, similar to UART, but with a possibility to send a negative acknowledgement
- IrDA, modification to the UART with a modulation scheme

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 3. 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. Figure 14-9 depicts the default configuration of the UART interface of the SCB.

Note UART interface does not support external clocking operation. Hence, UART operates only in the Active and Sleep system power modes.

14.3.3 UART Modes of Operation

14.3.3.1 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 bits 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.

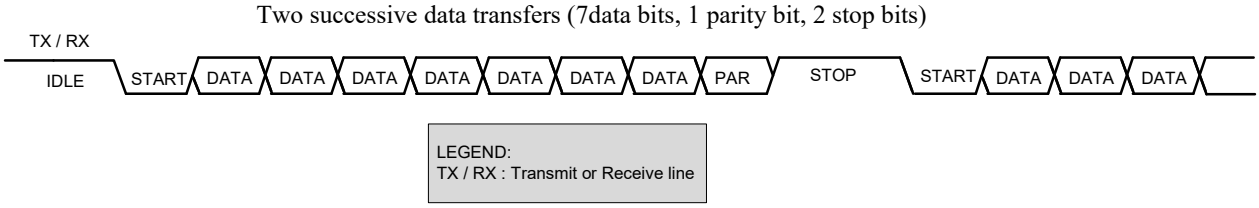
Because the interface does not have a clock, the transmitter and receiver need to agree upon the period of a bit transfer. The transmitter and receiver have their own internal clocks. The receiver clock runs at a higher frequency than the bit transfer frequency, such that the receiver may oversample 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.

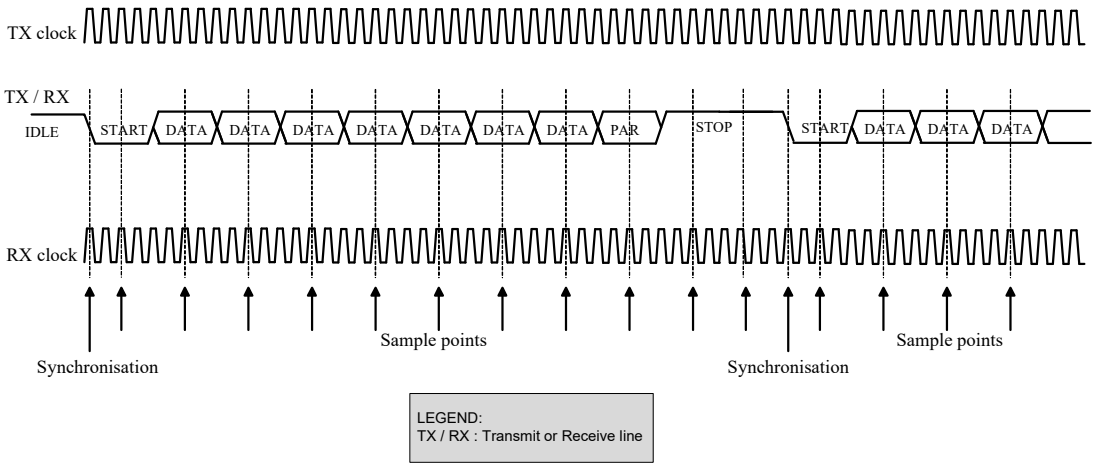
Figure 14-9 illustrates the UART protocol.

Figure 14-9. UART, Standard Protocol Example



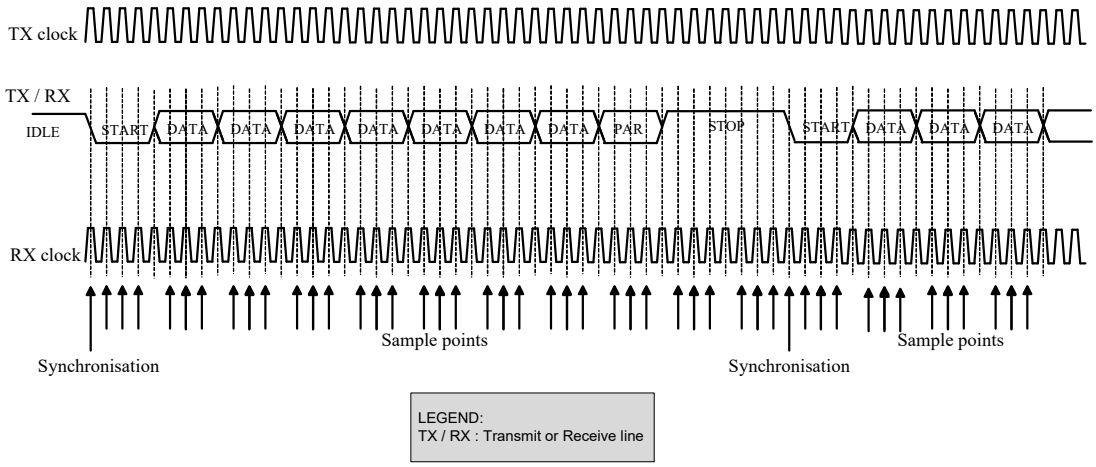
The receiver oversamples the incoming signal; the value of the sample point in the middle of the bit transfer period (on the receiver's clock) is used. Figure 14-10 illustrates this.

Figure 14-10. 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. Figure 14-11 illustrates this.

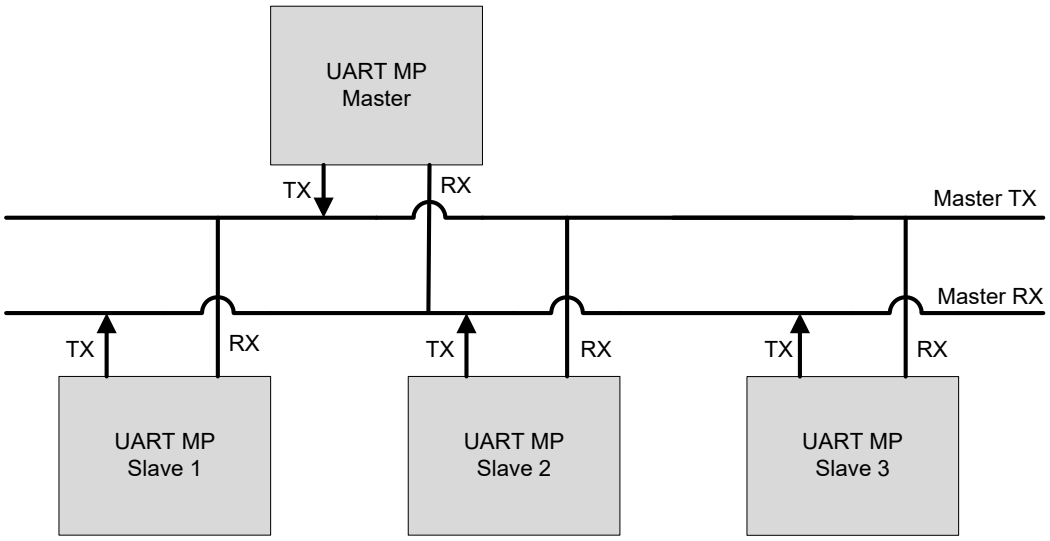
Figure 14-11. UART, Standard Protocol (Multiple Samples)



UART Multi-Processor Mode

The UART_MP (multi-processor) mode is defined with "single-master-multi-slave" topology, as shown in Figure 14-12. 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.

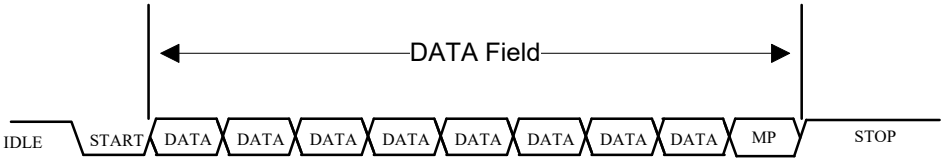
Figure 14-12. UART MP Mode Bus Connections



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 14-13
- Parity bit is disabled

Figure 14-13. UART MP Data Frame

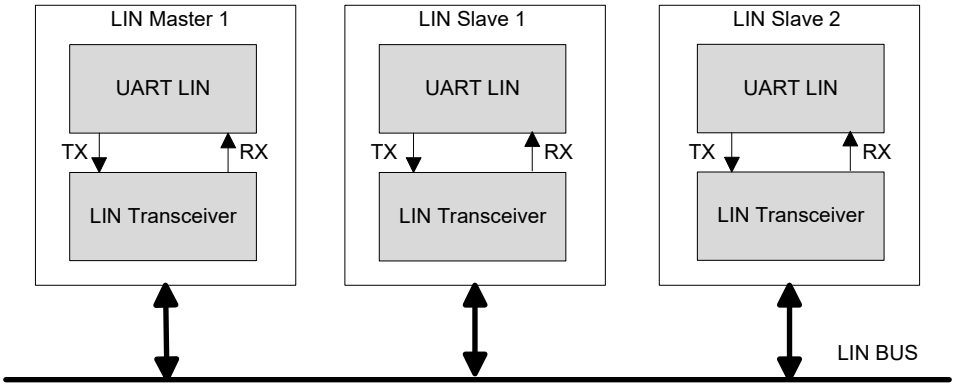


The SCB can be used as either master or slave device in UART_MP mode. Both SCB_TX_CTRL and SCB_RX_CTRL registers should be set to 9-bit data frame size. When the SCB works as UART_MP master device, the firmware changes the MP flag for every address or data frame. When it works as UART_MP slave device, 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 SCB_ADDRESS_ACCEPT field of the SCB_CTRL register is set to '1'. If received address does not match its own address, then the interface ignores the following data, until next address is received for compare.

UART LIN (Local Interconnect Network) 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. Figure 14-14 illustrates the UART_LIN and LIN Transceiver.

Figure 14-14. UART_LIN and LIN Transceiver

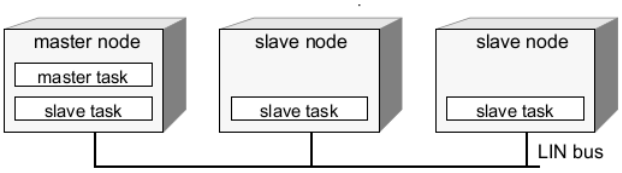


LIN protocol defines two tasks:

- 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 14-15.

Figure 14-15. LIN Bus Nodes and Tasks



LIN is based on the transmission of frames at pre-determined moments of time. A frame is divided into header and response fields.

- 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.

The UART LIN of SCB supports slave task, receiving the header and transmitting the response. It provides baud rate detection (using sync field - 0x55) operation. Apart from the break field, a frame transmission (both header and response) consist of one or multiple byte frame transmissions, with each byte transmission consisting of a start bit, 8 data bits and 1 or more stop bits (on both the UART TX and RX lines).

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).

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 '10' to the SCB_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 SCB_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 SCB_MP_MODE (bit 11) or SCB_LIN_MODE (bit 12) respectively of the SCB_UART_RX_CTRL register.

4. Follow steps 2 to 4 described in Enabling and Initializing UART on page 86.

For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

14.3.3.2 SmartCard (ISO7816)

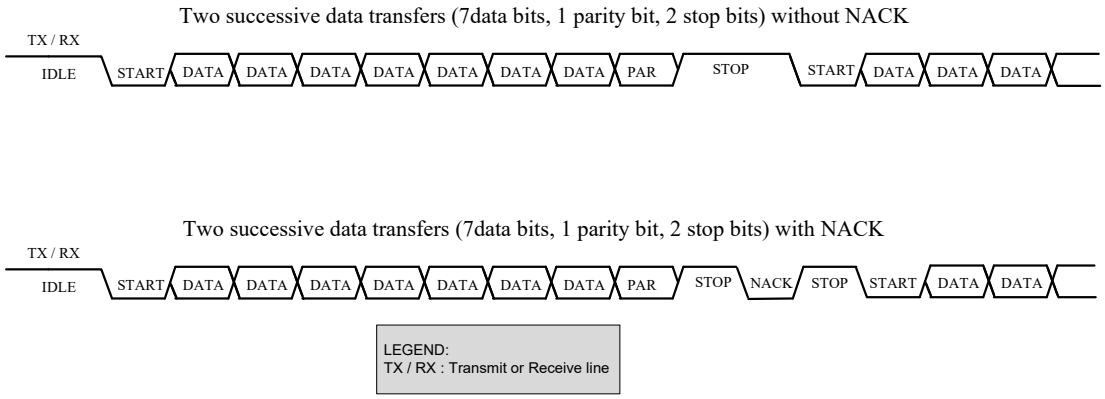
ISO7816 is asynchronous serial interface, defined with single-master-single slave topology. ISO7816 defines both Reader (master) and Card (slave) functionality. For more information, refer to the ISO7816 Specification. Only master (reader) function is supported by the SCB. This block provides the basic physical layer support with asynchronous character transmission. UART_TX line is connected to SmartCard IO line, by internally multiplexing between UART_TX and UART_RX control modules.

The SmartCard transfer is similar to a UART transfer, with the addition of a negative acknowledgement (NACK) that may be sent from the receiver to the transmitter. A NACK is always '0'. Both master and slave may drive the same line, although never at the same time.

A SmartCard transfer has the transmitter drive the start bit and data bits (and optionally a parity bit). After these bits, it enters its stop period by releasing the bus. Releasing results in the line being '1' (the value of a stop bit). After one bit transfer period into the stop period, the receiver may drive a NACK on the line (a value of '0') for one bit transfer period. This NACK is observed by the transmitter, which reacts by extending its stop period by one bit transfer period. For this protocol to work, the stop period should be longer than one bit transfer period. Note that a data transfer with a NACK takes one bit transfer period longer, than a data transfer without a NACK. Typically, implementations use a tristate driver with a pull-up resistor, such that when the line is not transmitting data or transmitting the Stop bit, its value is '1'.

Figure 14-16 illustrates the SmartCard protocol.

Figure 14-16. SmartCard Example



The communication Baud rate for ISO7816 is given as:

Baud rate= f7816 × (D/F)

Where f7816 is the clock frequency, F is the clock rate conversion integer, and D is the baud rate adjustment integer.

By default, F = 372, D = f1, and the maximum clock frequency is 5 MHz. Thus, maximum baud rate is 13.4 Kbps. Typically, a 3.57-MHz clock is selected. The typical value of the baud rate is 9.6 Kbps.

Configuring SCB as UART SmartCard Interface

To configure the SCB as a UART SmartCard interface, set various register bits in the following order. For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

1. Configure the SCB as UART interface by writing '10' to the SCB_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 SCB_MODE (bits [25:24]) of the SCB_UART_CTRL register.
3. Follow steps 2 to 4 described in Enabling and Initializing UART on page 86.

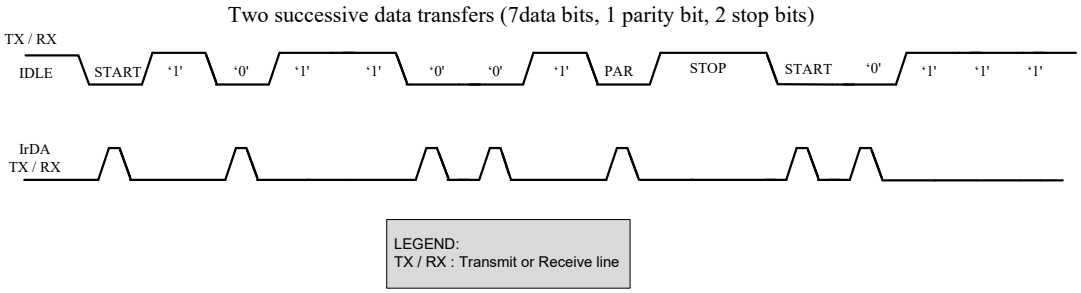
14.3.3.3 IrDA

The SCB supports the Infrared Data Association (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.

Different communication speeds under 115.2 Kbps can be achieved by configuring corresponding block clock frequency. Additional allowable rates are 2.4 Kbps, 9.6 Kbps, 19.2 Kbps, 38.4 Kbps, and 57.6 Kbps. An IrDA serial infrared interface operates at 9.6 Kbps. Figure 14-17 shows how a UART transfer is IrDA modulated.

Figure 14-17. IrDA Example



Configuring the SCB as UART IrDA Interface

To configure the SCB as a UART IrDA interface, set various register bits in the following order. For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

1. Configure the SCB as UART interface by writing '10' to the SCB_MODE (bits [25:24]) of the SCB_CTRL register.
2. Configure the UART interface to operate as IrDA protocol by writing '10' to the SCB_MODE (bits [25:24]) of the SCB_UART_CTRL register.
3. Configure the SCB as described in Enabling and Initializing UART on page 86.

14.3.4 UART Registers

The UART interface is controlled using a set of 32-bit registers listed in Table 14-9. For more information on these registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

Table 14-9. UART Registers

Register Name	Operation
SCB_UART_CTRL	Used to select the sub-modes of UART (standard UART, SmartCard, IrDA), also used for local loop back control.
SCB_UART_STATUS	Used to specify the BR_COUNTER value that determines the bit period.
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 enable the transmitter, also 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.

14.3.5 UART Interrupts

The UART supports both internal and external interrupt requests. The internal interrupt events are listed in this section.

The UART predefined interrupts can be classified as TX interrupts and RX interrupts. 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 RX interrupt output is the logical OR of the group of all possible RX interrupt sources. This signal goes high when any of the enabled RX interrupt sources are true. The UART provides interrupts on the following events:

- UART transmission done.
- UART TX received a NACK in SmartCard mode.
- UART arbitration lost (in LIN or SmartCard modes).
- Frame error in received data frame.
- Parity error in received data frame.
- LIN baud rate detection is completed.
- LIN break detection is successful.

14.3.6 Enabling and Initializing UART

The UART must be programmed in the following order:

1. Program protocol specific information using the SCB_UART_CTRL register, according to Table 14-10. 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, as shown in Table 14-11.
   a. Specify the data frame width.
   b. Specify whether MSB or LSB is the first bit to be transmitted or received.
   c. Enable the transmitter and receiver.
3. Program the transmitter and receiver FIFOs using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers respectively, as shown in Table 14-12.
   a. Set the trigger level.
   b. Clear the transmitter and receiver FIFO and Shift registers.
   c. Freeze the TX and RX FIFOs.
4. Program the SCB_CTRL register to enable the SCB block. Also select the mode of operation, as shown in Table 14-13.

Table 14-10. SCB_UART_CTRL Register

Bits	Name	Value	Description
[25:24]	MODE	00	Standard UART
		01	SmartCard
		10	IrDA
		11	Reserved
16	LOOP_BACK	Loop back control. This allows a SCB UART transmitter to communicate with its receiver counterpart.	

Table 14-11. SCB_TX_CTRL/SCB_RX_CTRL Registers

Bits	Name	Description
[3:0]	DATA_ WIDTH	'DATA_WIDTH + 1' is the no. of bits in the transmitted or received data frame. The valid range is [3, 15]. This does not include start, stop, and parity bits.
8	MSB_FIRST	1= MSB first 0= LSB first
31	ENABLED	Transmitter enable bit for SCB_TX_CTRL and receiver enable bit for SCB_RX_CTRL registers. They must be enabled for all the protocols. Otherwise, the block may not function or the data may get lost.

Table 14-12. SCB_TX_FIFO_CTRL/SCB_RX_FIFO_CTRL Registers

Bits	Name	Description
[2:0]	TRIGGER_LEVEL	Trigger level. When the transmitter FIFO has less entries or receiver FIFO has more entries than the value of this field, a transmitter or receiver trigger event is generated in the respective case.
16	CLEAR	When '1', the transmitter or receiver FIFO and the shift registers are cleared/invalidated.
17	FREEZE	When '1', hardware reads/writes to the transmitter or receiver FIFO have no effect. Freeze will not advance the TX or RX FIFO read/write pointer.

Table 14-13. SCB_CTRL Register

Bits	Name	Value	Description
[25:24]	MODE	00	I^2C mode
		01	SPI mode
		10	UART mode
		11	Reserved
31	ENABLED	0	SCB block enabled
		1	SCB block disabled

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).

The last step of initialization should always be to enable the block (write a '1' to the ENABLED bit of the SCB_CTRL register).

14.4 Inter Integrated Circuit (I^2C)

EZ-PD™ PMG1-S2 MCU contains a Serial Communication Block (SCB) configured to operate as a I^2C block. This section explains the I^2C implementation in EZ-PD™ PMG1-S2 MCU. For more information on the I^2C protocol specification, refer to the I^2C-bus specification available on the NXP website.

14.4.1 Features

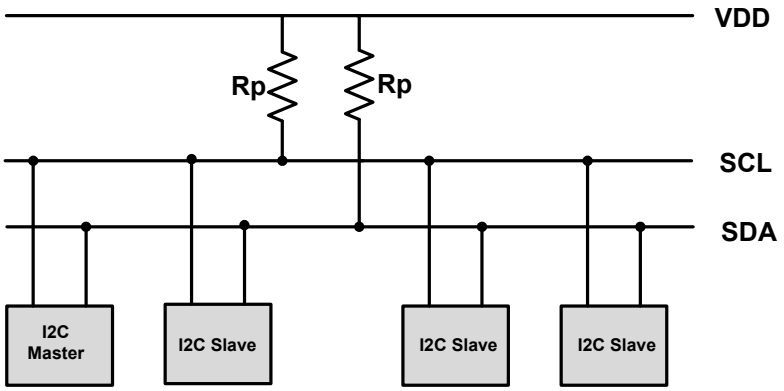
EZ-PD™ PMG1-S2 MCU supports the following I^2C features:

- Master, slave, and master/slave mode
- Slow-mode (50 kbps), standard-mode (100 kbps), fast-mode (400 kbps), and fast-mode plus (1000 kbps) data-rates
- 7- or 10-bit slave addressing (10-bit addressing requires firmware support)
- Clock stretching and collision detection
- Programmable oversampling of I^2C clock signal (SCL)
- Error reduction using an digital median filter on the input path of the I^2C data signal (SDA)
- Glitch-free signal transmission with an analog glitch filter
- Interrupt or polling CPU interface
- Supports EZ mode of operation (Easy I^2C (EZI2C) Protocol - applicable only for I^2C slave functionality)

14.4.2 General Description

Figure 14-18 illustrates an example of an I^2C communication network.

Figure 14-18. I^2C Interface Block Diagram



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

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

I^2C devices are connected to these lines using open collector or open-drain output stages, with pull-up resistors (R_P). 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.

14.4.3 Terms and Definitions

Table 14-14 explains the commonly used terms in an I²C communication network.

Table 14-14. Definition of I²C 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 without corrupting the message
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

Bus Stalling (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 postponing the generation of a positive edge on the SCL line, effectively synchronizing with the slave device that is stretching the clock.

Bus Arbitration

The I²C 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.

14.4.4 I²C Modes of Operation

I²C 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 14-15 illustrates the I²C modes of operation.

Table 14-15. I²C Modes

Mode	Description
Slave	Slave only operation (default)
Master	Master only operation
Multi-master	Supports more than one master on the bus
Multi-master-slave	Simultaneous slave and multi-master operation

Data transfer through the I²C bus follows a specific format. Table 14-16 lists some common bus events that are part of an I²C data transfer. The Write Transfer and Read Transfer sections explain the format of bits on an I²C bus during data transfer.

Table 14-16. I²C Bus Events Terminology

Bus Event	Description
START	A HIGH to LOW transition on the SDA line while SCL is HIGH
STOP	A LOW to HIGH transition on the SDA line while SCL is HIGH
ACK	The receiver pulls the SDA line LOW and it remains LOW during the HIGH period of the clock pulse after the transmitter transmits each byte.
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.
Repeated START	START condition generated by master at the end of a transfer instead of a STOP condition

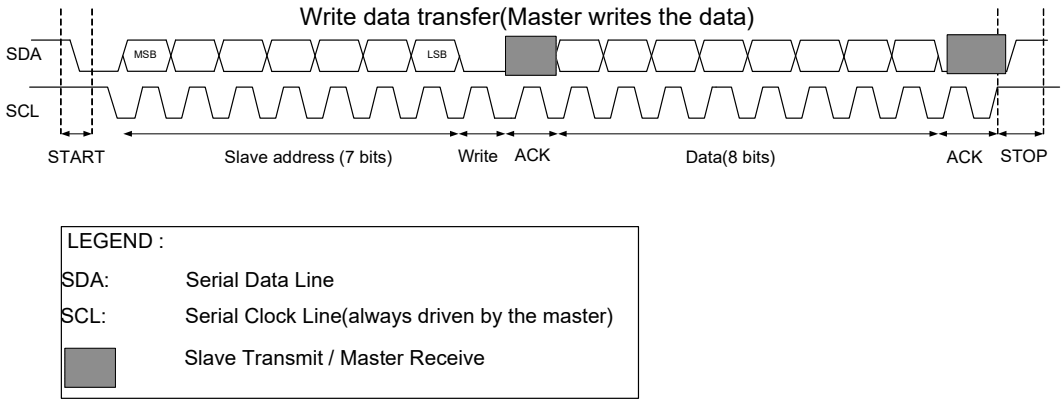
When operating in multi-master mode, the bus should always be checked to see if it is busy; another master may already be communicating with a slave. In this case, the master must wait until the current operation is complete before issuing a START signal (see Table 14-16, Figure 14-19 and Figure 14-20). The master looks for a STOP signal as an indicator that it can start its data transmission.

When operating in multi-master-slave mode, if the master loses arbitration during data transmission, the hardware reverts to slave mode and the received byte generates a slave address interrupt.

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. Write and read transfer examples are available in Master Mode Transfer Examples on page 97, Slave Mode Transfer Examples on page 99, and Multi-Master Mode Transfer Example on page 103.

14.4.4.1 Write Transfer

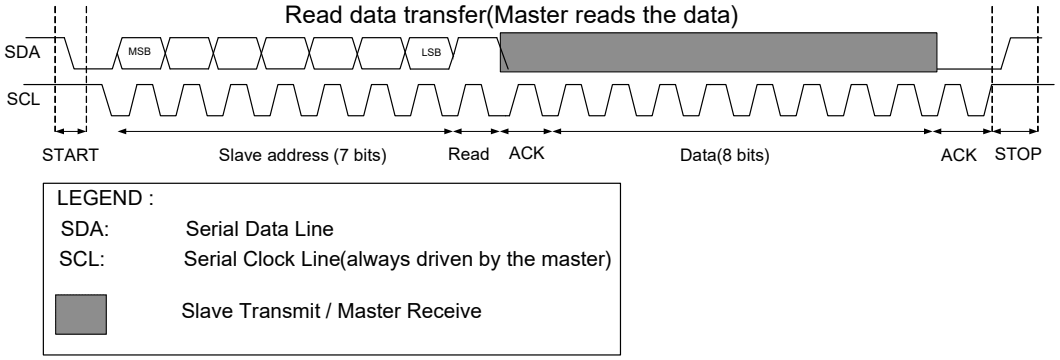
Figure 14-19. Master Write Data Transfer



- A typical write transfer begins with the master generating a START condition on the I²C bus. The master then writes a 7-bit I²C slave address and a write indicator ('0') after the START condition. The addressed slave transmits an acknowledgement 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 acknowledgement (NACK). The absence of an acknowledgement, results in an SDA line value of '1' due to the pull-up resistor implementation.
- If no acknowledgement 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 write data to the bus if it receives an acknowledgement. The addressed slave transmits an acknowledgement to confirm the receipt of the write data. Upon receipt of this acknowledgement, the master may transmit another data byte.
- When the transfer is complete, the master generates a STOP condition.

14.4.4.2 Read Transfer

Figure 14-20. Master Read Data Transfer



- A typical read transfer begins with the master generating a START condition on the I^{2}C bus. The master then writes a 7-bit I^{2}Cslave address and a read indicator ('1') after the START condition. The addressed slave transmits an acknowledgement 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 acknowledgement (NACK) is transmitted. The absence of an acknowledgement, results in an SDA line value of '1' due to the pull-up resistor implementation.
- If no acknowledgement 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 acknowledgement signal. The master transmits an acknowledgement to confirm the receipt of each data byte sent by the slave. Upon receipt of this acknowledgement, 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.

14.4.5 Easy I^{2}C (EZI2C) Protocol

The Easy I^{2}C (EZI2C) protocol is a unique communication scheme built on top of the I^{2}C protocol by Cypress. EZI2C mode is applicable only for slave functionality. It uses a software wrapper 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 at the level of individual frames.

The EZI2C protocol defines an 8-bit EZ address that indexes a memory array (8-bit wide 32 locations) located on the slave device. Five lower bits of the EZ address is used to address these 32 locations. The number of bytes transferred to or from the EZI2C memory array can be found by comparing the EZ address at the START event and the EZ address at the STOP event.

Note The I^{2}C block has a hardware FIFO memory, which is 16 bits wide and 16 locations deep with byte write enable. The access methods for EZ and non-EZ functions are different. In non-EZ mode, the FIFO is split into TXFIFO and RXFIFO. Each has 16-bit wide eight locations. In EZ mode, the FIFO is used as a single memory unit with 8-bit wide 32 locations.

EZI2C has two types of transfers: an EZ write of data from the master to an addressed slave memory location, and a read by the master from an addressed slave memory location.

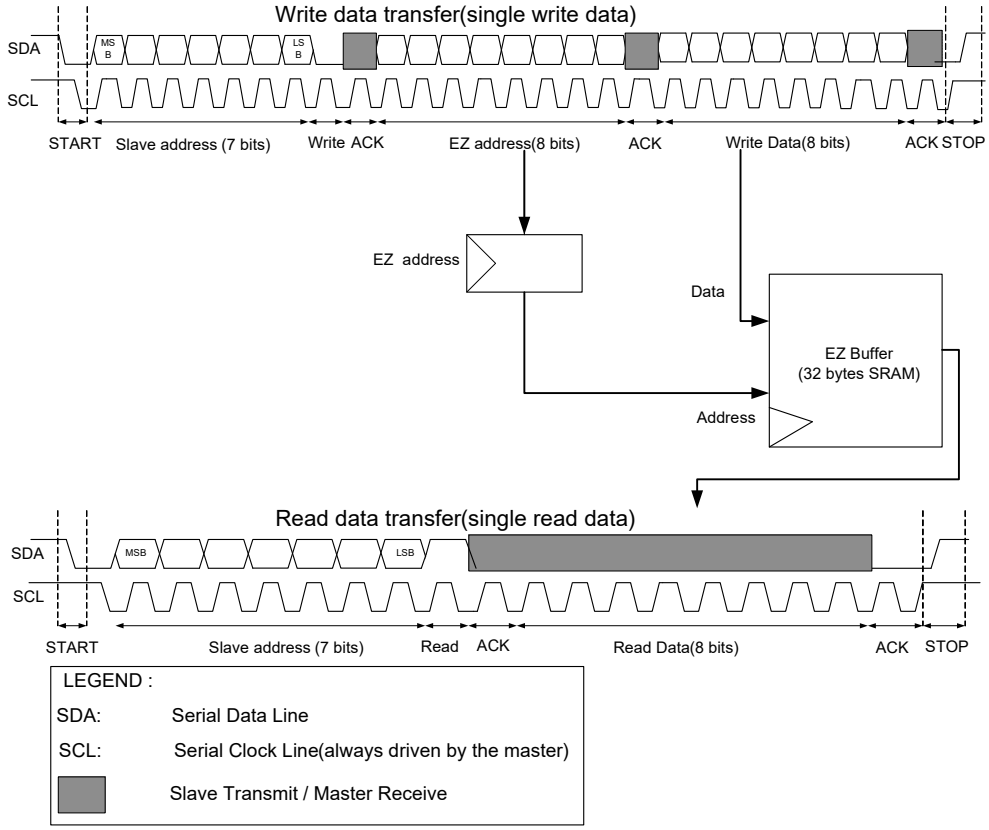
14.4.6 Memory Array Write

An EZ write to a memory array index is by means of an I^{2}C write transfer. The first transmitted write data is used to send an EZ address from the master to the slave. The five lowest significant bits of the write data are used as the "new" EZ address at the slave. Any additional write data elements in the write transfer are bytes that are written to the memory array. The EZ address is automatically incremented by the slave as bytes are written into the memory array. When the EZ address exceeds 32 memory entries, it wraps around to 0.

14.4.7 Memory Array Read

An EZ read from a memory array index is by means of an I^{2}C read transfer. The EZ read relies on an earlier EZ write to have set the EZ address at the slave. The first received read data is the byte from the memory array at the EZ address memory location. The EZ address is automatically incremented as bytes are read from the memory array. When the EZ address exceeds the 32 memory entries, it wraps around to 0.

Figure 14-21. EZI2C Write and Read Data Transfer



See EZ Slave Mode Transfer Example on page 101 for examples.

14.4.8 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 14-17.

Table 14-17. I^{2}C Registers

Register	Function
SCB_CTRL	Enables the SCB I^{2}C 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.

Table 14-17. I²C Registers

Register	Function
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_TX_CTRL	Enables the transmitter and 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.
SCB_RX_FIFO_CTRL	Performs the same function as that of the SCB_TX_FIFO_CTRL register, but for the receiver.
SCB_RX_FIFO_STATUS	Performs the same function as that of the SCB_TX_FIFO_STATUS register, but for the receiver.
SCB_RX_FIFO_RD	Holds the data read from the receiver FIFO. Reading a data frame removes the data frame from the FIFO; behavior is similar to that of a POP operation. This register has a side effect when read by software: a data frame is removed from the FIFO.
SCB_RX_FIFO_RD_SILENT	Holds the data read from the receiver FIFO. Reading a data frame does not remove the data frame from the FIFO; behavior is similar to that of a PEEK operation.
SCB_RX_MATCH	Stores slave device address and is also used as slave device address MASK.
SCB_EZ_DATA	Holds the data in an EZ memory location.

Note Detailed descriptions of the I²C register bits are available in the EZ-PD™ PMG1-S2 MCU Registers TRM.

14.4.9 I²C Interrupts

The I²C block generates interrupts for the following conditions.

- Arbitration lost
- Slave address match
- I2C bus stop/start condition detected
- I2C bus error detected
- I2C byte/word transfer complete
- I2C TX FIFO not full
- I2C TX FIFO empty
- I2C RX FIFO empty
- I2C RX FIFO not empty
- I2C RX FIFO overrun
- I2C RX FIFO full

The I²C interrupt signal is hard-wired to the Cortex-M0 NVIC and cannot be routed to external pins.

The interrupt output is the logical OR of the group of all possible interrupt sources. The interrupt is triggered when any of the enabled interrupt conditions are met. Interrupt status registers are used to determine the actual source of the interrupt. For more information on interrupt registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

14.4.10 Enabling and Initializing the I²C

The following section describes the method to configure the I²C block for standard (non-EZ) mode and EZI2C mode.

Configuring for I²C Standard (Non-EZ) Mode

The I²C interface must be programmed in the following order.

1. Program protocol specific information using the SCB_I2C_CTRL register according to Table 14-18. This includes selecting master - slave functionality.
2. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers, as shown in Table 14-19.
   a. Specify the data frame width.
   b. Specify whether MSB or LSB is the first bit to be transmitted/received.
   c. Enable the transmitter and receiver.
3. Program transmitter and receiver FIFO using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers, respectively, as shown in Table 14-20.
   a. Set the trigger level.
   b. Clear the transmitter and receiver FIFO and Shift registers.
4. Program the SCB_CTRL register to enable the SCB block and select the I²C mode. These register bits are shown in Table 14-21. For a complete description of the I²C registers, see the EZ-PD™ PMG1-S2 MCU Registers TRM.

Table 14-18. SCB_I2C_CTRL Register

Bits	Name	Value	Description
30	SLAVE_MODE	1	Slave mode
31	MASTER_MODE	1	Master mode

Table 14-19. SCB_TX_CTRL/SCB_RX_CTRL Register

Bits	Name	Description
[3:0]	DATA_ WIDTH	'DATA_WIDTH + 1' is the number of bits in the transmitted or received data frame. The valid range is [3, 15].
8	MSB_FIRST	1= MSB first
		0= LSB first
31	ENABLED	Transmitter enable bit for SCB_TX_CTRL and receiver enable bit for SCB_RX_CTRL registers. They must be enabled for all the protocols. Otherwise, the block may not function or the data may get lost.

Table 14-20. SCB_TX_FIFO_CTRL/ SCB_RX_FIFO_CTRL

Bits	Name	Description
[2:0]	TRIGGER_LEVEL	Trigger level. When the transmitter FIFO has less entries or the receiver FIFO has more entries than the value of this field, a transmitter or receiver trigger event is generated in the respective case.
16	CLEAR	When '1', the transmitter or receiver FIFO and the shift registers are cleared.
17	FREEZE	When '1', hardware reads/writes to the transmitter or receiver FIFO have no effect. Freeze does not advance the TX or RX FIFO read/write pointer.

Table 14-21. SCB_CTRL Registers

Bits	Name	Value	Description
[25:24]	MODE	00	I²C mode
		01	SPI mode
		10	UART mode
		11	Reserved
31	ENABLED	0	SCB block enabled
		1	SCB block disabled

Configuring for EZI2C Mode

To configure the SCB block for EZI2C mode, set the following I²C register bits

1. Select the EZI2C mode by writing '1' to the EZ_MODE bit (bit 10) of the SCB_CTRL register.
2. Follow steps 2 to 4 mentioned in **Configuring for I2C Standard (Non-EZ) Mode**.
3. Set the S_READY_ADDR_ACK (bit 12) and S_READY_DATA_ACK (bit 13) bits of the SCB_I2C_CTRL register.

14.4.11 Internal and External Clock Operation in I²C

The SCB supports both internally and externally clocked operation for data-rate generation. Internally clocked operations use a clock signal derived from the EZ-PD™ PMG1-S2 MCU system bus clock. Externally clocked operations use a clock provided by the user. Externally clocked operation allows limited functionality in the Deep-Sleep power mode, in which on-chip clocks are not active. For more information on system clocking, see the Clocking System chapter on page 50.

Externally clocked operation is limited to the following cases:

- Slave functionality.
- EZ functionality. TX and RX FIFOs do not support externally clocked operation; therefore, it is not used for non-EZ functionality.

Internally and externally clocked operations are determined by two register fields of the SCB_CTRL register:

- **EC_AM_MODE (Externally Clocked Address Matching Mode)**: Indicates whether I²C address matching is internally ('0') or externally ('1') clocked.
- **EC_OP_MODE (Externally Clocked Operation Mode)**: Indicates whether the rest of the protocol operation (besides I²C address match) is internally ('0') or externally ('1') clocked. As mentioned earlier, externally clocked operation does not support non-EZ functionality.

These two register fields determine the functional behavior of I²C. The register fields should be set based on the required behavior in Active, Sleep, and Deep-Sleep system power modes. Improper setting may result in faulty behavior in certain power modes. Table 14-22 and Table 14-22 describe the settings for I²C in EZ and non-EZ mode.

14.4.11.1 I²C Non-EZ Operation Mode

Externally clocked operation is not supported for non-EZ functionality because there is no FIFO support for this mode. So, the EC_OP_MODE should always be set to '0' for non-EZ mode. However, EC_AM_MODE can be set to '0' or '1'. Table 14-9 gives an overview of the possibilities. The combination EC_AM_MODE = 0 and EC_OP_MODE = 1 is invalid and the block will not respond.

EC_AM_MODE is '0' and EC_OP_MODE is '0'.

This setting only works in Active and Sleep system power modes. All the I²C's functionality is provided in the internally clocked domain.

EC_AM_MODE is '1' and EC_OP_MODE is '0'.

This setting works in Active and Deep-Sleep system power modes. I²C address matching is performed by the externally clocked logic in both these modes. When the externally clocked logic matches the address, it sets a wakeup interrupt cause bit, which can be used to generate an interrupt to wakeup the CPU.

- In Active system power mode, the CPU is active and the wakeup interrupt cause is disabled (associated MASK bit is '0'). The externally clocked logic takes care of the address matching and the internally locked logic takes care of the rest of the I²C transfer.
- In the Sleep mode, wakeup interrupt cause can be either enabled or disabled based on the application. The remaining operations are similar to the Active mode.
- In the Deep-Sleep mode, the CPU is shut down and will wake up on I²C activity if the wakeup interrupt cause is enabled. CPU wakeup up takes time and the ongoing I²C transfer is either negatively acknowledged (NACK) or the clock is stretched. In the case of a NACK, the internally clocked logic takes care of the first I²C transfer after it wakes up. For clock stretching, the internally clocked logic takes care of the ongoing/stretched transfer when it wakes up. The register bit S_NOT_READY_ADDR_NACK (bit 14) of the SCB_I2C_CTRL register determines whether the externally clocked logic performs a negative acknowledge ('1') or clock stretch ('0').

14.4.11.2 EZ Operation Mode

EZ mode has three possible settings. EC_AM_MODE can be set to '0' or '1' when EC_OP_MODE is '0' and EC_AM_MODE must be set to '1' when EC_OP_MODE is '1'. Table 14-22 gives an overview of the possibilities. The grey cells indicate a possible, yet not recommended setting because it involves a switch from the externally clocked logic (slave selection) to the internally clocked logic (rest of the operation). The combination EC_AM_MODE=0 and EC_OP_MODE=1 is invalid and the block will not respond.

Table 14-22. I²C EZ Mode

I²C, EZ Mode				
System Power Mode	EC_OP_MODE= 0		EC_OP_MODE = 1	
	EC_AM_MODE = 0	EC_AM_MODE = 1	EC_AM_MODE = 1	EC_AM_MODE=0
Active and Sleep	Address match using internal clock Operation using internal clock	Address match using external clock Operation using internal clock	Address match using external clock Operation using external clock	Invalid configuration
Deep-Sleep	Not supported	Address match using external clock Operation using internal clock	Address match using external clock Operation using external clock	

- **EC_AM_MODE is '0' and EC_OP_MODE is '0'**. This setting only works in Active/Sleep system power mode.
- **EC_AM_MODE is '1' and EC_OP_MODE is '0'**. This setting works same as I²C non-EZ mode.
- **EC_AM_MODE is '1' and EC_OP_MODE is '1'**. This setting works in Active system power mode and Deep-Sleep system power mode.

The SCB functionality is provided in the externally clocked domain. Note that this setting results in externally clocked accesses to the block's SRAM. These accesses may conflict with internally clocked accesses from the device. This may cause wait states or bus errors. The field FIFO_BLOCK (bit 17) of the SCB_CTRL register determines whether wait states ('1') or bus errors ('0') are generated.

14.4.12 Wake up from Sleep

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

Address ACK - The I²C slave executes clock stretching and waits until the device wakes up and ACKs the address.

Address NACK - The I²C 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 or multi-master-slave modes.

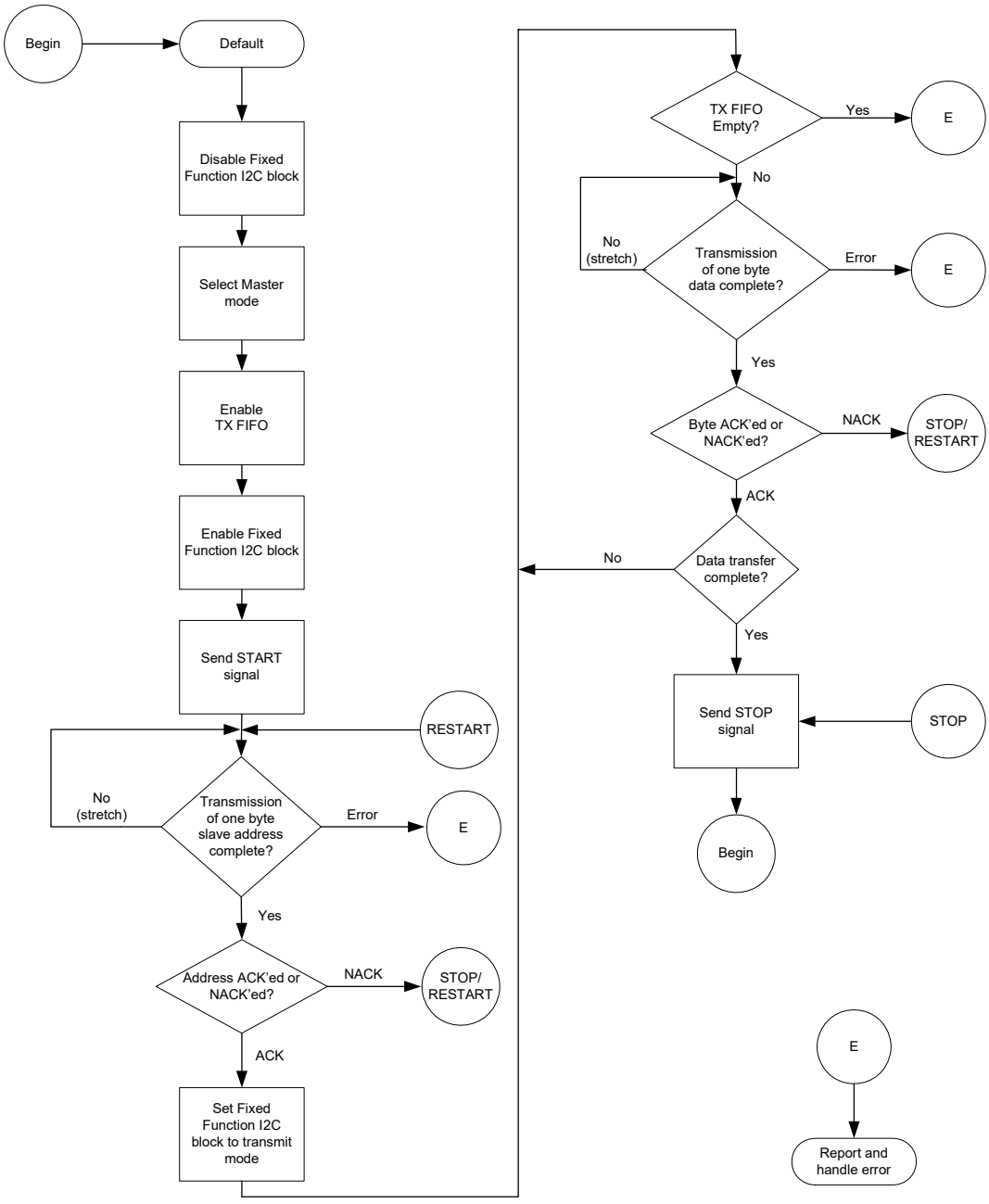
Note The interrupt bit WAKEUP (bit 0) of the INTR_I2C_EC register must be enabled for the I²C to wake up the device on slave address match while switching to the Sleep mode.

14.4.13 Master Mode Transfer Examples

Master mode transmits or receives data.

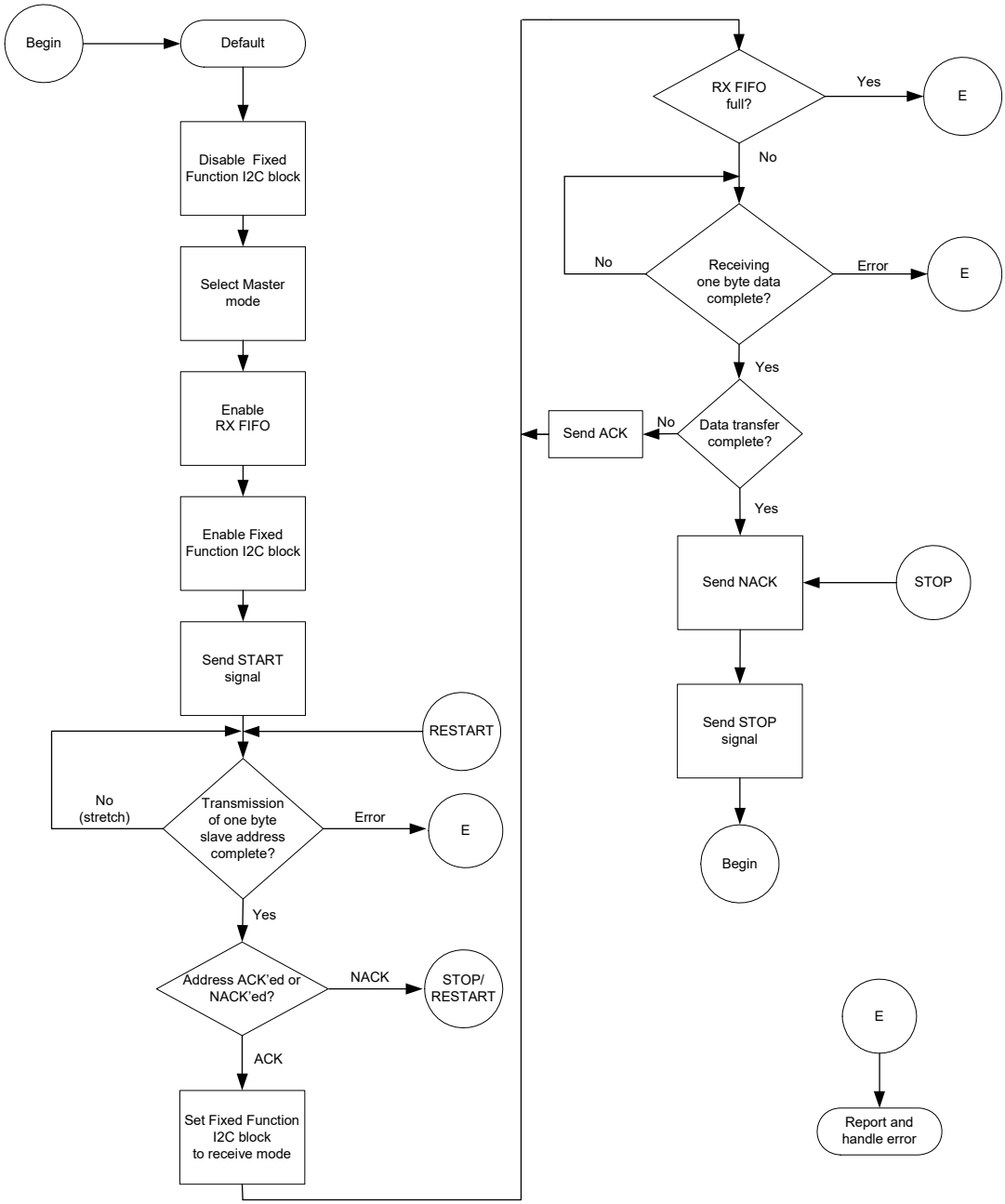
14.4.13.1 Master Transmit

Figure 14-22. Single Master Mode Write Operation Flow Chart



14.4.13.2 Master Receive

Figure 14-23. Single Master Mode Read Operation Flow Chart

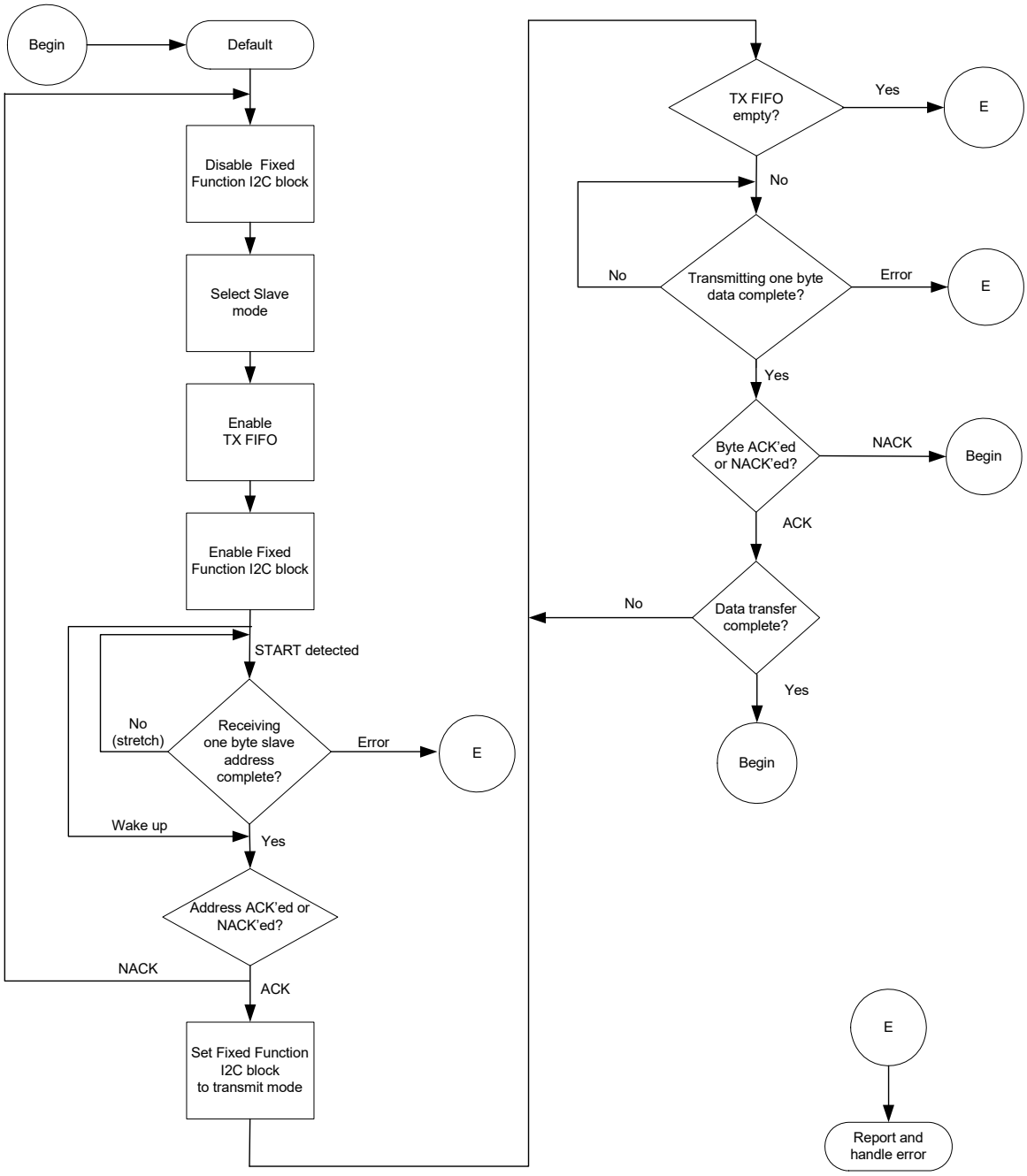


14.4.14 Slave Mode Transfer Examples

Slave mode transmits or receives data.

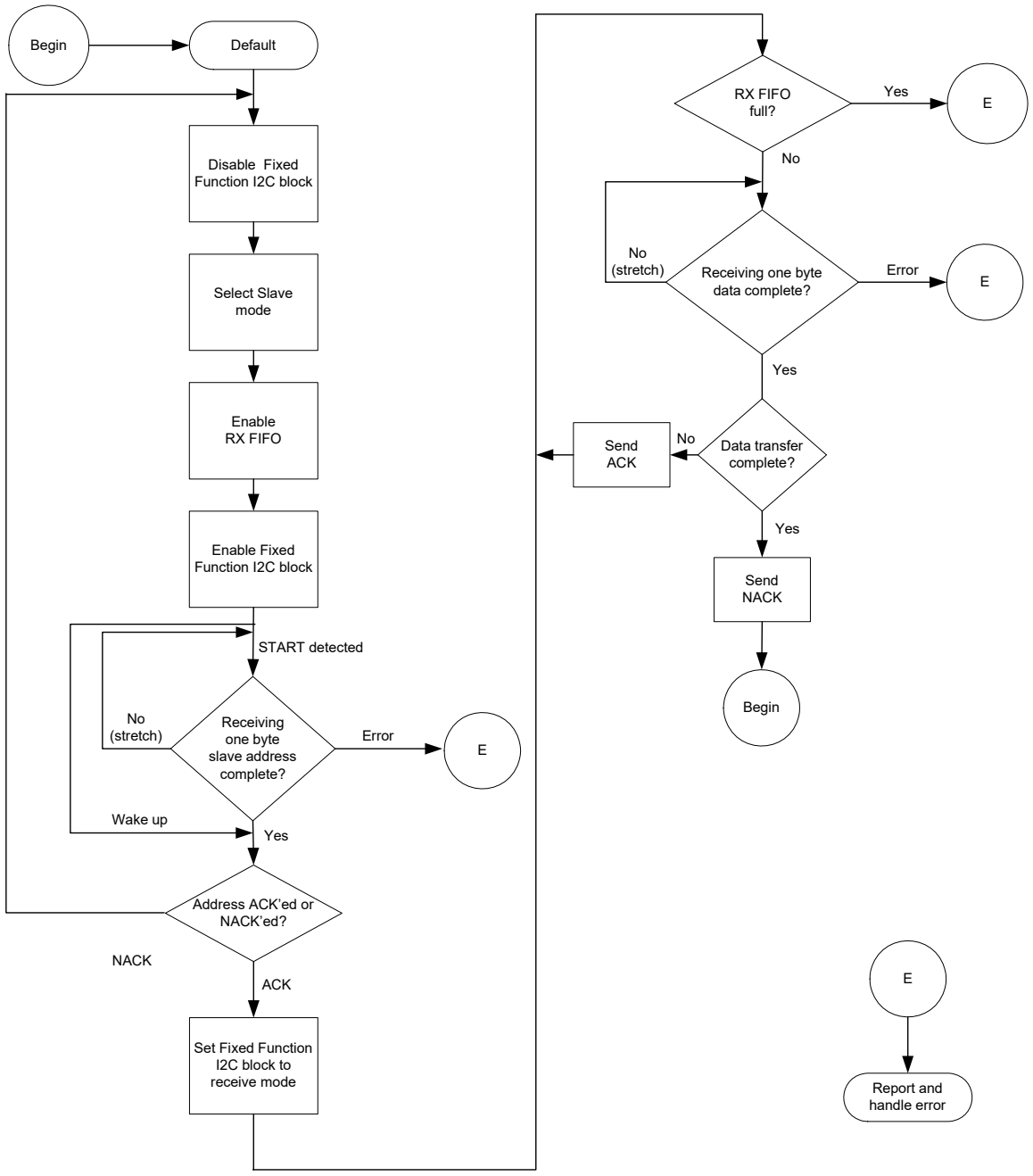
14.4.14.1 Slave Transmit

Figure 14-24. Slave Mode Write Operation Flow Chart



14.4.14.2 Slave Receive

Figure 14-25. Slave Mode Read Operation Flow Chart

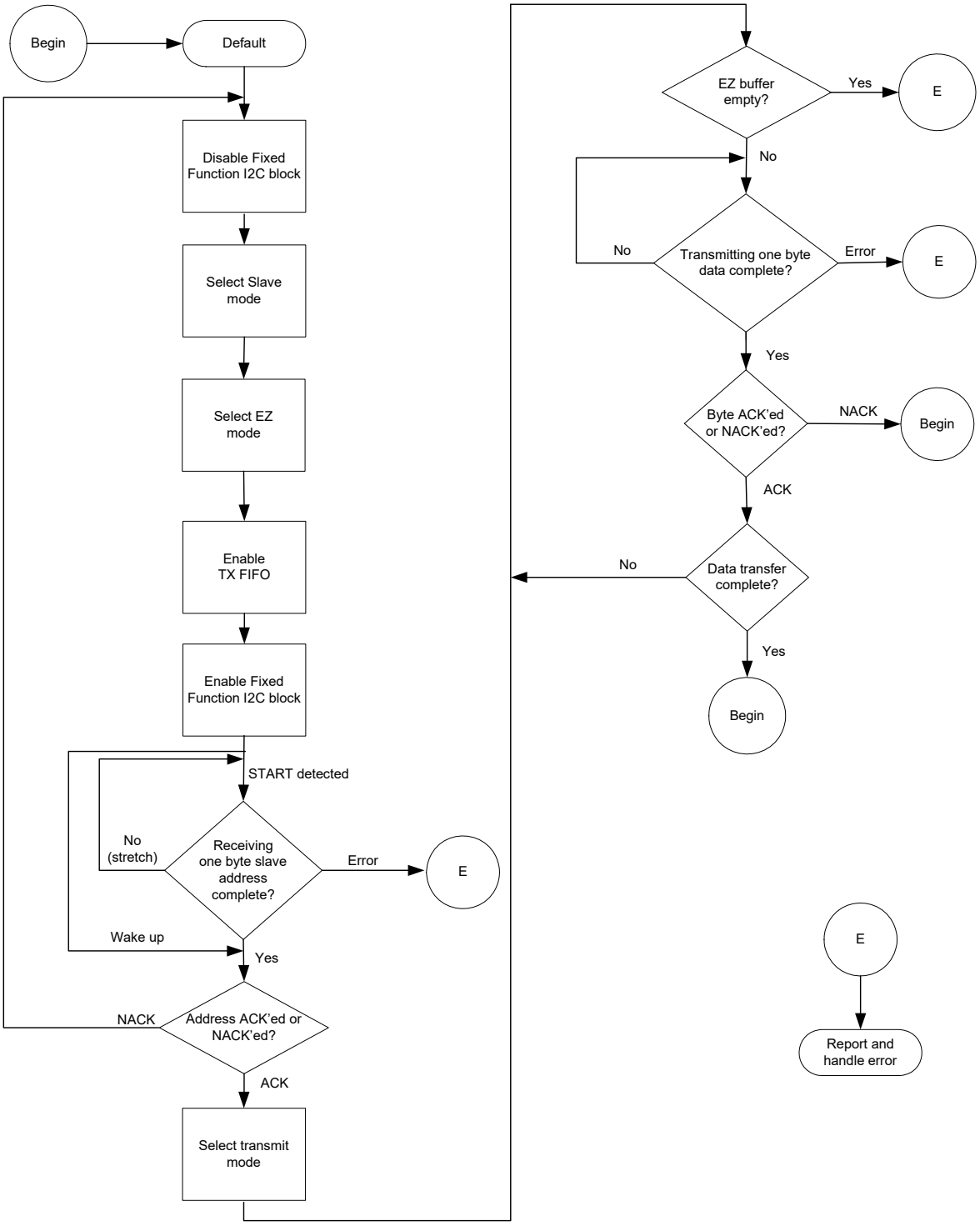


14.4.15 EZ Slave Mode Transfer Example

The EZ Slave mode transmits or receives data.

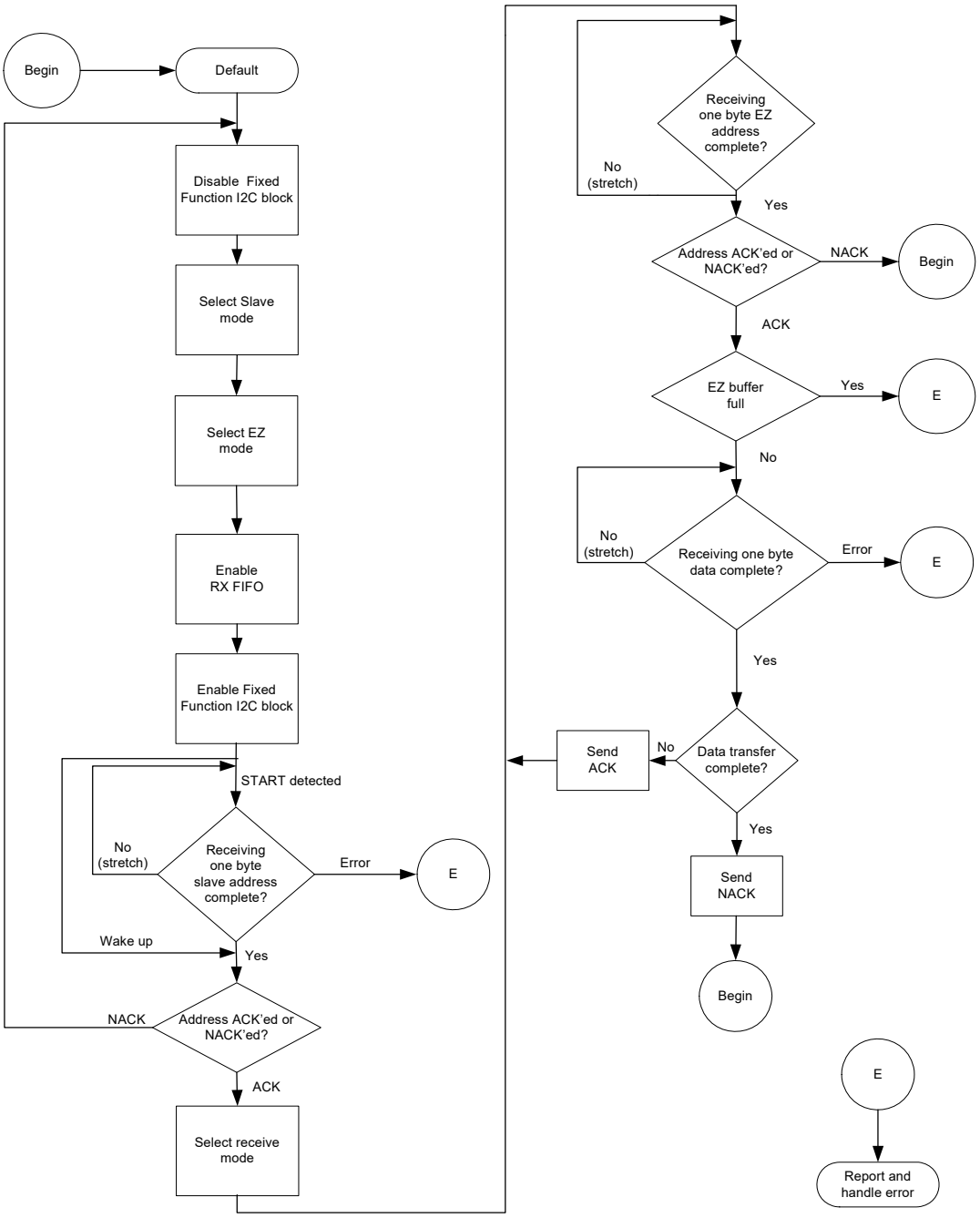
14.4.15.1 EZ Slave Transmit

Figure 14-26. EZI2C Slave Mode Write Operation Flow Chart



14.4.15.2 EZ Slave Receive

Figure 14-27. EZI2C Slave Mode Read Operation Flow Chart

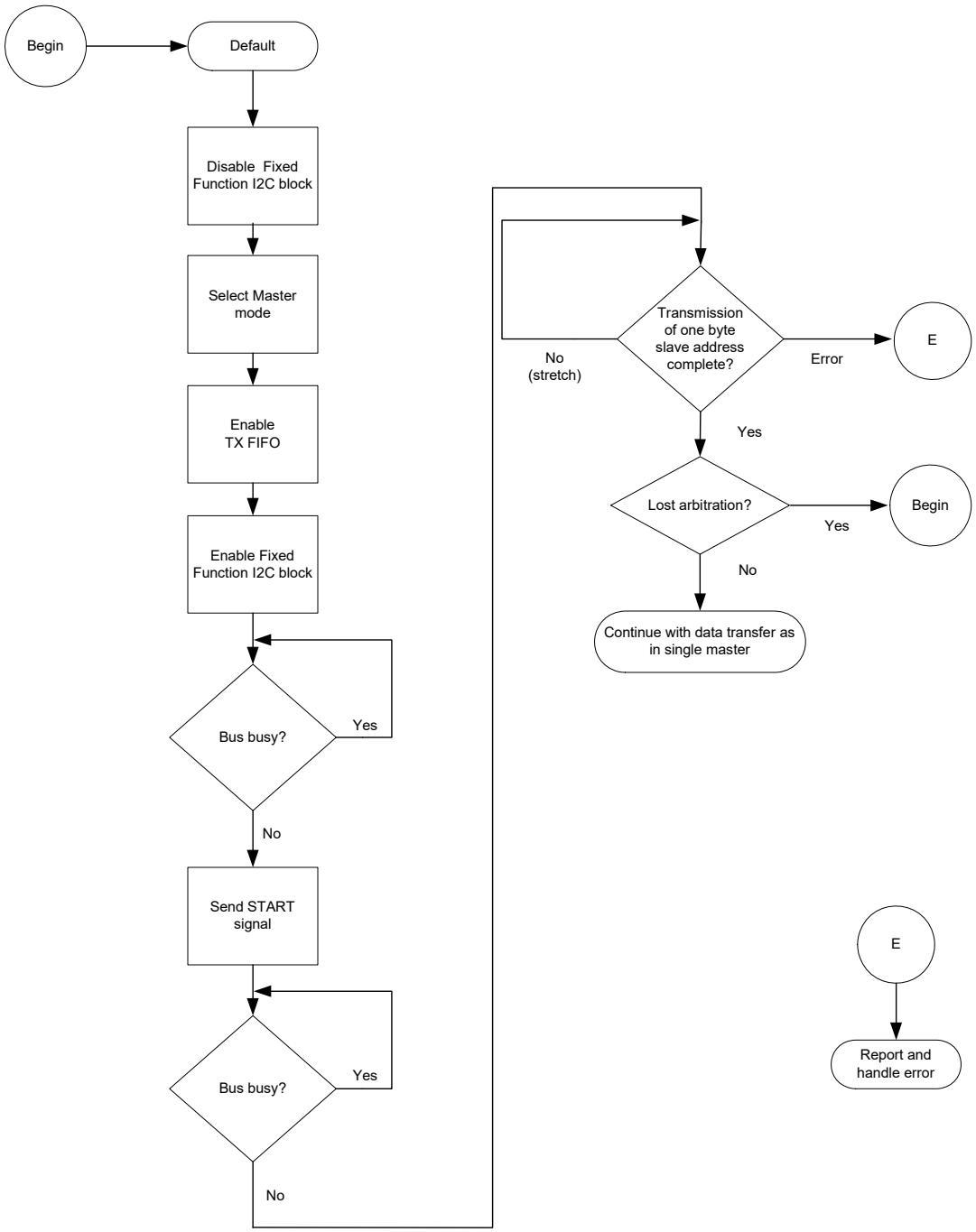


14.4.16 Multi-Master Mode Transfer Example

In multi-master mode, data can be transferred with the slave mode enabled or not enabled.

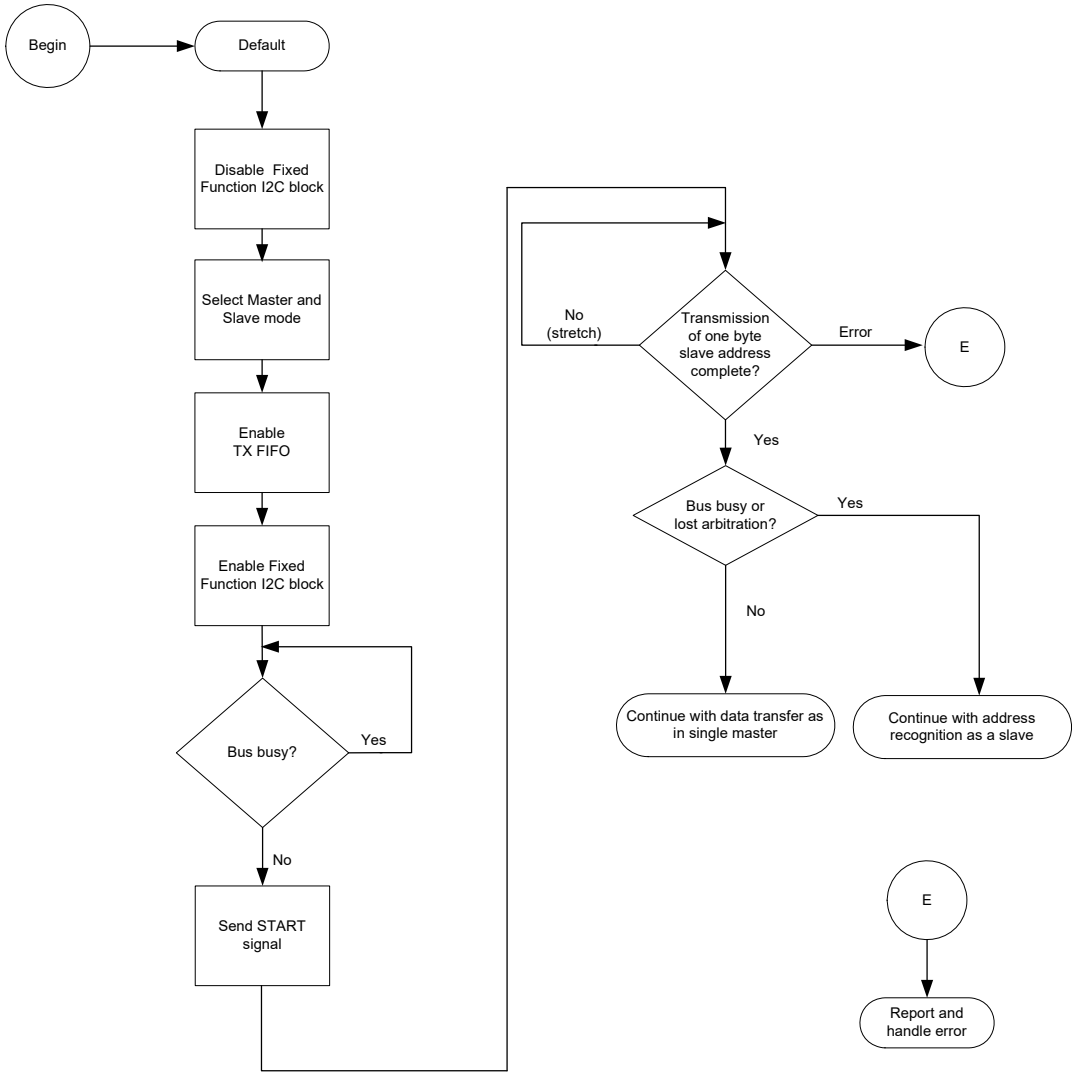
14.4.16.1 Multi-Master - Slave Not Enabled

Figure 14-28. Multi-Master, Slave Not Enabled Flow Chart



14.4.16.2 Multi-Master - Slave Enabled

Figure 14-29. Multi-Master, Slave Enabled Flow Chart



15. Timer, Counter, and PWM

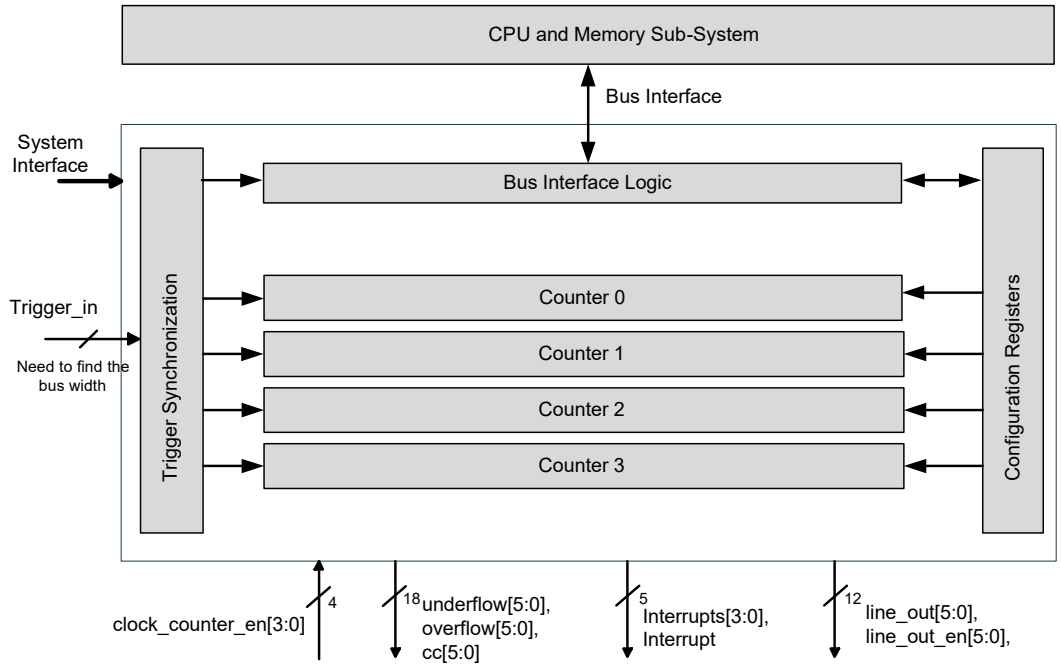
EZ-PD™ PMG1-S2 MCU includes four blocks of the Timer, Counter, and Pulse Width Modulator (TCPWM) block. Each TCPWM in EZ-PD™ PMG1-S2 MCU implements the 16-bit timer, counter, pulse width modulator (PWM), and quadrature decoder functionality. The block can be used to measure the period and pulse width of an input signal (timer), find the number of times a particular event occurs (counter), generate PWM signals, or decode quadrature signals. This chapter explains the features, implementation, and operational modes of the TCPWM block.

15.1 Features

- Four 16-bit timers, counters, or pulse width modulators (PWM)
- The TCPWM block supports the following operational modes:
  - Timer
  - Counter
  - Capture
  - Quadrature decoding
  - Pulse width modulation
  - Pseudo-random PWM
  - PWM with dead time
- Multiple counting modes – up, down, and up/down
- Clock prescaling (division by 1, 2, 4, ... 64, 128)
- Double buffering of compare/capture and period values
- Supports interrupt on:
  - Terminal Count (TC) – The final value in the counter register is reached
  - Capture/Compare (CC) – The count is captured to the capture/compare register or the counter value equals the compare value
- Synchronized counters – The counters can reload, start, stop, and count at the same time
- Complementary line output for PWMs

15.2 Block Diagram

Figure 15-1. TCPWM Block Diagram



The block has these interfaces:

- Bus interface: Connects the block to the CPU subsystem.
- Interrupts: Provides interrupt request signals from each counter, based on terminal count (TC) or CC conditions, and a combined interrupt signal generated by the logical OR of all six interrupt request signals.
- System interface: Consists of control signals such as clock and reset from the system resources subsystem.

This TCPWM block can be configured by writing to the TCPWM registers. See TCPWM Registers on page 126 for more information on all registers required for this block.

15.2.1 Enabling and Disabling Counter in TCPWM Block

The counter can be enabled by setting the COUNTER_ENABLED field (bit 0) of the control register TCPWM_CTRL.

Note The counter must be configured before enabling it. If the counter is enabled after being configured, registers are updated with the new configuration values. Disabling the counter retains the values in the registers until it is enabled again (or reconfigured).

15.2.2 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.

Clock Prescaling: counter_clock can be prescaled, with divider values of 1, 2, 4… 64, 128. This is done by modifying the GENERIC field of the counter control (TCPWM_CNT_CTRL) register, as shown in Table 15-1.

Table 15-1. Bit-Field Setting to Prescale Counter Clock

GENERIC[10:8]	Description
0	Divide by 1
1	Divide by 2
2	Divide by 4
3	Divide by 8
4	Divide by 16
5	Divide by 32
6	Divide by 64
7	Divide by 128

Note Clock prescaling cannot be done in quadrature mode and pulse width modulation mode with dead time (PWM-DT).

15.2.3 Events Based on Trigger Inputs

These are the events triggered by hardware or software.

- Reload
- Start
- Stop
- Count
- Capture/switch

Hardware triggers can be level signal, rising edge, falling edge, or both edges.

Figure 15-2. TCPWM Trigger Selection and Event Detection

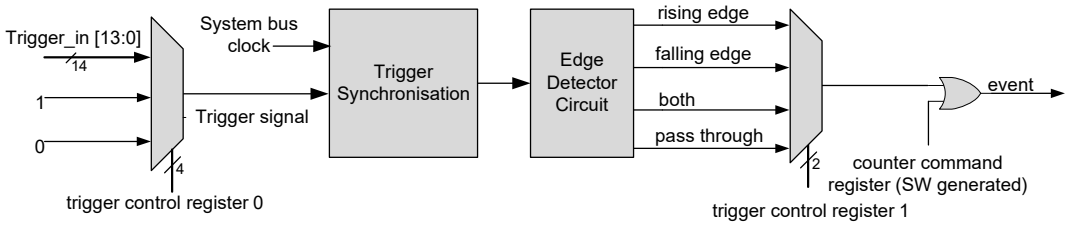


Figure 15-2 shows the trigger selection and event detection in the TCPWM block. The trigger control register 0 (TCPWM_CNT_TR_CTRL0) selects one of the 16 trigger inputs as the event signal. Additionally, a constant '0' and '1' signals are available to be used as the event signal.

Any edge (rising, falling, or both) or level (high or low) can be selected for the occurrence of an event by configuring the trigger control register 1 (TCPWM_CNT_TR_CTRL1). This edge/level configuration can be selected for each trigger event separately. Alternatively, firmware can generate an event by writing to the counter command register (TCPWM_CMD), as shown in Figure 15-2.

The events derived from these triggers can have different definitions in different modes of the TCPWM block.

- **Reload:** A reload event initializes and starts the counter.
  - In up counting mode, the count register (TCPWM_CNT_COUNTER) is initialized with ‘0’.
  - In down counting mode, the counter is initialized with the period value stored in the TCPWM_CNT_PERIOD register.
  - In up/down counting mode, the count register is initialized with ‘0’.
  - In quadrature mode, the reload event acts as a quadrature index event. An index/reload event indicates a completed rotation and can be used to synchronize quadrature decoding.
- **Start:** A start event is used to start counting; it can be used after a stop event or after re-initialization of the counter register to any value by software. Note that the count register is not initialized on this event.
  - In quadrature mode, the start event acts as quadrature phase input phiB, which is explained in detail in Quadrature Decoder Mode on page 116.

- **Count:** A count event causes the counter to increment or decrement, depending on its configuration.
  - In quadrature mode, the count event acts as quadrature phase input phiA.
- **Stop:** A stop event stops the counter from incrementing or decrementing. A start event will start the counting again.
  - In the PWM modes, the stop event acts as a kill event. A kill event disables all the PWM output lines.
- **Capture:** A capture event copies the counter register value to the capture register and capture register value to the buffer capture register. In the PWM modes, the capture event acts as a switch event. It switches the values of the capture/compare and period registers with their buffer counterparts. This feature can be used to modulate the pulse width and frequency.

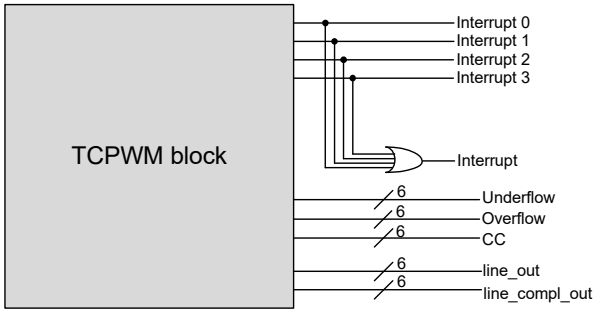
Notes

- All trigger inputs are synchronized to the HFCLK.
- When more than one event occurs in the same counter clock 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 prescaled (divided) counter clock is used.

15.2.4 Output Signals

The TCPWM block generates several output signals, as shown in Figure 15-3.

Figure 15-3. TCPWM Output Signals



15.2.4.1 Signals upon Trigger Conditions

- Counter generates an internal overflow (OV) condition when counting up and the count register reaches the period value.
- Counter generates an internal underflow (UN) condition when counting down and the count register reaches zero.
- The capture/compare (CC) condition is generated by the TCPWM when the counter is running and one of the following conditions occur:
  - The counter value equals the compare value.
  - A capture event occurs - When a capture event occurs, the TCPWM_CNT_COUNTER register value is copied to the capture register and the capture register value is copied to the buffer capture register.

Note These signals, when they occur, remain at logic high for one cycle of the HFCLK. For reliable operation, the condition that causes this trigger should occur in a frequency less than a quarter of the HFCLK. For example, if the HFCLK is running at 24 MHz, the condition causing the trigger should occur at a frequency less than 6 MHz.

15.2.4.2 Interrupts

The TCPWM block provides a dedicated interrupt output signal from the counter. An interrupt can be generated for a TC condition or a CC condition. The exact definition of these conditions is mode-specific. All six interrupt output signals from the four TCPWMs are also OR'ed together to produce a single interrupt output signal.

Four registers are used for interrupt handling in this block, as shown in Table 15-2.

Table 15-2. 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 with '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.

15.2.4.3 Outputs

The TCPWM has two outputs, line_out and line_compl_out (complementary of line_out). Note that the OV, UN, and CC conditions can be used to drive line_out and line_compl_out if needed, by configuring the TCPWM_CNT_TR_CTRL2 register (see Table 15-3).

Table 15-3. Configuring Output Line for OV, UN, and CC Conditions

Field	Bit	Value	Event	Description
CC_MATCH_MODE Default Value = 3	1:0	0	Set line_out to '1	Configures output line on a compare match (CC) event
		1	Clear line_out to '0	
		2	Invert line_out	
		3	No change	
OVERFLOW_MODE Default Value = 3	3:2	0	Set line_out to '1	Configures output line on a overflow (OV) event
		1	Clear line_out to '0	
		2	Invert line_out	
		3	No change	

Table 15-3. Configuring Output Line for OV, UN, and CC Conditions

Field	Bit	Value	Event	Description
UNDERFLOW_-MODE Default Value = 3	5:4	0	Set line_out to '1	Configures output line on a under-flow (UN) event
		1	Clear line_out to '0	
		2	Invert line_out	
		3	No change	

15.2.5 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 15-4.

Table 15-4. Power Modes in TCPWM Block

Power Mode	Block Status
Active	This block is fully operational in this mode with clock running and power switched on.
Sleep	All counter clocks are on, but bus interface cannot be accessed.
Deep-Sleep	In this mode, the power to this block is still on but no bus clock is provided; hence, the logic is not functional. All the configuration registers will keep their state.

15.3 Modes of Operation

The counter block can function in six operational modes, as shown in Table 15-5. The MODE [26:24] field of the counter control register (TCPWM_CNTx_CTRL) configures the counter in the specific operational mode.

Table 15-5. Operational Mode Configuration

Mode	MODE Field [26:24]	Description
Timer	000	Implements a timer or counter. The counter increments or decrements by '1' at every counter clock cycle in which a count event is detected.
Capture	010	Implements a timer or counter with capture input. The counter increments or decrements by '1' at every counter clock cycle in which a count event is detected. When a capture event occurs, the counter value copies into the capture register.
Quadrature Decoder	011	Implements a quadrature decoder, where the counter is decremented or incremented, based on two phase inputs according to the selected (X1, X2 or X4) encoding scheme.
PWM	100	Implements edge/center-aligned PWMs with an 8-bit clock prescaler and buffered compare/period registers.
PWM-DT	101	Implements edge/center-aligned PWMs with configurable 8-bit dead time (on both outputs) and buffered compare/period registers.
PWM-PR	110	Implements a pseudo-random PWM using a 16-bit linear feedback shift register (LFSR).

The counter can be configured to count up, down, and up/down by setting the UP_DOWN_MODE[17:16] field in the TCPWM_CNTx_CTRL register, as shown in Table 15-6.

Table 15-6. Counting Mode Configuration

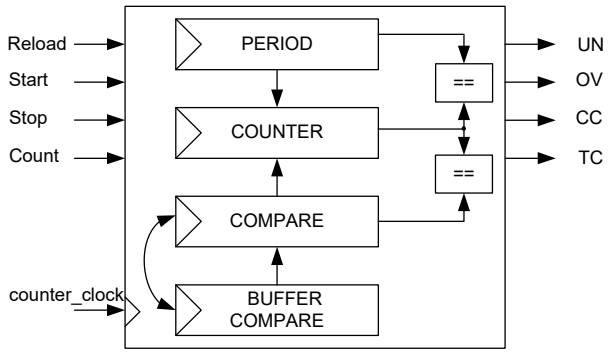
Counting Modes	UP_DOWN_MODE[17:16]	Description
UP Counting Mode	00	Increments the counter until the period value is reached. A Terminal Count (TC) condition is generated when counter reaches the period value.
DOWN Counting Mode	01	Decrements the counter from the period value until 0 is reached. A TC condition is generated when the counter reaches ‘0’.
UP/DOWN Counting Mode 0	10	Increments the counter until the period value is reached, and then decrements the counter until ‘0’ is reached. A TC condition is generated only when ‘0’ is reached.
UP/DOWN Counting Mode 1	11	Similar to up/down counting mode 0 but a TC condition is generated when the counter reaches ‘0’ and when the counter value reaches the period value.

15.3.1 Timer Mode

The timer mode is commonly used to measure time of occurrence of an event or to measure the time difference between two events.

15.3.1.1 Block Diagram

Figure 15-4. Timer Mode Block Diagram



15.3.1.2 How It Works

The timer can be configured to count in up, down, and up/down counting modes. It can also be configured to run in either continuous mode or one-shot mode.

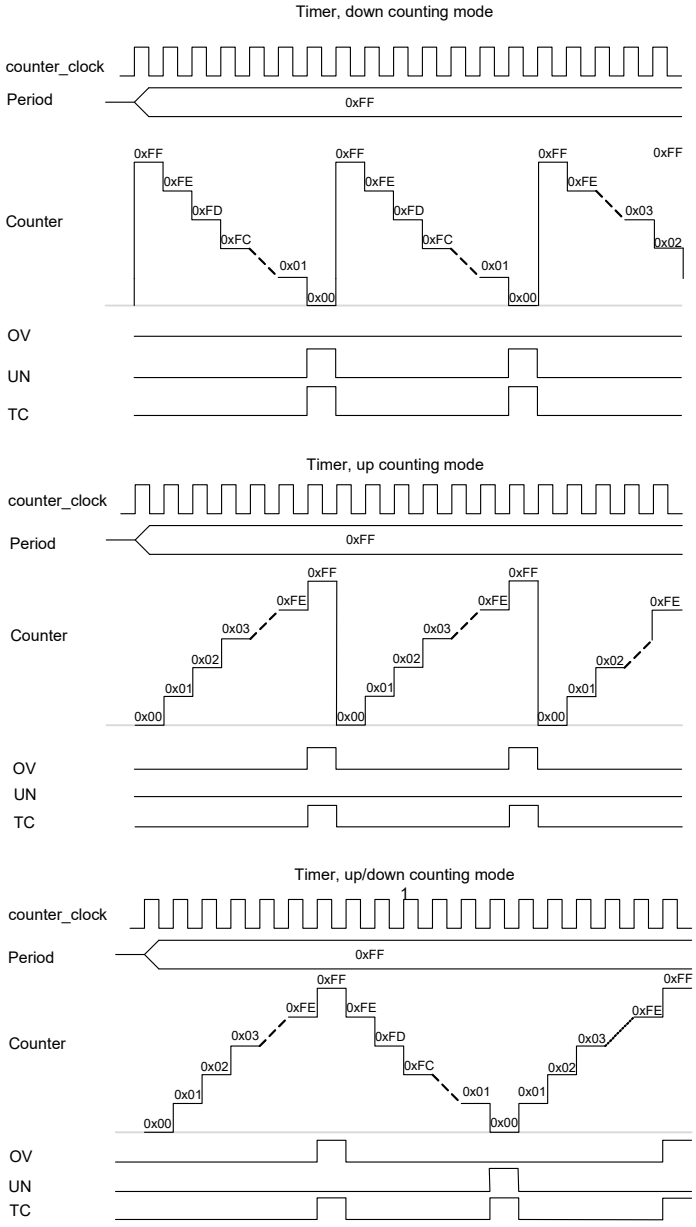
The following explains the working of the timer:

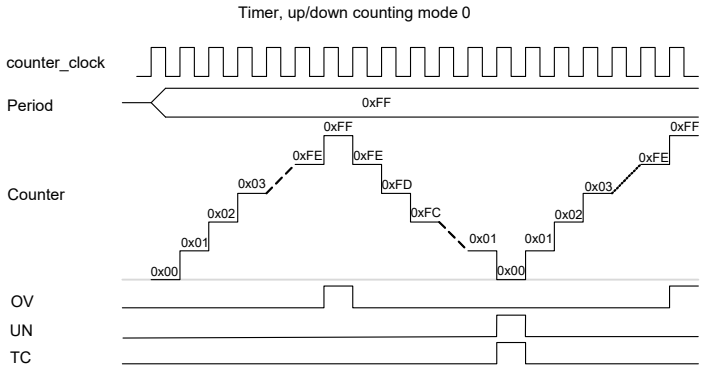
- The timer is an up, down, and up/down counter.
  - The current count value is stored in the count register (TCPWM_CNTx_COUNTER). **Note** It is not recommended to write values to this register while the counter is running.
  - The period value for the timer is stored in the period register.
- The counter is re-initialized in different counting modes as follows:
  - In the up counting mode, after the count reaches the period value, the count register is automatically reloaded with 0.
  - In the down counting mode, after the count register reaches zero, the count register is reloaded with the value in the period register.
  - In the up/down counting modes, the count register value is not updated upon reaching the terminal values. Instead the direction of counting changes when the count value reaches 0 or the period value.
- The CC condition is generated when the count register value equals the compare register value. Upon this condition, the compare register and buffer compare register switch their values if enabled by the AUTO_RELOAD_CC bit-field of the counter control (TCPWM_CNT_CTRL) register. This condition can be used to generate an interrupt request.

Figure 15-5 shows the timer operational mode of the counter in four different counting modes. The period register contains the maximum counter value.

- In the up counting mode, a period value of A results in A+1 counter cycles (0 to A).
- In the down counting mode, a period value of A results in A+1 counter cycles (A to 0).
- In the two up/down counting modes (both modes 0 and 1 both), a period value of A results in 2*A counter cycles (0 to A and back to 0).

Figure 15-5. Timing Diagram for Timer in Multiple Counting Modes





Note The OV and UN signals remain at logic high for one cycle of the HFCLCK, as explained in Signals upon Trigger Conditions on page 109. The figures in this chapter assumes that HFCLK and counter clock are the same.

15.3.1.3 Configuring Counter for Timer Mode

The steps to configure the counter for Timer mode of operation and the affected register bits are as follows.

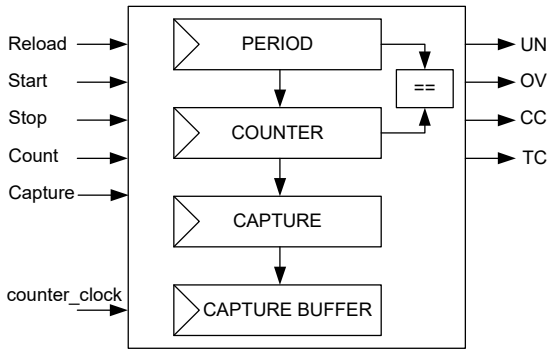
1. Disable the counter by writing '0' to the COUNTER_ENABLED field of the TCPWM_CTRL register.
2. Select Timer mode by writing '000' to the MODE[26:24] field of the TCPWM_CNTx_CTRL register.
3. Set the required 16-bit period in the TCPWM_CNT_PERIOD register.
4. Set the 16-bit compare value in the TCPWM_CNT_CC register and the buffer compare value in the TCPWM_CNT_CC_BUFF register. Set AUTO_RELOAD_CC field of counter control register, if required to switch values at every CC condition.
5. Set clock prescaling by writing to the GENERIC[10:8] field of the counter control (TCPWM_CNT_CTRL) register, as shown in Table 15-1.
6. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register, as shown in Table 15-6.
7. 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.
8. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (Reload, Start, Stop, Capture, and Count).
9. Set the TCPWM_CNT_TR_CTRL1 register to select the edge of the trigger that causes the event (Reload, Start, Stop, Capture, and Count).
10. If required, set the interrupt upon TC or CC condition, as shown in Interrupts on page 109.
11. 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.

15.3.2 Capture Mode

In the capture mode, the counter value can be captured at any time either through a firmware write to command register (TCPWM_CMD) or a capture trigger input. This mode is used for period and pulse width measurement.

15.3.2.1 Block Diagram

Figure 15-6. Capture Mode Block Diagram



15.3.2.2 How it Works

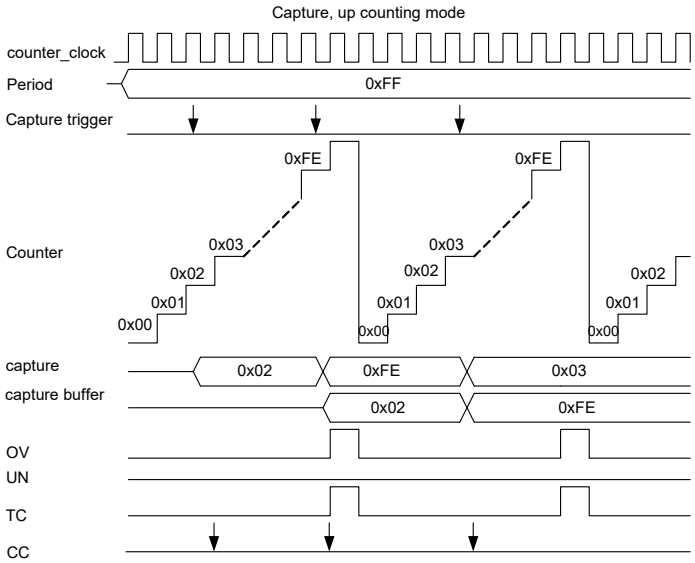
The counter can be set to count in up, down, and up/down counting modes by configuring the UP_DOWN_MODE[17:16] bit-field of the counter control register (TCPWM_CNT_CTRL).

Operation in capture mode occurs as follows:

- During a capture event, generated either by hardware or software, the current count register value is copied to the capture register (TCPWM_CNT_CC) and the capture register value is copied to the buffer capture register (TCPWM_CNT_C-C_BUFF).
- A pulse on the CC output signal is generated when the counter value is copied to the capture register. This condition can also be used to generate an interrupt request.

Figure 15-7 illustrates the capture behavior in the up counting mode.

Figure 15-7. Timing Diagram of Counter in Capture Mode, Up Counting Mode



In the figure, observe that:

- The period register contains the maximum count value.
- Internal overflow (OV) and TC conditions are generated when the counter reaches the period value.
- A capture event is only possible at the edges or through software. Use trigger control register 1 to configure the edge detection.
- Multiple capture events in a single clock cycle are handled as:
  - Even number of capture events - no event is observed

- Odd number of capture events - single event is observed

This happens when the capture signal frequency is greater than the counter_clock frequency.

15.3.2.3 Configuring Counter for Capture Mode

The steps to configure the counter for Capture mode operation and the affected register bits are as follows.

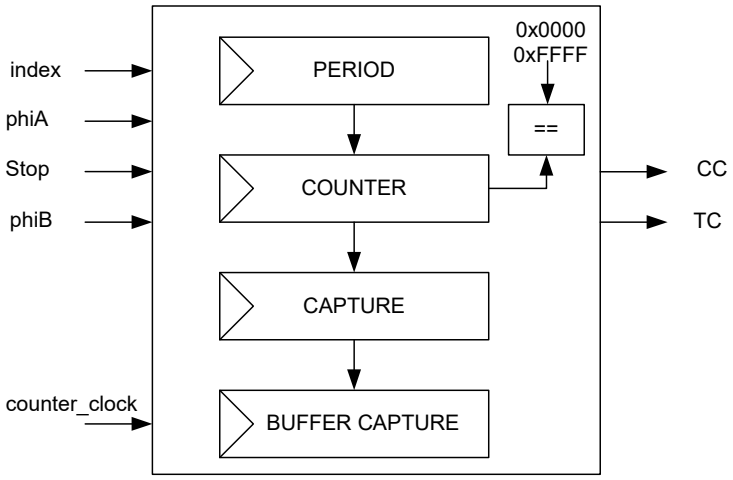
1. Disable the counter by writing '0' to the COUNTER_ENABLED field of the TCPWM_CTRL register.
2. Select Capture mode by writing '010' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required 16-bit period in the TCPWM_CNT_PERIOD register.
4. Set clock prescaling by writing to the GENERIC[10:8] field of the TCPWM_CNT_CTRL register, as shown in Table 15-1.
5. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register, as shown in Table 15-6.
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, as shown in Interrupts on page 109.
10. 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.

15.3.3 Quadrature Decoder Mode

Quadrature decoders are used to determine speed and position of a rotary device (such as servo motors, volume control wheels, and PC mice). The quadrature encoder signals are used as phiA and phiB inputs to the decoder.

15.3.3.1 Block Diagram

Figure 15-8. Quadrature Mode Block Diagram



15.3.3.2 How It Works

Quadrature decoding only runs on counter_clock. It can operate in three sub-modes: X1, X2, and X4. These encoding modes can be controlled by the QUADRATURE_MODE[21:20] field of the counter control register (TCPWM_CNT_CTRL). This mode uses double buffered capture registers.

The Quadrature mode operation occurs as follows:

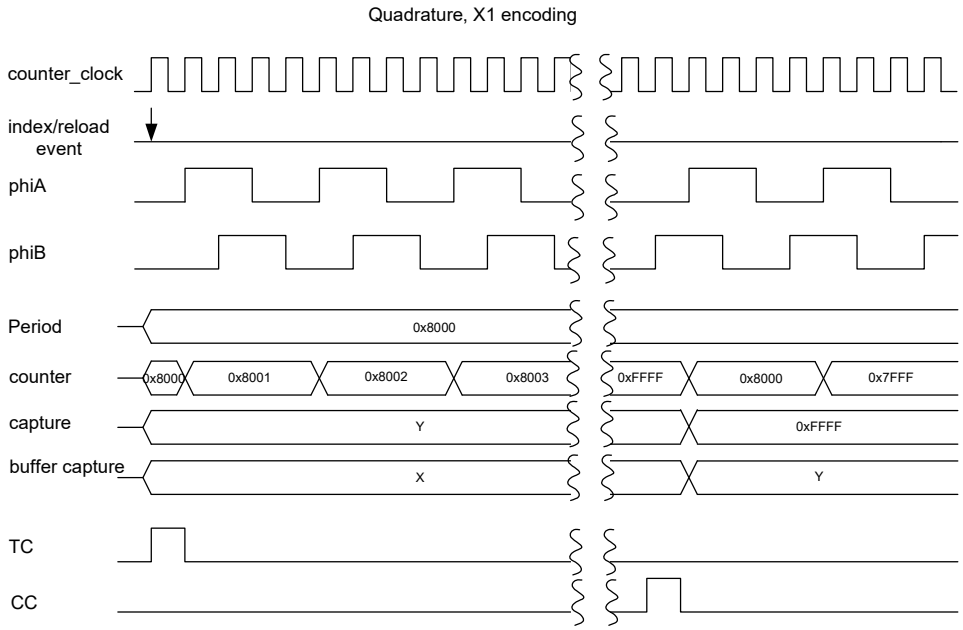
- Quadrature phases phiA and phiB: Counting direction is determined by the phase relationship between phiA and phiB. These phases are connected to the count and the start trigger inputs, respectively as hardware input to the decoder.
- Quadrature index signal: This is connected to the reload signal as a hardware input. This event generates a TC condition, as shown in Figure 15-9.

  On TC, the counter is set to 0x0000 (in the up counting mode) or to the period value (in the down counting mode).

  Note The down counting mode is recommended to be used with a period value of 0x8000 (the mid-point value).
- A pulse on CC output signal is generated when the count register value reaches 0x0000 or 0xFFFF. On a CC condition, the count register is set to the period value (0x8000 in this case).
- On TC or CC condition:
  - Count register value is copied to the capture register
  - Capture register value is copied to the buffer capture register
  - This condition can be used to generate an interrupt request
- The value in the capture register can be used to determine which condition caused the event and whether:
  - A counter underflow occurred (value 0)
  - A counter overflow occurred (value 0xFFFF)
  - An index/TC event occurred (value is not equal to either 0 or 0xFFFF)
- The DOWN bit field of counter status (TCPWM_CNTx_STATUS) register can be read to determine the current counting direction. Value '0' indicates a previous increment operation and value '1' indicates previous decrement operation. Figure 15-9 illustrates quadrature behavior in the X1 encoding mode.
  - A positive edge on phiA increments the counter when phiB is '0' and decrements the counter when phiB is '1'.
  - The count register is initialized with the period value on an index/reload event.
  - Terminal count is generated when the counter is initialized by index event. This event can be used to generate an interrupt.

- When the count register reaches 0xFFFF (the maximum count register value), the count register value is copied to the capture register and the count register is initialized with period value (0x8000).

Figure 15-9. Timing Diagram for Quadrature Mode, X1 Encoding

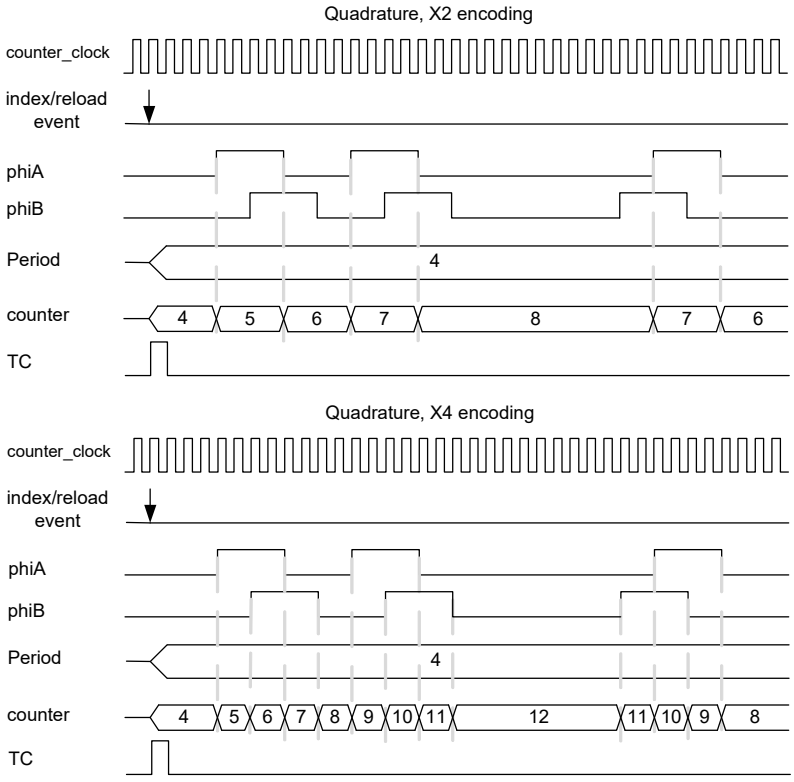


The quadrature phases are detected on the counter_clock. Within a single counter period, the phases should not change value more than once.

The X2 and X4 quadrature encoding modes count twice and four times as fast as the X1 encoding mode.

Figure 15-10 illustrates the quadrature mode behavior in the X2 and X4 encoding modes.

Figure 15-10. Timing Diagram for Quadrature Mode, X2 and X4 Encoding



15.3.3.3 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 16-bit period in the TCPWM_CNT_PERIOD register.
4. Set the required encoding mode by writing to the QUADRATURE_MODE[21:20] field of the TCPWM_CNT_CTRL register.
5. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (Index and Stop).
6. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (Index and Stop).
7. If required, set the interrupt upon TC or CC condition, as shown in Interrupts on page 109.
8. Enable the counter by writing '1' to the COUNTER_ENABLED field of the TCPWM_CTRL register.

15.3.4 Pulse Width Modulation Mode

The PWM mode is also called the Digital Comparator mode. The comparison output is a PWM signal whose period depends on the period register value and duty cycle depends on the compare and period register values.

PWM period = (period value/counter clock frequency) in left- and right-aligned modes

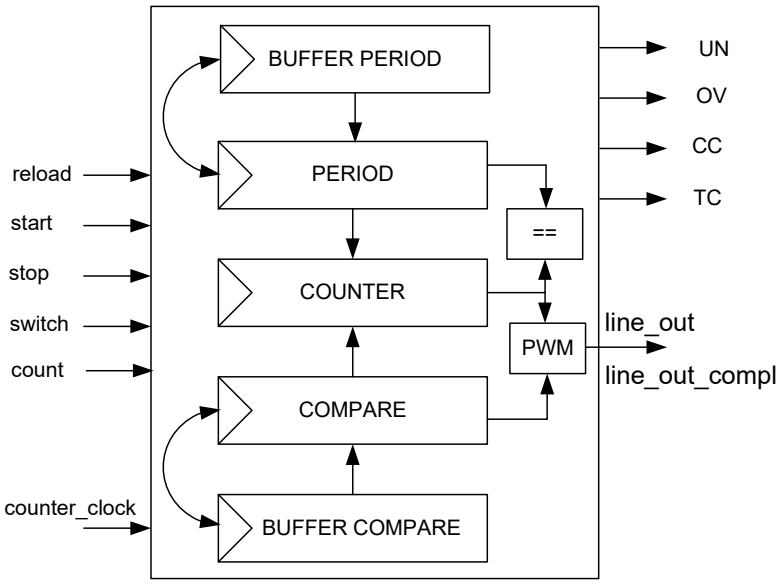
PWM period = (2 × (period value/counter clock frequency)) in center-aligned mode

Duty cycle = (compare value/period value) in left- and right-aligned modes

Duty cycle = ((period value-compare value)/period value) in center-aligned mode

15.3.4.1 Block Diagram

Figure 15-11. PWM Mode Block Diagram



15.3.4.2 How It Works

The PWM mode can output left, right, center, or asymmetrically aligned PWM signals. The desired output alignment is achieved by using the counter's up, down, and up/down counting modes selected using UP_DOWN_MODE [17:16] bits in the TCPWM_CNT_CTRL register, as shown in Table 15-6.

This CC signal along with OV and UN signals control the PWM output line. The signals can toggle the output line or set it to a logic '0' or '1' by configuring the TCPWM_CNT_TR_CTRL2 register. By configuring how the signals impact the output line, the desired PWM output alignment can be obtained.

The recommended way to modify the duty cycle is:

- The buffer period register and buffer compare register are updated with new values.
- On TC, the period and compare registers are automatically updated with the buffer period and buffer compare registers when there is an active switch event. The AUTO_RELOAD_CC and AUTO_RELOAD_PERIOD fields of the counter control register are set to ‘1’. When a switch event is detected, it is remembered until the next TC event. Pass through signal (selected during event detection setting) cannot trigger a switch event.
- Updates to the buffer period register and buffer compare register should be completed before the next TC with an active switch event; otherwise, switching does not reflect the register update, as shown in Figure 15-13.

In the center-aligned mode, the output line is set to '0' at Terminal Count and toggled at the CC condition.

At the reload event, the count register is initialized and starts counting in the appropriate mode. At every count, the count register value is compared with compare register value to generate the CC signal on match.

Figure 15-12 illustrates center-aligned PWM with buffered period and compare registers (up/down counting mode 0).

Figure 15-12. Timing Diagram for Center Aligned PWM

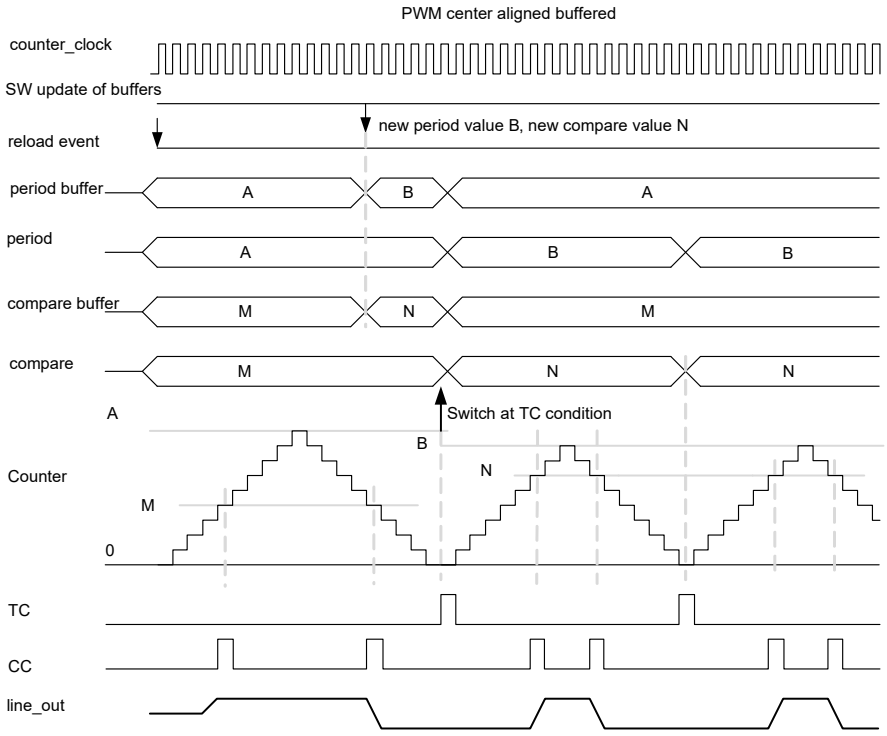
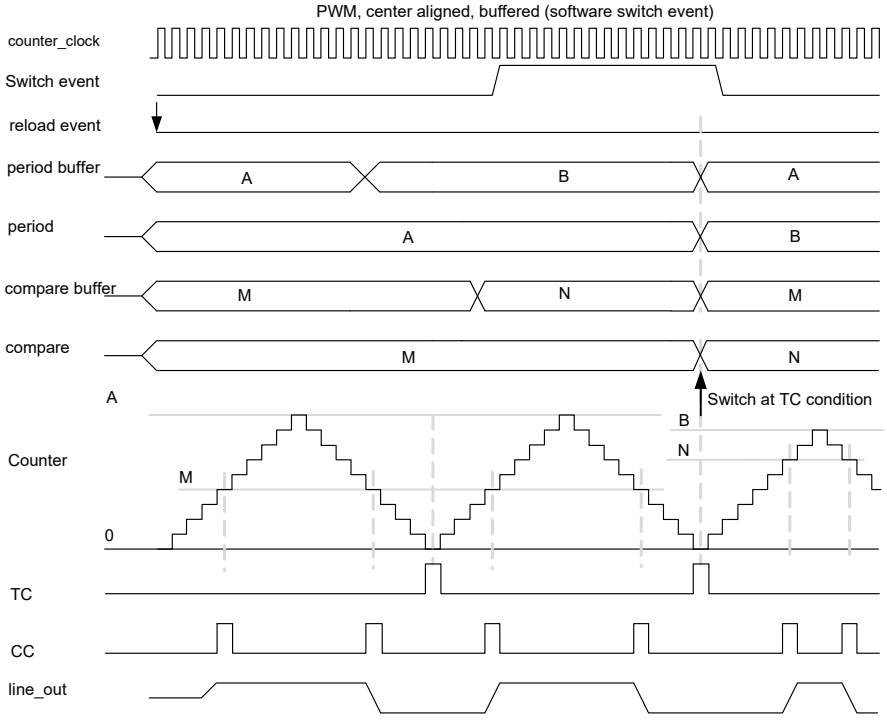


Figure 15-13 illustrates center-aligned PWM with software generated switch events:

- Software generates a switch event only after both the period buffer and compare buffer registers are updated.
- Because the updates of the second PWM pulse come late (after the terminal count), the first PWM pulse is repeated.
- Note that the switch event is automatically cleared by hardware at TC after the event takes effect.

Figure 15-13. Timing Diagram for Center Aligned PWM (software switch event)



15.3.4.3 Other Configurations

- For asymmetric PWM, the up/down counting mode 1 should be used. This causes a TC when the counter reaches either ‘0’ or the period value. To create an asymmetric PWM, the compare register is changed at every TC (when the counter reaches either ‘0’ or the period value), whereas the period register is only changed at every other TC (only when the counter reaches ‘0’).
- For left-aligned PWM, use the up counting mode; configure the OV condition to set output line to '1' and CC condition to reset the output line to '0'. See Table 15-3.
- For right-aligned PWM, use the down counting mode; configure UN condition to reset output line to '0' and CC condition to set the output line to '1'. See Table 15-3.

15.3.4.4 Kill Feature

Kill feature gives the ability to disable both output lines immediately. This event can be programmed to stop the counter by modifying the PWM_STOP_ON_KILL and PWM_SYNC_KILL fields of the counter control register, as shown in Table 15-7.

Table 15-7. Field Setting for Stop on Kill Feature

PWM_STOP_ON_KILL Field	Comments
0	The kill trigger temporarily blocks the PWM output line but the counter is still running.
1	The kill trigger temporarily blocks the PWM output line and the counter is also stopped.

A kill event can be programmed to be asynchronous or synchronous, as shown in Table 15-8.

Table 15-8. Field Setting for Synchronous/Asynchronous Kill

PWM_SYNC_KILL Field	Comments
0	An asynchronous kill event lasts as long as it is present. This event requires pass through mode.
1	A synchronous kill event disables the output lines until the next TC event. This event requires rising edge mode.

In the synchronous kill, PWM cannot be started before the next TC. To restart the PWM immediately after kill input is removed, kill event should be asynchronous (see Table 15-8). The generated stop event disables both output lines. In this case, the reload event can use the same trigger input signal but should be used in falling edge detection mode.

15.3.4.5 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.

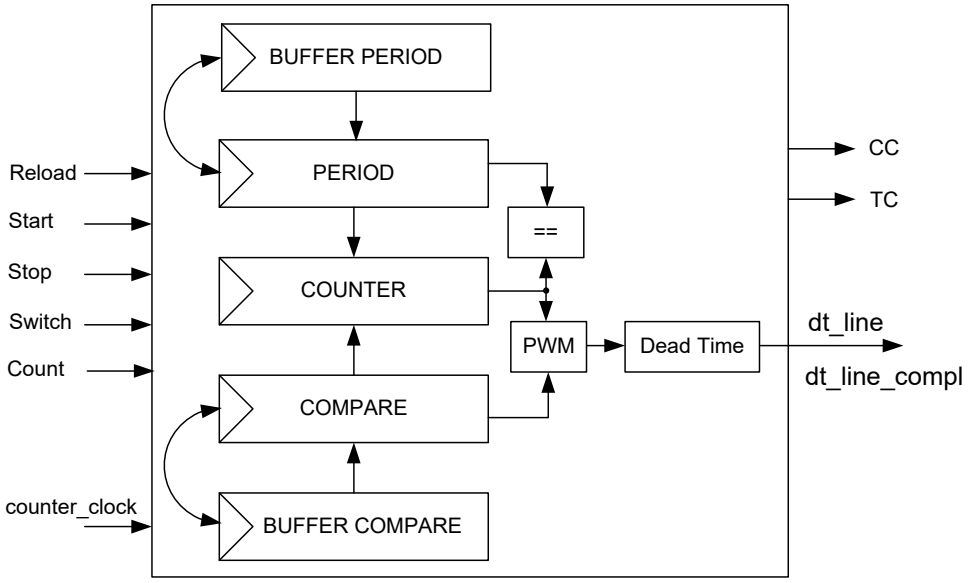
1. Disable the counter by writing '0' to the COUNTER_ENABLED field of the TCPWM_CTRL register.
2. Select PWM mode by writing '100' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set clock prescaling by writing to the GENERIC[10:8] field of the TCPWM_CNT_CTRL register, as shown in Table 15-1.
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 switch 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 switch 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, as shown in Table 15-6.
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, Switch, and Count).
9. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (Reload, Start, Kill, Switch, and Count).
10. line_out and line _out_compl 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, as shown in Interrupts on page 109.
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.

15.3.5 Pulse Width Modulation with Dead Time Mode

Dead time is used to delay the transitions of both ‘line_out’ and ‘line_out_compl’ signals. It separates the transition edges of these two signals by a specified time interval. Two complementary output lines 'dt_line' and 'dt_line_compl' are derived from these two lines. During the dead band period, both compare output and complement compare output are at logic ‘0’ for a fixed period. The dead band feature allows the generation of two non-overlapping PWM pulses. A maximum dead time of 255 clocks can be generated using this feature.

15.3.5.1 Block Diagram

Figure 15-14. PWM-DT Mode Block Diagram



15.3.5.2 How It Works

The PWM operation with Dead Time mode occurs as follows:

- On the rising edge of the PWM line_out, depending upon UN, OV, and CC conditions, the dead time block sets the dt_line and dt_line_compl to '0'.
- The dead band period is loaded and counted for the period configured in the register.
- When the dead band period is complete, dt_line is set to '1'.
- On the falling edge of the PWM line_out depending upon UN, OV, and CC conditions, the dead time block sets the dt_line and dt_line_compl to '0'.
- The dead band period is loaded and counted for the period configured in the register.
- When the dead band period has completed, dt_line_compl_is set to '1'.
- A dead band period of zero has no effect on the dt_line and is the same as line_out.
- When the duration of the dead time equals or exceeds the width of a pulse, the pulse is removed.

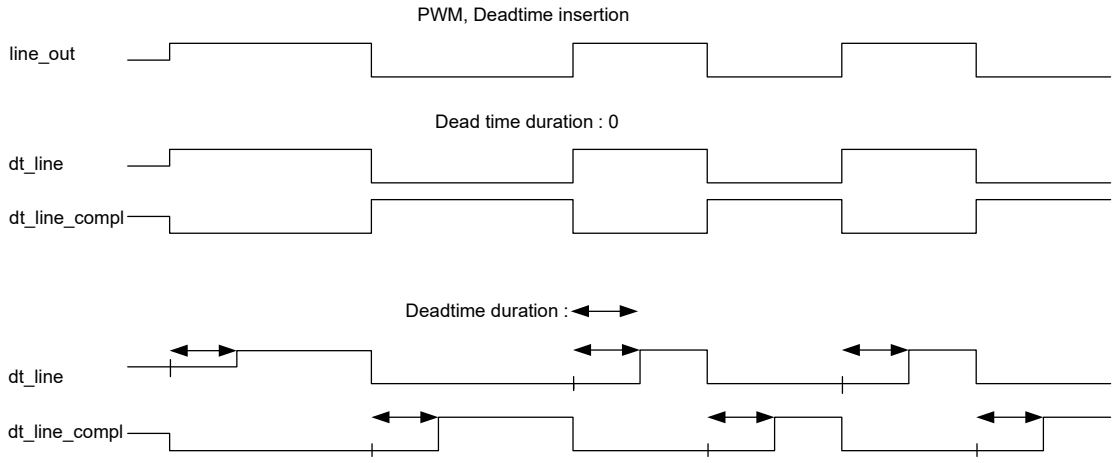
This mode follows PWM mode and supports the following features available with that mode:

- Various output alignment modes
- Two complementary output lines, dt_line and dt_line_compl, derived from PWM "line_out" and "line _out_compl", respectively
  - Stop/kill event with synchronous and asynchronous modes
  - Conditional switch event for compare and buffer compare registers and period and buffer period registers

This mode does not support clock prescaling.

Figure 15-15 illustrates how the complementary output lines "dt_line" and "dt_line_compl" are generated from the PWM output line, "line_out".

Figure 15-15. Timing Diagram for PWM, with and without Dead Time



15.3.5.3 *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:

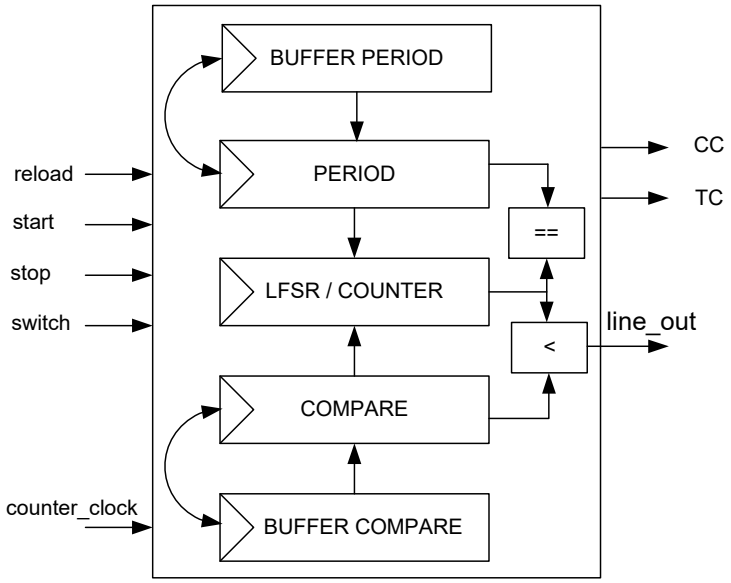
1. Disable the counter by writing '0' to the COUNTER_ENABLED field of the TCPWM_CTRL register.
2. Select PWM with Dead Time mode by writing '101' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required dead time by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register, as shown in Table 15-1.
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 switch 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 switch 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, as shown in Table 15-6.
7. Set the PWM_STOP_ON_KILL and PWM_SYNC_KILL fields of the TCPWM_CNT_CTRL register as required, as shown in the Pulse Width Modulation Mode on page 119.
8. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (Reload, Start, Kill, Switch, and Count).
9. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (Reload, Start, Kill, Switch, and Count).
10. dt_line and dt_line_compl 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, as shown in Interrupts on page 109.
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 hardware start signal is not enabled.

15.3.6 Pulse Width Modulation Pseudo-Random Mode

This mode uses the linear feedback shift register (LFSR). LFSR is a shift register whose input bit is a linear function of its previous state.

15.3.6.1 Block Diagram

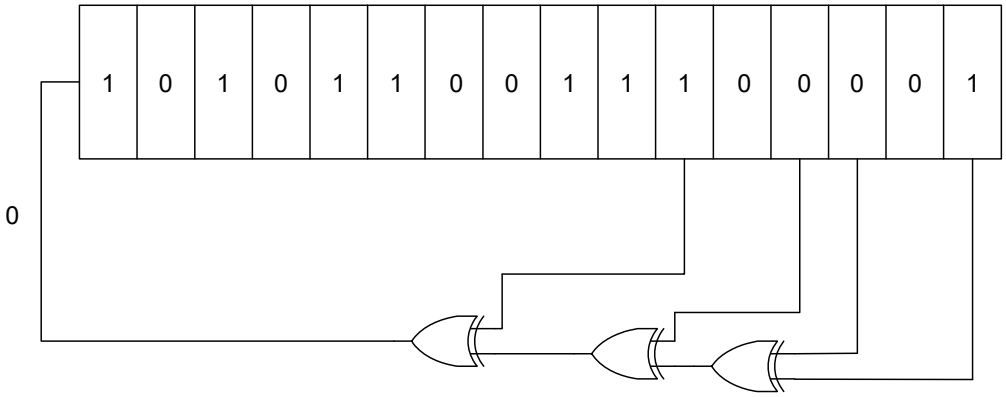
Figure 15-16. PWM-PR Mode Block Diagram



15.3.6.2 How It Works

The counter register is used to implement LFSR with the polynomial: $x^{16}+x^{14}+x^{13}+x^{11}+1$, as shown in Figure 15-17. It generates all the numbers in the range [1, 0xFFFF] in a pseudo-random sequence. Note that the counter register should be initialized with a non-zero value.

Figure 15-17. Pseudo-Random Sequence Generation using Counter Register

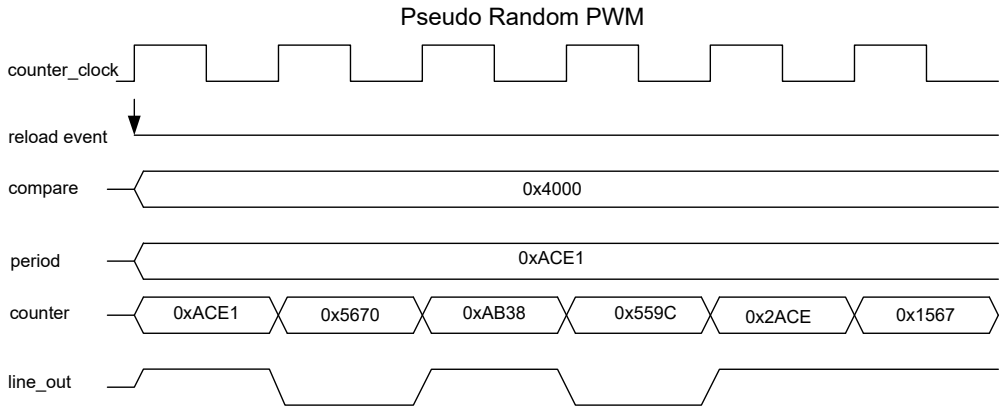


The following steps describe the process:

- The PWM output line ‘line_out’ is driven with '1' when the lower 15-bit value of the counter register is smaller than the value in the compare register (when counter[14:0] < compare[15:0]). A compare value of ‘0x8000’ or higher always results in a '1' on the PWM output line. A compare value of ‘0’ always results in a '0' on the PWM output line.
- A reload event behaves similar to a start event; however, it does not initialize the counter.
- Terminal count is generated when the counter value equals the period value. LFSR generates a predictable pattern of counter values for a certain initial value. This predictability can be used to calculate the counter value after a certain amount of LFSR iterations ‘n’. This calculated counter value can be used as a period value and the TC is generated after ‘n’ iterations.
- At TC, a switch/capture event conditionally switches the compare and period register pairs (based on the AUTO_RELOAD_CC and AUTO_RELOAD_PERIOD fields of the counter control register).

- A kill event can be programmed to stop the counter as described in previous sections.
- One shot mode can be configured by setting the ONE_SHOT field of the counter control register. At terminal count, the counter is stopped by hardware.
- In this mode, underflow, overflow, and trigger condition events do not occur.
- CC condition occurs when the counter is running and its value equals compare value. Figure 15-18 illustrates pseudo-random noise behavior.
- A compare value of 0x4000 results in 50 percent duty cycle (only the lower 15 bits of the 16- bit counter are used to compare with the compare register value).

Figure 15-18. Timing Diagram for Pseudo-Random PWM



A capture/switch input signal may switch the values between the compare and compare buffer registers and the period and period buffer registers. This functionality can be used to modulate between two different compare values using a trigger input signal to control the modulation.

Note Capture/switch input signal can only be triggered by an edge (rising, falling, or both). This input signal is remembered until the next terminal count.

15.3.6.3 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 switch 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 switch 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 Switch).
7. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (Reload, Start, Kill, and Switch).
8. line_out and line_out_compl 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, as shown in Interrupts on page 109.
10. Enable the counter by writing '1' to the COUNTER_ENABLED field of the TCPWM_CTRL register.

15.4 TCPWM Registers

Table 15-9. 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_CNTx_CTRL	Counter control register	Configures counter mode, encoding modes, one shot mode, switching, kill feature, dead time, clock prescaling, and counting direction
TCPWM_CNTx_STATUS	Counter status register	Reads the direction of counting, dead time duration, and clock prescaling; checks if counter is running
TCPWM_CNTx_COUNTER	Count register	Contains the 16-bit counter value
TCPWM_CNTx_CC	Counter compare/capture register	Captures the counter value or compares the value with the counter value
TCPWM_CNTx_CC_BUFF	Counter buffered compare/capture register	Buffer register for counter CC register; switches compare value
TCPWM_CNTx_PERIOD	Counter period register	Contains upper value of the counter
TCPWM_CNTx_PERIOD_BUFF	Counter buffered period register	Buffer register for counter period register; switches period value
TCPWM_CNTx_TR_CTRL0	Counter trigger control register 0	Selects trigger for specific counter events
TCPWM_CNTx_TR_CTRL1	Counter trigger control register 1	Determines edge detection for specific counter input signals
TCPWM_CNTx_TR_CTRL2	Counter trigger control register 2	Controls counter output lines upon CC, OV, and UN conditions
TCPWM_CNTx_INTR	Interrupt request register	Sets the register bit when TC or CC condition is detected
TCPWM_CNTx_INTR_SET	Interrupt set request register	Sets the corresponding bits in the interrupt request register
TCPWM_CNTx_INTR_MASK	Interrupt mask register	Mask for interrupt request register
TCPWM_CNTx_INTR_MASKED	Interrupt masked request register	Bitwise AND of interrupt request and mask registers

16. Cryptography Block

The Cryptography block in EZ-PD™ PMG1-S2 MCU provides up to five cryptographic functionalities including:

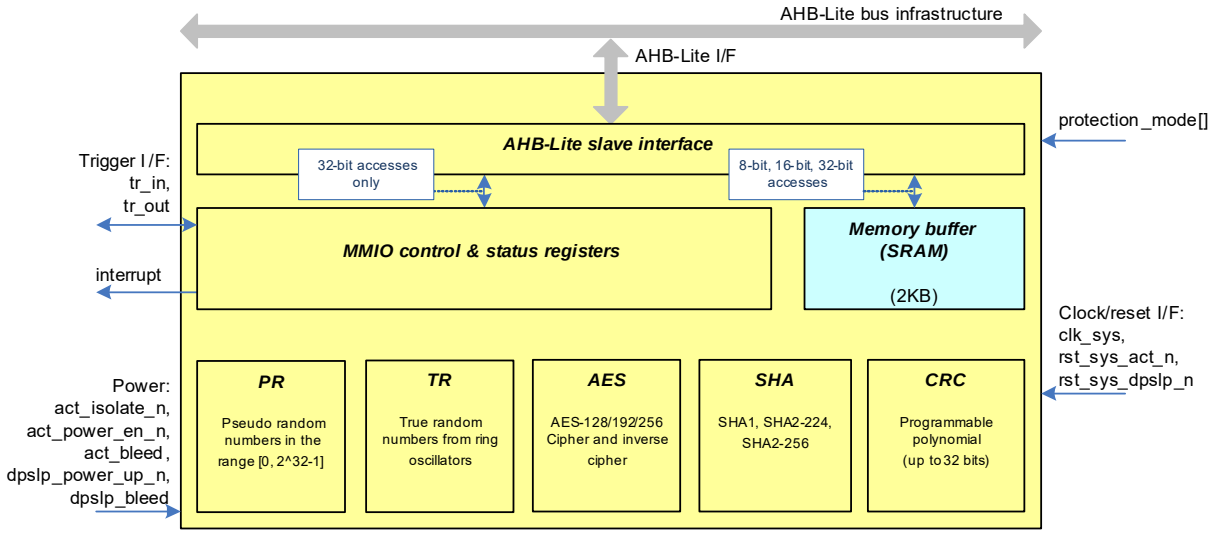
- Advanced Encryption Standard (AES) functionality: The AES component can be used to encrypt/decrypt data blocks of 128-bit length and it supports programmable key length (128/192/256-bit key).
- Secure Hash Algorithm (SHA) functionality: This component can be used to produce a fixed length hash (also called message digest) from a variable length input data (called message).
- Cyclic Redundancy Check (CRC) functionality: This component performs a cyclic redundancy check with a programmable polynomial of up to 32-bits.
- Pseudo Random Number Generator (PR): This component generates pseudo random numbers in a fixed range. This generator is based on three Linear Feedback Shift Registers (LFSRs).
- True Random Number Generator (TR): This component generates true random numbers using up six sets of ring oscillators.

16.1 Features

- AES functionality supports both forward block cipher and inverse block cipher
- Programmable key length (128/192/256-bit key) for AES
- SHA functionality supports 160-bit hash or 224-bit hash or 256-bit hash
- Pseudo random (PR) number generator
- True random number generator (TR)
- Polynomial for CRC functionality is programmable up to 32-bits
- 2-KB SRAM memory buffer to store operands and results of all operations
- Trigger interface to support hardware initiation of an operation

16.2 Block Diagram

Figure 16-1. Crypto Block Diagram



The Cryptography block functionality includes:

- AES functionality (block cipher), per FIPS 197 standard
  - Forward block cipher (plaintext to ciphertext) with 128/192/256-bit key
  - Inverse block cipher (ciphertext to plaintext) with 128/192/256-bit key
  - SHA functionality (hash), per FIPS 180-4 standard
  - SHA1, 160-bit hash
  - SHA2, 224-bit hash
  - SHA2, 256-bit hash (not supported: 384-bit and 512-bit hash)
- CRC functionality
  - Programmable polynomial of up to 32-bits
- Pseudo Random (PR) number generator
- True Random (TR) number generator

Note: Only one of the functions can be used at any given time.

This block is connected to the bus infrastructure as an AHB-Lite slave: it can receive AHB-Lite transfers (from the CPU or another bus master) but it cannot initiate AHB-Lite transfers.

16.2.1 Block Application Overview

The AES component can be used to encrypt plain text data or decrypt cipher data.

The SHA component can be used to generate the signature (or message digest) for the data block to ensure integrity and authentication of the data.

The PR component will be used to obtain pseudo random values, which can be used as initialization vector (IV) for AES encryption and for other purposes.

The CRC is an optional component and can be used to generate CRC of the given data, if the application requires it.

This block has the following interfaces:

- An AHB-Lite slave interface connects the block to the AHB-Lite infrastructure. This interface supports 8/16/32-bit AHB-Lite transfers. MMIO register accesses are 32-bit accesses only (8/16-bit accesses to MMIO registers results in an AHB-Lite bus error). Memory buffer access can be 8/16/32-bit accesses.
- Protection mode (protection_mode[]) signals from the CPU subsystem are provided to restrict the debug access port (DAP) accesses to the memory buffer.

- A trigger interface allows a block operation to be started by an external trigger (tr_in). When the operation completes, the block generates a trigger (tr_out).
- A single interrupt signal (interrupt) is used to signal the completion of an operation.
- A clock and reset signal interface connects to the system resources subsystem (SRSS). The block operates a gated version of clk_sys and uses both Active (rst_sys_act_n) and Deep Sleep (rst_sys_dpslp_n) reset signals.

16.2.2 Sub Block Descriptions

16.2.2.1 Pseudo Random Number Generator

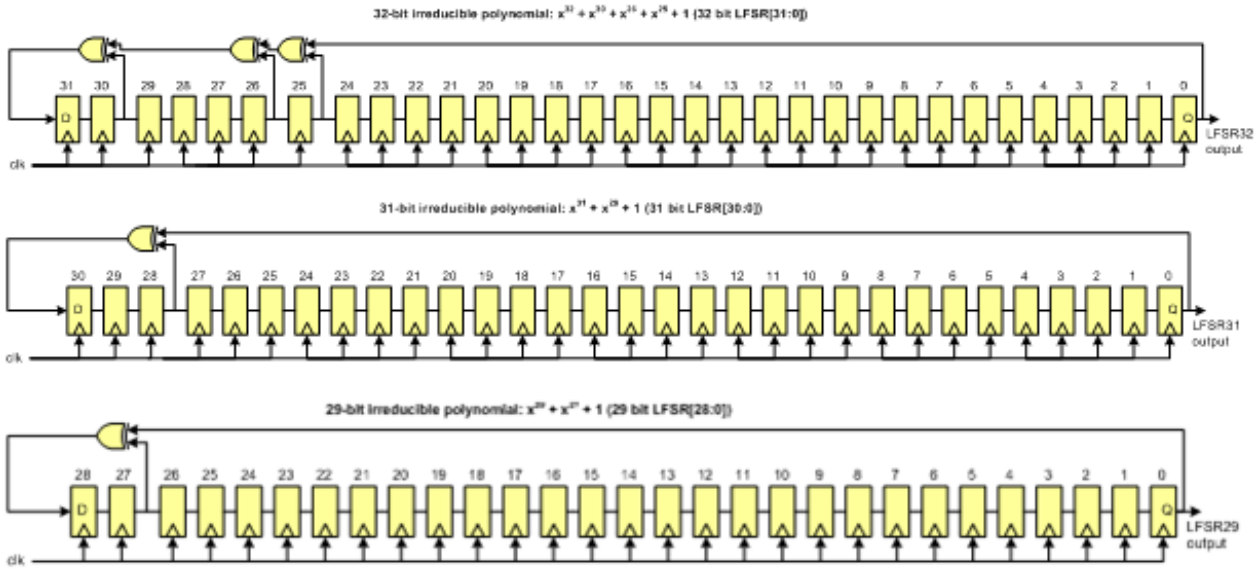
The PRNG component generates pseudo random numbers in a fixed range [0, PR_CTL.MAX[31:0]]. The generator is based on three Fibonacci-based Linear Feedback Shift Registers (LFSRs).

The following three polynomials are implemented:

- 32-bit polynomial: x32 + x30 + x26 + x25 + 1
- 31-bit polynomial: x31 + x28 + 1
- 29-bit polynomial: x29 + x27 + 1

Figure 16-2 illustrates the LFSR functionality.

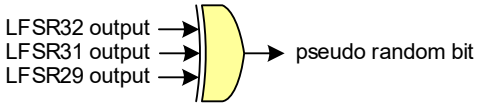
Figure 16-2. Fixed Fibonacci-based LFSRs



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

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

Figure 16-3. XOR Reduction Logic



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

A pseudo random number of n bits "pr[n-1:0]" uses 'n' pseudo random bits from the pseudo random number generator. The pseudo random number generator component uses a total of 33 pseudo random bits to generate a result in the range [0, PR_CTL.MAX[31:0]].

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

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

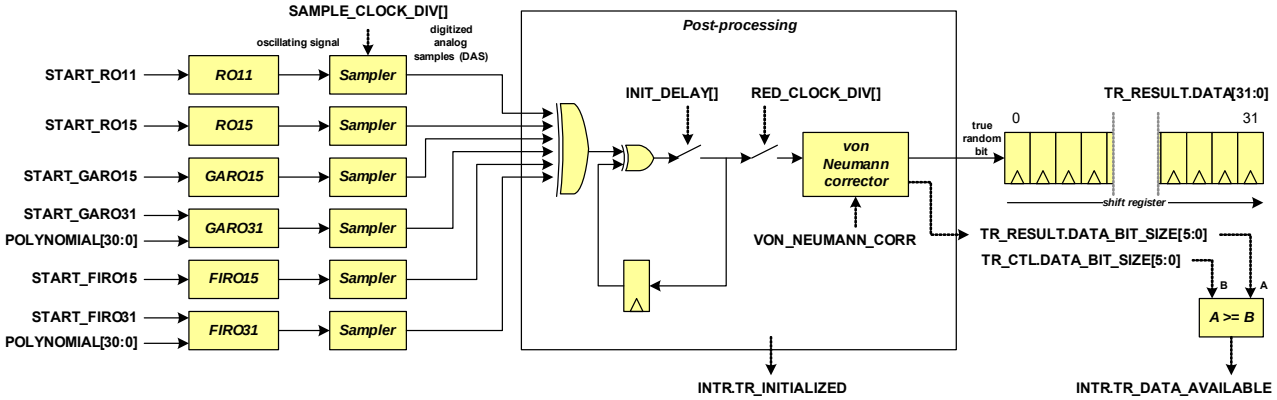
The result is provided through the MMIO register PR_RESULT.

16.2.2.2 True Random Number Generator

The TRNG component generates true random numbers. The bit size of these generated numbers is programmable in the range [0, 32]. The TRNG relies on up to six ring oscillators to provide physical noise sources. A ring oscillator consists of a series of inverters connected in a feedback loop to form a ring.

Post-processing produces bit samples that are considered true random bit samples. The true random bit samples are shifted into a register, to provide random values of up to 32 bits. Figure 16-4 gives an overview of the TRNG component.

Figure 16-4. TRNG Overview

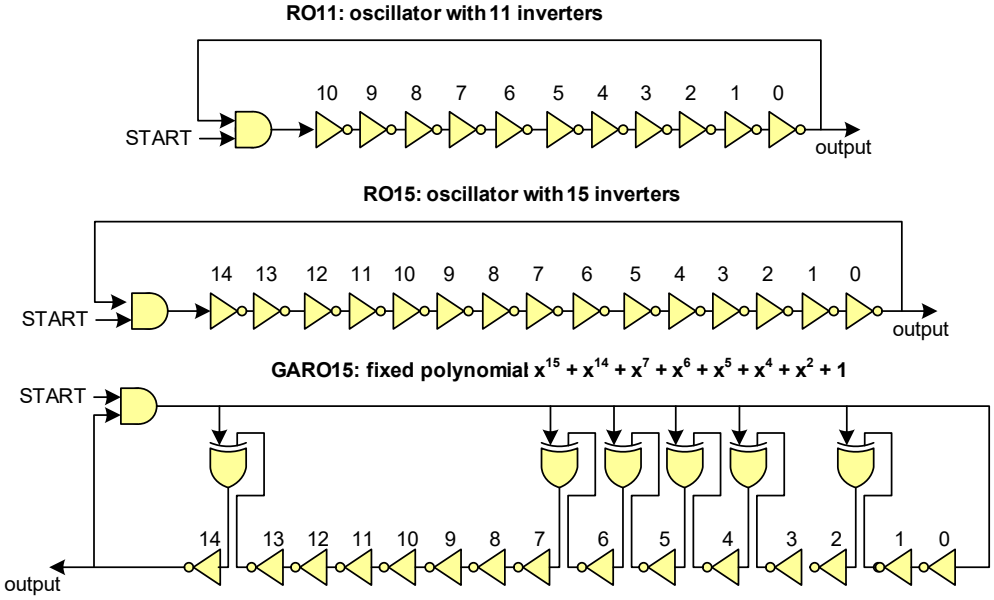


The following are the ring oscillators on which the TRNG relies:

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

Each ring oscillator can be started or stopped. When stopped, the ring is "broken" to prevent switching. The following figures illustrate the schematics of the fixed ring oscillators.

Figure 16-5. Fixed Ring Oscillators: RO11, RO15, GARO15, and FIRO15

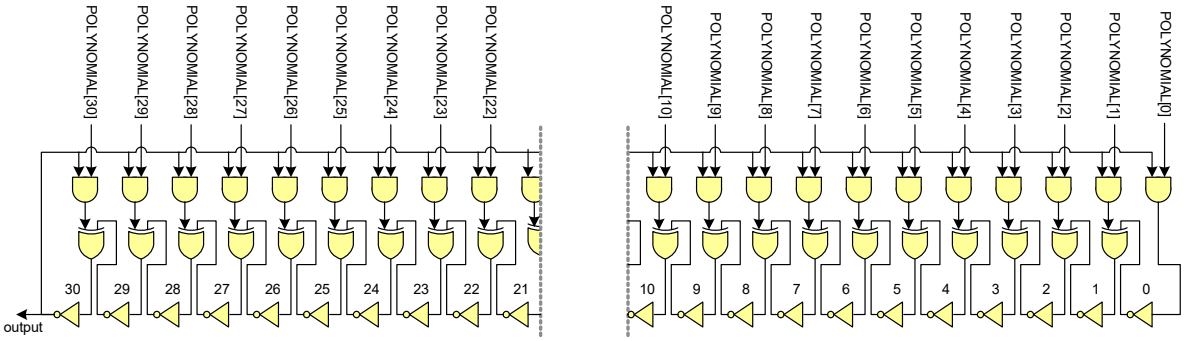


The START signals originate from a MMIO register field.

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

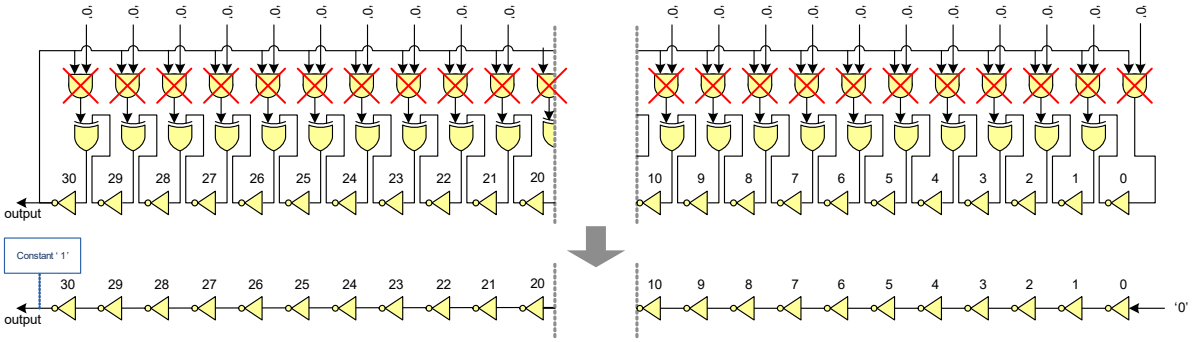
Figure 16-6 gives an overview of the Galois-based ring oscillator.

Figure 16-6. Flexible Galois Ring Oscillator GARO31



When the ring oscillator is stopped, the polynomial is forced to “0” and the ring is broken as illustrated by Figure 16-7.

Figure 16-7. Flexible GARO31 when not Running



The programmable polynomial specifies the oscillator feedback. Figure 16-8 illustrates two examples.

Figure 16-8. Flexible GARO31 Examples

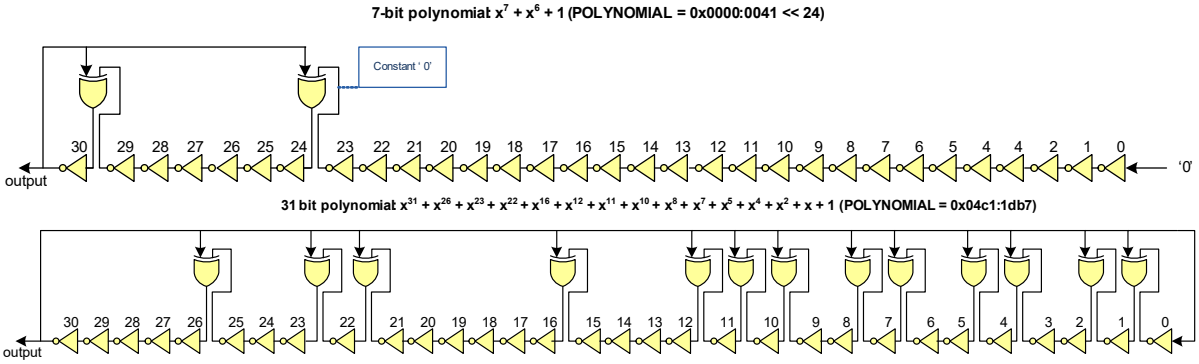
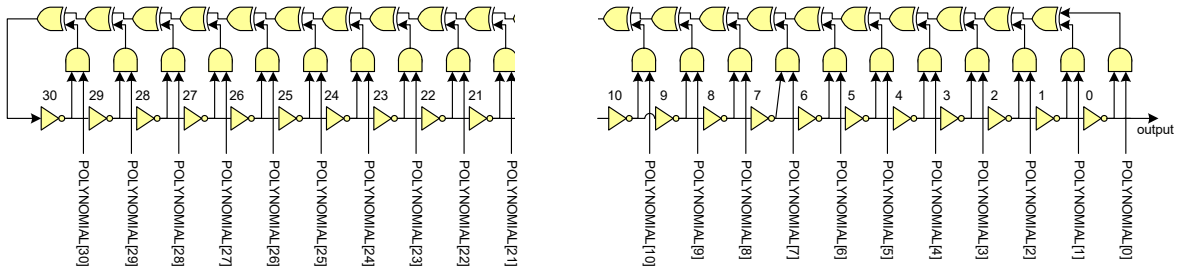


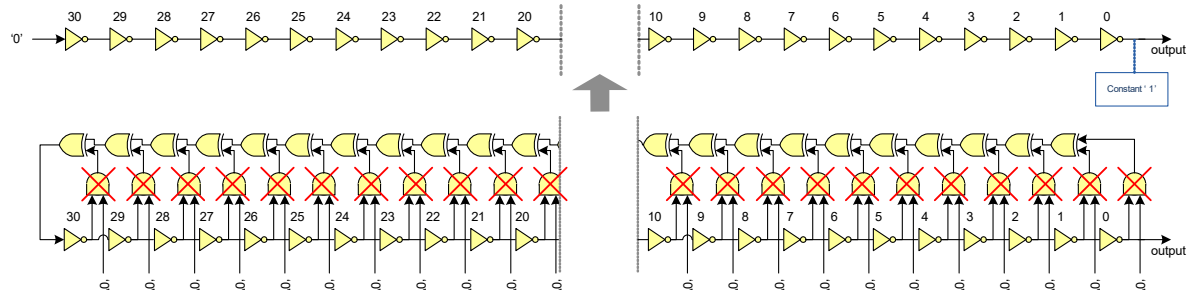
Figure 16-9 gives an overview of the Fibonacci-based ring oscillator.

Figure 16-9. Flexible Fibonacci Ring Oscillator FIRO31



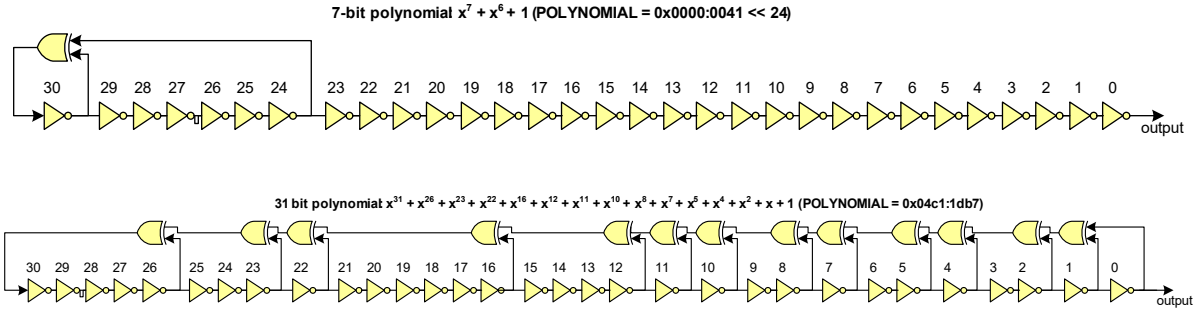
When the ring oscillator is stopped, the polynomial is forced to “0” and the ring is broken as illustrated by Figure 16-10.

Figure 16-10. Flexible FIRO31 when not Running



The programmable polynomial specifies the oscillator feedback. Figure 16-11 illustrates two examples.

Figure 16-11. Flexible FIRO31 Examples



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

- DAS bitstream. This is XOR of the digitized analog samples.

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

The health monitor performs two different tests:

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

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

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

A detection stops the TRNG functionality (all TR_CMD START fields are set to '0') if TR_CTL.STOP_ON_RC_DETECT is set to '1'.

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

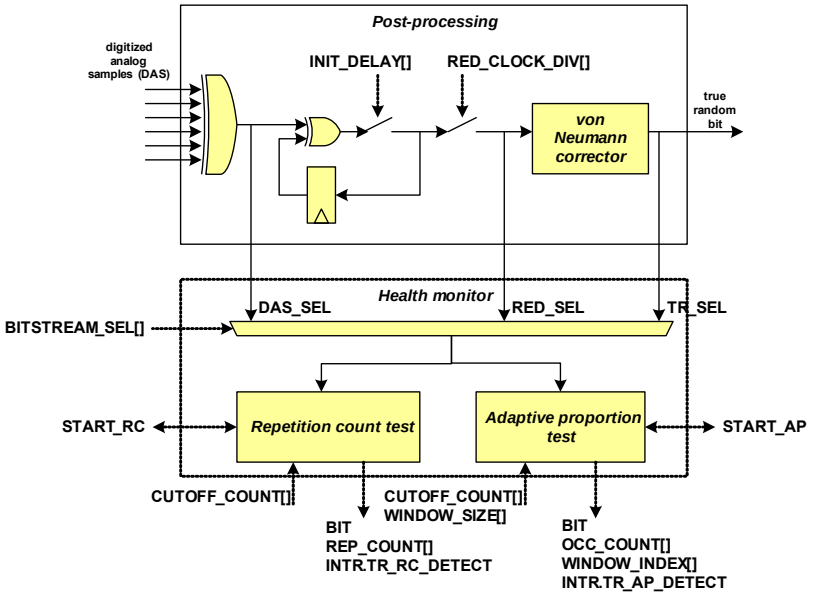
If the test is started (specified by START_AP field), the bitstream is partitioned in bit sequences of a specific window size. At the first bit of a bit sequence, the active bit value BIT is set to the first bit value, the counter WINDOW_INDEX is set to ‘0’ and the counter OCC_COUNT is set to ‘1’. For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

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

A detection stops the TRNG functionality (all TR_CMD START fields are set to '0') if TR_CTL.STOP_ON_AP_DETECT is set to '1'.

Figure 16-12 illustrates the health monitor functionality.

Figure 16-12. TRNG Health Monitor Overview



16.2.2.3 AES

The AES component performs a block cipher or inverse block cipher per the AES standard (FIPS 197).

- The block cipher translates a 128 bit block of plaintext data into a 128 bit block of ciphertext data.
- The inverse block cipher translates a 128 bit block of ciphertext data into a 128 bit block of plaintext data

AES is a symmetric block cipher: the cipher and inverse cipher keys are the same. The component supports 128 bit, 192 bit, and 256 bit keys. The AES algorithm generates so called round keys "on the fly" from the (start) round key as provided by software. Round key generation is reversed for the cipher and inverse cipher.

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

The inverse block cipher uses the round key of the final cipher round as the (start) round key for the first inverse cipher round. The round key for the second inverse cipher round is derived from the round key of the first inverse cipher round, and so forth.

The round key of the final inverse cipher round is the same as the symmetric key.

The generated round keys are only dependent on the start key of the first round (in the block cipher this is the symmetric key). It is possible to derive the start round key for the inverse block cipher from the symmetric key, by performing a (forward) block cipher with arbitrary plaintext data.

The number of cipher (or inverse cipher) rounds depends on the key size. Table 16-1 gives the number of rounds.

Table 16-1. AES Key vs Rounds

AES Key Size	Cipher (or inverse cipher) Rounds
128-bit	10 rounds
192-bit	12 rounds
256-bit	14 rounds

All AES operand data is provided by the memory buffer. For example, an AES block cipher operation has two source operands:

- SRC_CTL0 specifies the offset of the start key (symmetric key).
- SRC_CTL1 specifies the offset of the plaintext.

An AES block cipher operation has two destination operands:

- DST_CTL0 specifies the offset of the last round key.
- DST_CTL1 specifies the offset of the ciphertext.

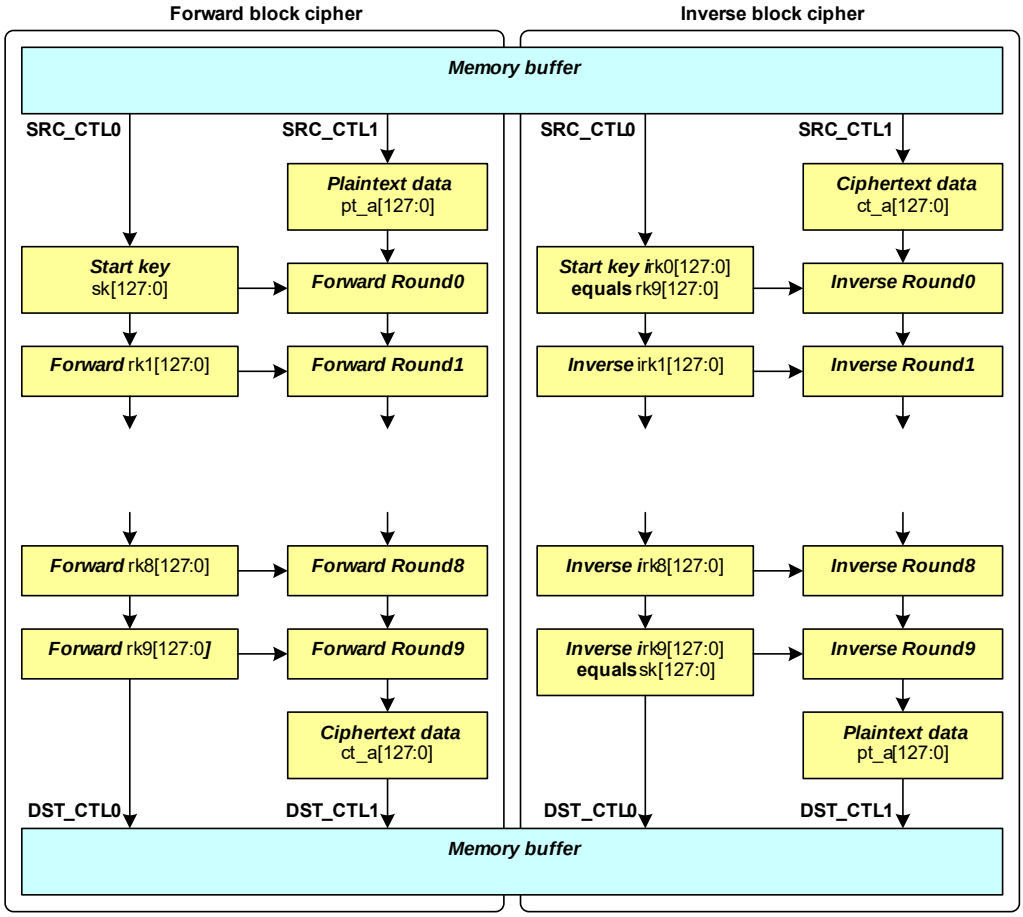
Note: It is required that both source and destination keys and both data buffers are fully (as in all bytes[1]) in the same memory type. For example if Start Key (SRC_CTRL0) is privileged then the Round Key (DST_CTL0) must be in privileged too (to avoid revealing the privileged key). The same must hold for the plaintext (SRC_CTL1) and ciphertext (DST_CTL1). If this is not the case then the algorithm is aborted and the ACCESS_ERROR interrupt is generated.

Note: To reduce space needed in the buffer it is allowed to overwrite the 'start key' with the 'last round key' and similarly it is also allowed to overwrite the input data with the output data.

Figure 16-13 illustrates the AES-128 cipher functionality. It illustrates how a forward block cipher translates a plaintext "pt_a[127:0]" into ciphertext "ct_a[127:0]" using a 128-bit start key "sk[127:0]". Besides the plaintext, the component produces the round key of the last cipher round "rk9[127:0]". If the round key of the last cipher round "rk9[127:0]" is used as the start key for an inverse block cipher on ciphertext "ct_a[127:0]", the original plaintext "pt_a[127:0]" is produced. Furthermore, the round key of the last inverse cipher round is the same as the forward block cipher key "sk[127:0]".

1. Take into account buffer wraparound.

Figure 16-13. AES Operation Flow



16.2.2.4 SHA

The SHA component performs a Secure Hash Algorithm (SHA) according to the SHA standard (FIPS 180-4). The SHA algorithm calculates a fixed length hash value from a variable length message. The hash value is used to produce a message digest or signature.

It is computationally impossible to change the message without changing the signature.

The method is stateless: a given message always produces the same hash value. To prevent "replay attacks", a counter may be included in the message.

The SHA component supports a subset of the algorithms in the SHA standard: SHA1, SHA2-224, and SHA2-256.

Table 16-2. SHA Standard Hash Sizes

Algorithm	Block Size	Word Size	Hash Value Size	Message Digest Size
SHA1	512 bits	32 bits	160 bits	160 bits
SHA2-224	512 bits	32 bits	256 bits	224 bits
SHA2-256	512 bits	32 bits	256 bits	256 bits

The memory buffer provides a message block on which the hash function is performed (SRC_CTL0) and the initial hash value (SRC_CTL1). The message block must be laid out in little endian format.

The memory buffer provides working space for message schedule round constants (DST_CTL0). The hardware derives these constants from the message. The round constants working space size depends on the algorithm.

Table 16-3. SHA SRAM Working Space Sizes

Algorithm	Working Space For Round Constants
SHA1	320 Bytes
SHA2-224	256 Bytes
SHA2-256	256 Bytes

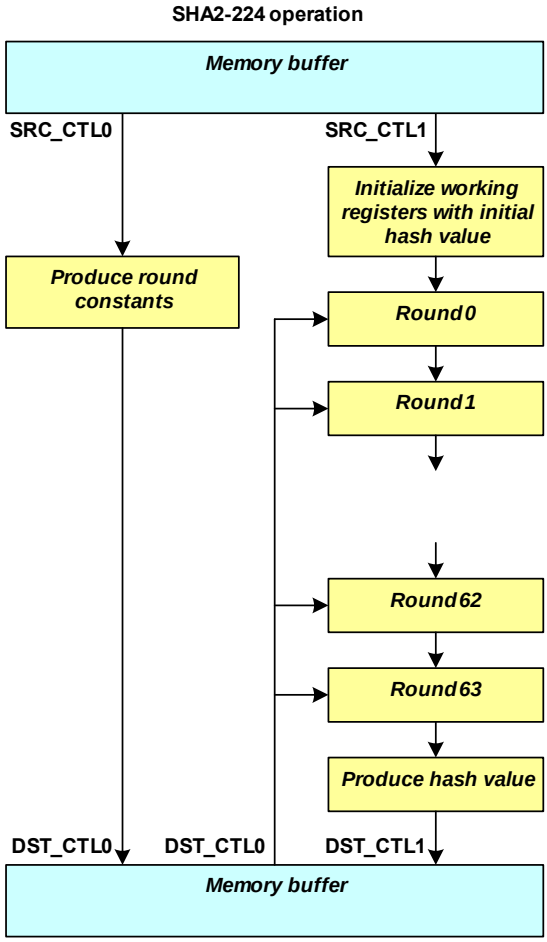
The memory buffer is used for the produced hash value (DST_CTL1).

The message must be preprocessed: a '1' bit must be appended to the message followed by '0's and a 64-bit field. The pre-processed message consists of multiple 512-bit blocks. The SHA component processes a single 512-bit block at a time. The first SHA operation on the first message block uses the initial SHA hash value (as defined by the standard); subsequent SHA operations on successive blocks use the produced hash value of the previous SHA operation.

The SHA operation on the last message block produces the final hash value. The message digest is a subset of the final hash value (SHA2-224) or the complete final hash value (SHA1 and SHA2-256).

Figure 16-14 illustrates the SHA2-224 operation.

Figure 16-14. SHA Operation Flow



Note that it is a requirement that the memory locations used by the SHA operation are all of the same memory type – either all user or all privileged. If this is not the case then the algorithm is aborted and the ACCESS_ERROR interrupt is generated. This restriction is required to prevent user code from using the SHA operation to reveal information on data in privileged locations.

Using the SHA operation on privileged data will require a new SystemCall API.

16.2.2.5 CRC

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

The memory buffer provides the data on which the CRC is performed (SRC_CTL0 specifies the offset of the data in the buffer). The data must be laid out in little endian format (least significant byte of a multi-byte word should be located at the lowest memory address of the word). The MMIO register field CRC_DATA_CTL.DATA_SIZE[10:0] specifies the byte size of the data. The MMIO register field CRC_DATA_CTL.DATA_XOR[7:0] specifies a byte pattern with which each data byte is XOR'd. This allows for inversion of the data byte value. The MMIO register field CRC_DATA_CTL.DATA_REVERSE allows for bit reversal of the data byte (this provides support for serial interfaces that transfer bytes in most-significant-bit first and least-significant bit first configurations).

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

CRC32: x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1

CRC_POL_CTL.POLYNOMIAL[31:0] = 0x04c11db7

CRC16-CCITT: x16 + x12 + x5 + 1

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

CRC16: x16 + x15 + x2 + 1

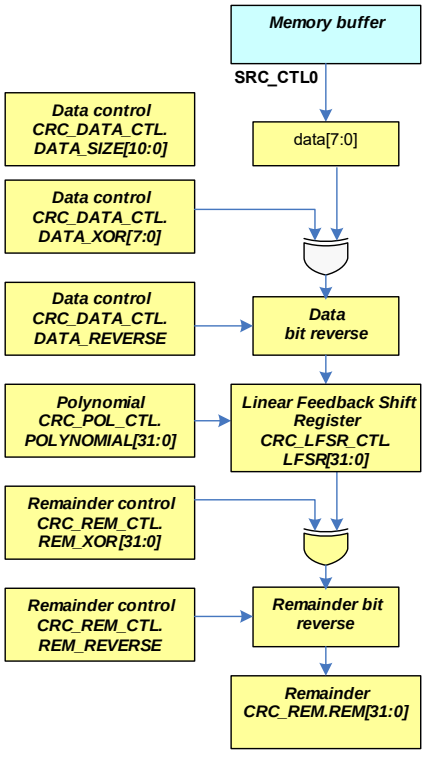
CRC_POL_CTL.POLYNOMIAL[31:0] = (0x8005 << 16)

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

The MMIO register field CRC_REM.REM[31:0] holds the result of the CRC calculation, and is derived from the end state of the CRC calculation (CRC_LFSR_CTL.LFSR[31:0]). The MMIO register field CRC_REM_CTL.REM_XOR[31:0] specifies a 32-bit pattern with which the end state is XOR'd. The MMIO register field CRC_REM_CTL.REM_REVERSE allows for bit reversal of the XOR'd state.

Figure 16-15 illustrates the CRC functionality.

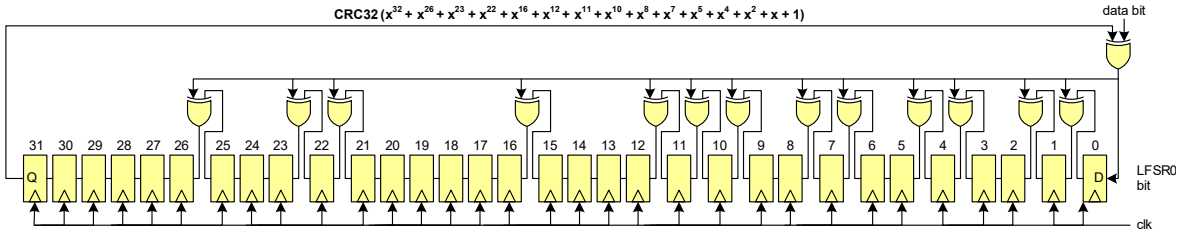
Figure 16-15. CRC Operation Flow



Note that the data must be in the user part of the buffer; if the data is in privileged locations then the algorithm is aborted and the ACCESS_ERROR interrupt is generated. The restriction is required to prevent user code from using the CRC operation to read privileged memory locations.

The Linear Feedback Shift Register functionality operates on the LFSR state. It uses the programmed polynomial and consumes a data bit for each iteration (eight iterations are performed per cycle to provide a throughput of one data byte per cycle). Figure 16-16 illustrates the functionality for the CRC32 polynomial (x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1).

Figure 16-16. CRC32 Polynomial



Different CRC algorithms require different seed values and have different requirements for the XOR functionality and bit reversal. Table 16-4 provides the proper settings for the CRC32, CRC16-CCITT and CRC16 algorithms.

Table 16-4. Standard CRC Settings and Results

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

Table 16-4 also provides the remainder after the algorithm is performed on a five-byte array {0x12, 0x34, 0x56, 0x78, 0x9a}.

16.2.2.6 Trigger Interface

The Crypto block operation is under software control (MMIO CMD register) or under hardware control (trigger interface). In both cases, the MMIO CTL.OPCODE register field specifies which operation needs to be performed.

In a hardware-controlled operation, the rising edge of the input trigger "tr_in" starts an operation. When the operation completes, a two cycle high/'1' pulse (on clk_sys) is generated on the output trigger "tr_out" to indicate operation completion.

Note that tr_out is generated for both software-initiated operation and hardware-initiated operation.

16.2.2.7 Memory Buffer

The block uses regular registers to capture its MMIO control and status information and uses SRAMs to capture its operand data. The SRAMs maintain their information in Deep Sleep power mode. In this power mode, the SRAM array is powered and the SRAM periphery is not powered. During Deep Sleep, the functional SRAM inputs and outputs are isolated (the unpowered periphery is isolated from powered logic).

The offsets of operand data in the memory buffer are under software control and as specified by the SRC_CTL0, SRC_CTL1, DST_CTL0 and DST_CTL1 MMIO registers. Offsets are 32-bit offsets in the memory buffer. Operand data "wraps around" in the memory buffer. Operand data should not overlap in the memory buffer as destination operands may overwrite source operands.

SRC_CTL0 and SRC_CTL1 are used for source operands and DST_CTL0 and DST_CTL1 are used for destination operands. Operand data is dependent on the specific operation. For example, an AES block cipher operation has two source operands:

- SRC_CTL0 specifies the offset of the start key (symmetric key).
- SRC_CTL1 specifies the offset of the plaintext.

An AES block cipher operation has two destination operands:

- DST_CTL0 specifies the offset of the last round key.
- DST_CTL1 specifies the offset of the ciphertext.

Only a single operation can be performed at a time; for example, it is not possible to perform an AES block cipher and SHA hash function simultaneously. However, it is possible to write the source operand data of the next operation in the memory buffer, when the current operation is active. Note that MMIO access to the SRAM memory has higher priority over the operation access to the SRAM memory.

16.2.2.8 Protection

The EZ-PD™ PMG1-S2 MCU chip platform uses a chip protection mode, which restricts DAP access to MMIO registers and memory regions. The block's memory buffer is a memory region and is subject to these restrictions.

Then the chip's platform uses a user/privileged execution mode. When the platform is in user execution mode AHB does not have access to privileged MMIO and memory regions. When the platform is in Privileged execution mode both user and privileged MMIO and memory are freely accessible from AHB. The privilege/user boundary of the memory buffer is set with the PRIV_BUF register, which itself is privileged.

Table 16-5 summarizes these restrictions.

Table 16-5. Protection

Protection Mode	CPU (and DW/DMA)		DAP	
	Memory Buffer Accesses	Mmio Register Accesses	Memory Buffer Accesses	Mmio Register Accesses
VIRGIN	Free access	Free access	Free access	Free access
OPEN	Privileges enforced	Privileges enforced	User mode only	User mode only
PROTECTED	Privileges enforced	Privileges enforced	Inoperable	Inoperable
KILL	Privileges enforced	Privileges enforced	Inoperable	Inoperable
BOOT	Free access	CPU: Free access		
DMA: User mode only	Inoperable	Inoperable		

Note: Blocking DAP access to MMIO registers in PROTECTED, BOOT and KILL modes is not implemented. It implements MMIO register protection through m0s8ahbb_ahbslv_if3 module and this module depends on the fact that the MMIO accesses from DAP are blocked at source (in m0s8cpussv3) in PROTECTED, BOOT and KILL modes.

For the AES block cipher, the memory buffer is used to store cipher and inverse cipher keys. As mentioned, the DAP has no access to the memory buffer in BOOT, PROTECTED and KILL protection modes. As a result, the DAP cannot access keys in the memory buffer. To prevent accesses from "untrusted" user mode CPU code, the chip platform's user/privileged execution mode is used. This is achieved as follows:

- Typically encryption keys can either be on chip secrets stored in privileged flash memory. Or the keys can be ephemeral and stored in user SRAM.
- The privileged flash memory region is only accessible from privileged code. Cypress provides FW functions (SystemCalls) to write key information in privileged flash memory and to copy key information from privileged flash memory to the privileged part of the memory buffer.
- Customers use the Cypress FW function to write key information in privileged flash memory during his manufacturing flow.
- The customer application uses the Cypress FW function to copy key information from privileged flash memory to the privileged region in the memory buffer. As a result, only the privileged Cypress FW function, privileged flash memory, and the block hardware have knowledge of the Secret Key.
- Ephemeral AES keys, typically resulting from authentication communication, are stored in SRAM user locations. The customer application (in user mode) can handle those keys directly.
- Each operation has restrictions on user/privilege locations of keys and data. These restrictions are required to prevent exposing privileged data to user code. If the restriction is violated by the application then the operation is aborted and the ACCESS_ERROR interrupt is set.
- The privileged/user boundary in the PRIV_BUF MMIO register (which is privileged) is typically set during boot with a value from the supervisory rows.
- A new SystemCall allows the boundary to be moved, but as part of this SystemCall the privileged part of the memory buffer will first be cleared.

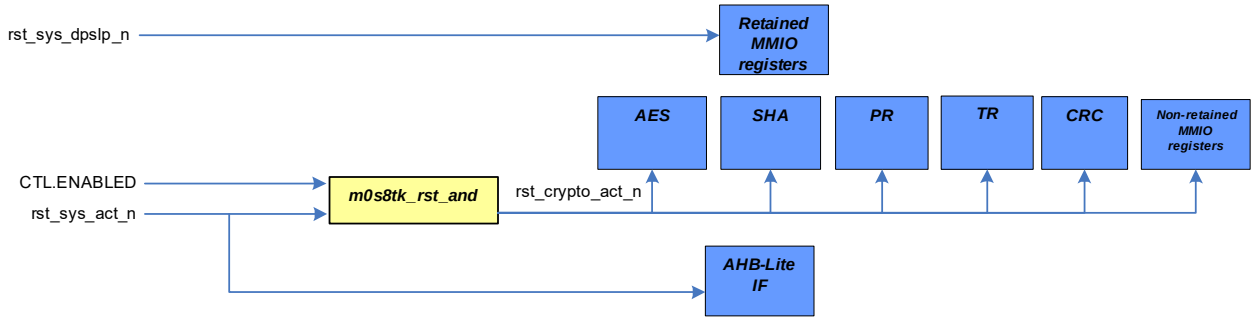
16.3 Block Operation

16.3.1 Reset and Initialization

This block receives two active low asynchronous reset inputs – rst_sys_act_n and rst_sys_dpslp_n.

Figure 16-17 shows the reset connection strategy for this block.

Figure 16-17. Reset Strategy



The rst_sys_dpslp_n is used to reset MMIO registers, which can retain their value during Deep Sleep power mode.

The AHB-Lite slave interface component is connected to rst_sys_act_n reset.

The remaining Active power domain logic (AES, SHA, PR, CRC, and non-retention MMIO registers) is connected to the rst_crypto_act_n reset signal as shown in Figure 16-17. Hence, this logic will be in reset state whenever the block is disabled.

16.3.2 Functional Modes and Usage Requirements

16.3.2.1 Pseudo Random Number Generator

Pseudo random number generation is under software control. It involves the following steps:

1. Enable the block.
2. Initialize the LFSRs with non-zero seed values.
3. Configure the range [0, PR_CTL.MAX[31:0]].
4. Start a pseudo random number generation operation.
5. Retrieve the results when the operation has completed. Waiting for the operation to be completed can be done by either polling or through an ISR.

The following code sequence illustrates how a pseudo random number in the range [0, 9] is generated.

```
*CRYPTO_CTL          = (1 << 31) | "CRC";// enable block
*CRYPTO_PR_LFSR_CTL0 = 0x12345678;// LFSR seed values
*CRYPTO_PR_LFSR_CTL1 = 0x7264f6a1;
*CRYPTO_PR_LFSR_CTL2 = 0x03456236;
*CRYPTO_PR_CTL       = 9;// results in the range [0, 9]
*CRYPTO_CMD          = (1 << 0);// start operation
while (*CRYPTO_CMD & (1 << 0)) ;// wait for operation to complete
result = *PR_RESULT;
```

16.3.2.2 AES

The AES encryption operation is under software control. It involves the following steps:

1. Enable the block.
2. Initialize the memory buffer with the start key.
3. Initialize the memory buffer with plaintext/ciphertext data.
4. Specify the memory buffer offsets of the source and destination operands.
5. Specify the key size.
6. Start a forward/inverse cipher operation.
7. Retrieve the results from the memory buffer when the operation has completed.

The following code sequence illustrates how a 128-bit forward block cipher operation is performed.

```
uint8_t Key[] = {               // encryption key
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
0x0f};
uint8_t PlainText[] = {         // plain text
 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee,
0xff};
uint8_t CipherText[16];  // cipher text
uint8_t Check[] = {             // expected cipher text
 0x69, 0xc4, 0xe0, 0xd8, 0x6a, 0x7b, 0x04, 0x30, 0xd8, 0xcd, 0xb7, 0x80, 0x70, 0xb4, 0xc5,
0x5a};

*CRYPTO_CTL          = (1 << 31) | "AES cipher"; // enable block
for (i = 0 ; i < 16; i++) *(CRYPTO_MEM_BUFF        + i) = Key[i];
for (i = 0 ; i < 16; i++) *(CRYPTO_MEM_BUFF + 0x10 + i) = PlainText[i];
*CRYPTO_SRC_CTL0     = 0x00;// key offset
*CRYPTO_SRC_CTL1     = 0x10;// plain text offset
*CRYPTO_DST_CTL0     = 0x20;// round key 9 offset
*CRYPTO_DST_CTL1     = 0x30;// cipher text offset
*CRYPTO_AES_CTL      = 0;// 128-bit key
*CRYPTO_CMD          = (1 << 0);// start block cipher operation
while (*CRYPTO_CMD & (1 << 0)) ;// wait for operation to complete
for (i = 0 ; i < 16; i++) CipherText[i] = *(CRYPTO_MEM_BUFF + 0x30 + i);
```

Note on AES operation modes

National Institute of Standards and Technology (NIST) publication 800-30A specifies block cipher modes of operation. It recommends five modes of operation for use with an underlying symmetric key block cipher algorithm. They are:

- ECB: Electronic Code Book
- CBC: Cipher Block Chaining
- CFB: Cipher Feedback
- OFB: Output Feedback
- CTR: Counter

The AES component of this block provides ECB mode of operation. The other modes of operation involve pre and/or post XOR operations of an initialization vector and plaintext/ciphertext. These XOR operations can be implemented in software to implement other operation modes of AES.

16.3.2.3 SHA

The SHA operation is under software control. It involves the following steps:

1. Enable the block.
2. Initialize the memory buffer with initial hash value.
3. Initialize the memory buffer with the message block.
4. Specify the memory buffer offsets of the source and destination operands.

5. Specify the hash operation.
6. Start a hash operation.
7. Retrieve the results from the memory buffer when the operation has completed.

The following code sequence illustrates how a SHA2-224 operation is performed on a three byte message "abc".

```
uint8_t  Message[]   = "abc";
uint8_t InitHash[32] = {// initial hash value per the standard
 0xc1, 0x05, 0x9e, 0xd8, 0x36, 0x7c, 0xd5, 0x07, 0x30, 0x70, 0xdd, 0x17, 0xf7, 0x0e, 0x59,
0x39,
 0xff, 0xc0, 0x0b, 0x31, 0x68, 0x58, 0x15, 0x11, 0x64, 0xf9, 0x8f, 0xa7, 0xbe, 0xfa, 0x4f,
0xa4};
uint8_t ProducedHash[32] = {// expected final hash
 0x23, 0x09, 0x7d, 0x22, 0x34, 0x05, 0xd8, 0x22, 0x86, 0x42, 0xa4, 0x77, 0xbd, 0xa2, 0x55,
0xb3,
 0x2a, 0xad, 0xbc, 0xe4, 0xbd, 0xa0, 0xb3, 0xf7, 0xe3, 0x6c, 0x9d, 0xa7, 0xd2, 0xda, 0x08,
0x2d};

*CRYPTO_CTL          = (1 << 31) | "SHA"; // enable block

size = sizeof (Message) - 1;    // forget about trailing 0x00.

for (i = 0 ; i < size; i++) *(CRYPTO_MEM_BUFF + i) = Message[i];
*(CRYPTO_MEM_BUFF + i) = 0x80; i++;// append '1'
for (i < 63; i++) *(CRYPTO_MEM_BUFF + i) = 0x00;
*(CRYPTO_MEM_BUFF + 63) = 0x18;// bit size of message

for (i = 0 ; i < 32; i++) *(CRYPTO_MEM_BUFF + 0x80 + i) = InitHash[i];

*CRYPTO_SRC_CTL0     = 0x00;// message block offset
*CRYPTO_SRC_CTL1     = 0x80;// initial hash offset
*CRYPTO_DST_CTL0     = 0x100;// round keys (64 words) offset
*CRYPTO_DST_CTL1     = 0x80;// produced hash
*CRYPTO_SHA_CTL      = 2;// SHA2-224
*CRYPTO_CMD          = (1 << 0);// start operation
while (*CRYPTO_CMD & (1 << 0));// wait for operation to complete
for (i = 0 ; i < 32; i++) ProducedHash[i] = *(CRYPTO_MEM_BUFF + 0x80 + i);
```

16.3.2.4 CRC

CRC is under software control. It involves the following steps:

1. Enable the block.
2. Initialize the memory buffer with the data.
3. Specify the size of the data array.
4. Specify the memory buffer offsets of the source operand (data).
5. Specify the processing of the data (XOR mask and bit reversal).
6. Specify the polynomial.
7. Specify the LFSR seed values.
8. Specify the processing of the remainder (XOR mask and bit reversal).
9. Start a CRC operation.
10. Retrieve the results from the CRC_REM MMIO register when the operation has completed.

The following code sequence illustrates how a CRC operation is performed with a 32-bit polynomial:

```
uint8_t Data[] = {
 0x12, 0x34, 0x56, 0x78, 0x9a};

*CRYPTO_CTL          = (1 << 31) | "CRC";// enable block
```

```
for (i = 0 ; i < 6; i++) *(CRYPTO_MEM_BUFF + i) = Data[i];
*CRYPTO_SRC_CTL0     = 0x00;// data offset
*CRYPTO_CRC_DATA_CTL0      = (5 << 16) | 1;// 6 Bytes, data byte bit reversal
*CRYPTO_CRC_DATA_CTL1      = 0x00;// data byte XOR pattern
*CRYPTO_CRC_POL_CTL        = 0x04c11db7;// polynomial
*CRYPTO_CRC_LFSR_CTL       = 0xffffffff;// seed value
*CRYPTO_CRC_DATA_CTL0      = 1;// remainder bit reversal
*CRYPTO_CRC_DATA_CTL1      = 0xffffffff;// remainder XOR pattern
*CRYPTO_CMD                = (1 << 0);// start CRC operation
while (*CRYPTO_CMD & (1 << 0)) ;// wait for operation to complete
result = *CRYPTO_CRC_REM;
```

16.4 Block Interface Requirements

16.4.1 Functional Timing Requirements and Diagrams

Note the following:

- The interrupt output (“interrupt”) is level-sensitive and remains high until software clears it.
- The trigger input can be asynchronous to clk_sys, but its width should be at least two cycle pulse on clk_sys.
- The trigger output generated (tr_out) is a two cycle pulse on clk_sys.

16.4.2 Bandwidth/Latency Requirements

Table 16-6 documents the performance of Crypto block. The clock cycles referred to are related to this particular block's clock cycles. The performance depends on the operation that is performed. The numbers are for the operation only, and do not include time that software takes to set up the MMIO registers or memory buffer, or the time that software takes to retrieve the results from MMIO registers or memory buffer.

Note: There will be one more cycle delay when the operation is initiated by the hardware trigger input (because the trigger is synchronized inside the block) if the number of cycles is measured between tr_in and tr_out.

Table 16-6. Performance

Operation	Performance (Clock cycles refer to the block's clock cycles)
AES-128 forward cipher	106 clock cycles
AES-128 inverse cipher	103 clock cycles
AES-192 forward cipher	126 clock cycles
AES-192 inverse cipher	123 clock cycles
AES-256 forward cipher	146 clock cycles
AES-256 inverse cipher	143 clock cycles
PR number generation	34 clock cycles
SHA1	516 clock cycles
SHA2-224	413 clock cycles
SHA2-256	413 clock cycles
CRC	1 clock cycle per data Byte + 4

Note: Software should be aware of the number of cycles that an operation is going to take when implementing Sleep power mode and wake up through CRYPTO interrupt. A case might happen where CRYPTO interrupt occurs before even the WFI instruction is executed (depending on the delay between start of CRYPTO operation and execution of WFI instruction).

16.4.3 Bit/Byte Ordering

All the Bit/Byte ordering in this block is little-endian.

The operand data in the SRAM memory should also be laid out in little-endian format (Least Significant Byte of multi byte word should be located at the lowest memory address of the word).

16.4.4 Data Underrun/Overrun

The SRAM memory stores the operand data and results of the operation. Operand data should not overlap in the memory buffer as destination operands may overwrite source operands. For AES, it is allowed to overwrite the 'start key' with the 'last round key' and similarly it is also allowed to overwrite the input data with the output data if user want to reduce the space needed in buffer.

Table 16-7 shows the space required for each operand pointer for AES and SHA operations.

Table 16-7. AES and SHA Operand Space Requirements

Pointer	AES-128	AES-192	AES-256	SHA1	SHA2
SRC_CTL0	16 Bytes	24 Bytes	32 Bytes	64 Bytes	64 Bytes
SRC_CTL1	16 Bytes	16 Bytes	16 Bytes	20 bytes	32 Bytes
DST_CTL0	16 Bytes	24 Bytes	32 Bytes	320 Bytes	256 Bytes
DST_CTL1	16 Bytes	16 Bytes	16 Bytes	20 bytes	32 Bytes

16.5 Power Architecture and Modes

The following sections explain the power architecture for the SRSS-Lite based SoCs.

16.5.1 SRSS-Lite

16.5.1.1 Power Modes

Active, Sleep, Deep Sleep, and XRES are the power modes defined in SRSS-Lite system resources subsystem. Table 16-8 describes the status of this block during different power modes in the SRSS-Lite based system.

Table 16-8. SRSS-Lite Power Modes

Power Mode	Description
Active, Sleep	All the logic is powered on along with the SRAM memory. The Crypto block is functional in Active and Sleep power modes.
Deep Sleep	All the logic is powered ON.
	System clock (clk_sys) is gated OFF.
	SRAM memory periphery is powered OFF (act_power_en_n = 1).
	SRAM array power is ON (dpslp_power_up_n = 0).
	SRAM content and configuration MMIO are retained

16.5.1.2 Power Domains

Figure 16-18 shows the power domains for SRSS-Lite based SoCs.

Figure 16-18. SRSS-Lite Power Domains

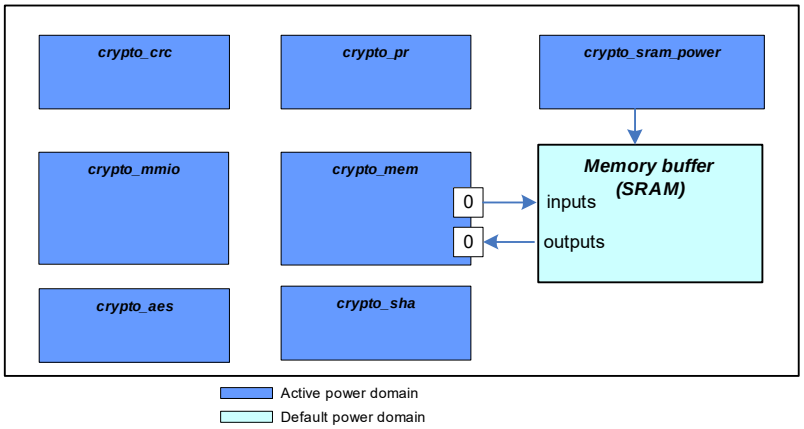
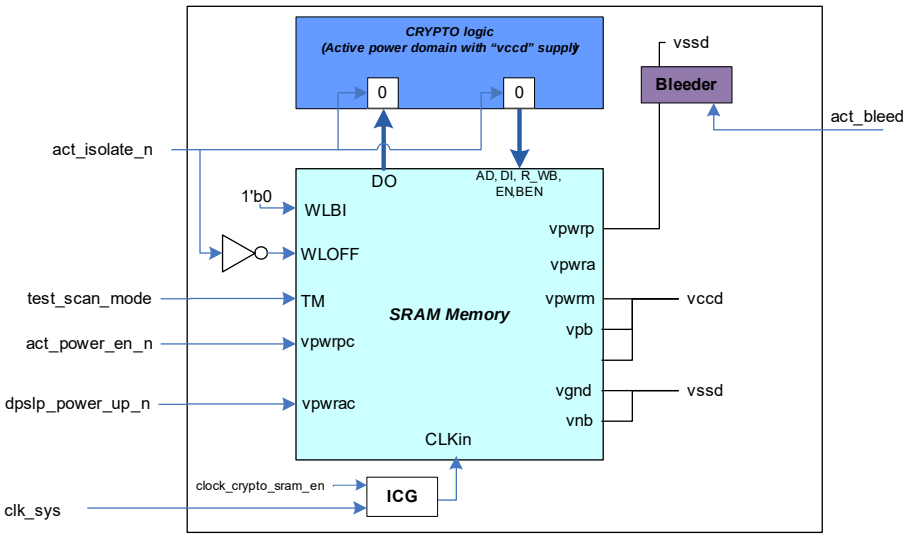


Figure 16-19 shows the detailed SRAM power connections.

Figure 16-19. SRSS-Lite SRAM Power Connections



Note the following:

- All the logic belongs in Active power domain only.
- The SRAM memory periphery will be switched OFF during Deep Sleep power mode.
- Isolation cells will be added both on inputs (AD[], DI[], EN, R_WB, and BEN[]) and outputs (DO[]) of SRAM memory as shown in Figure 16-19. Isolation cells on SRAM memory inputs are required to reduce leakage power due to pass gate multiplexing structures used in the SRAM circuit. UPF defines isolation strategy for these isolation cells.
- The scan related pins of SRAM memory (ScanOutCC, ScanInCC, ScanInDL, and ScanInDR) will be connected only after DfT insertion and should be isolated after DfT insertion at chip level.
- The SRAM memory contents will be retained in Deep Sleep power mode. Therefore, SRAM array power will not be switched OFF during Deep Sleep power mode (Note how the array power control "vpwrac" is connected to dpslp_power_up_en_n).

16.5.1.3 Retention Strategy

- The MMIO registers are connected to rst_sys_dpslp_n and hence the MMIO configuration registers are retained in Deep Sleep power mode.
- The SRAM memory contents will be retained in Deep Sleep power mode.

17. USB Full Speed (USB FS)

The EZ-PD™ PMG1-S2 MCU USB block acts as a USB device that communicates with a USB host. The USB block is available as a fixed-function digital block in the EZ-PD™ PMG1-S2 MCU device. It supports full-speed communication (12 Mbps) and is designed to be compliant with the USB Specification Revision.2.0. USB devices can be designed for plug and play applications with the host and also support hot swapping. This chapter details the EZ-PD™ PMG1-S2 MCU USB block and transfer modes. For details about the USB specification, see the USB Implementers Forum website.

17.1 Features

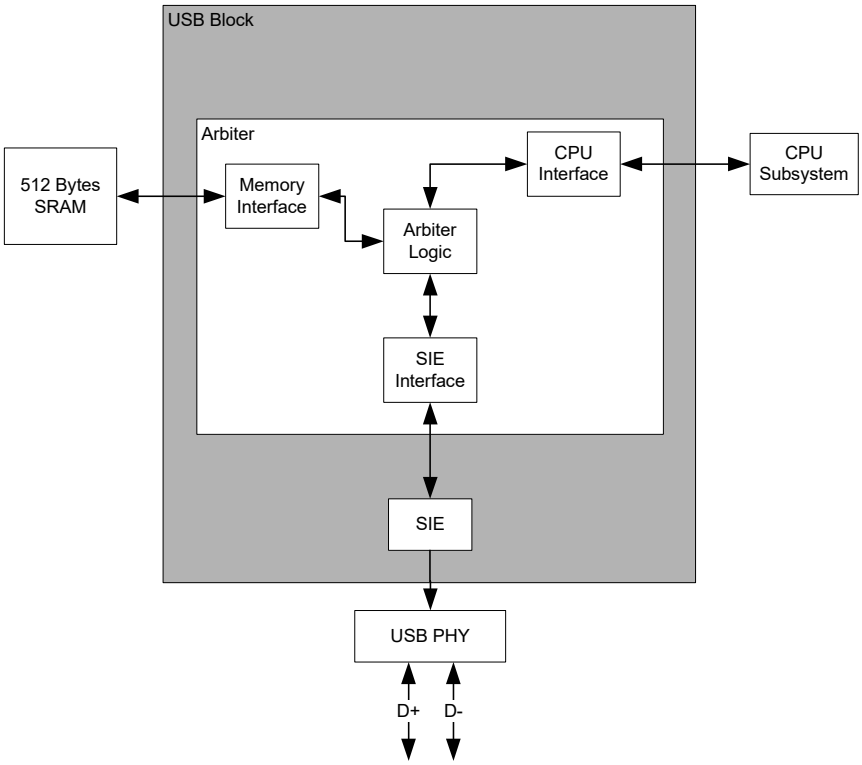
The EZ-PD™ PMG1 MCU USB has these features:

- Complies with USB Specification 2.0
- Supports full-speed peripheral device operation with a signaling bit rate of 12 Mbps
- Supports eight data endpoints and one control endpoint
- Supports four types of transfers – bulk, interrupt, isochronous, and control
- Supports bus- and self-powered configurations
- USB suspend mode for low power
- Supports the following logical transfer mode:
  - Store and Forward mode
- Differential signal (D+ and D-) output
- Integrated 22-Ω series resistors on D+ and D– lines, and 1.5-kΩ pull-up resistor on the D+ line
- Supports maximum packet size of 64 bytes using the Store and Forward mode for bulk endpoints and maximum packet size of 512 for isochronous transfers in store and forward mode

17.2 Block Diagram

Figure 17-1 illustrates the architecture of the USB block. It consists of the USB Physical Layer (USB PHY), Serial Interface Engine (SIE), Arbiter, and the local 512-byte memory buffer.

Figure 17-1. USB FS Block Diagram



17.2.1 USB Physical Layer (USB PHY)

This block takes care of physical layer communication with the USB host through the D+ and D– pins. It handles the differential mode communication with the host and monitoring events such as SE0 on the USB bus.

17.2.2 Serial Interface Engine (SIE)

The Serial Interface Engine (SIE) is responsible for handling the decoding and creating of data and control packets during transmit and receive. It decodes the USB bit streams into USB packets during receive and creates USB bit streams during transmit. The following are the features of the SIE block:

- Conforms to the USB 2.0 Specification
- Supports one device address
- Supports eight data endpoints and control endpoint
- Supports interrupt trigger event for each endpoint
- Integrates an 8-byte buffer in the control endpoint

The registers for this block are mainly used to configure the data endpoint operations and the control endpoint data buffers. This block also controls the interrupt events available for each endpoint.

17.2.3 Arbiter

The Arbiter is the block that handles access of the SRAM memory by the endpoints. The SRAM memory can be accessed by the CPU or the SIE. The Arbiter handles the arbitration between the CPU and the SIE. The Arbiter consists of the following blocks:

- SIE Interface Module
- CPU Interface Module

- Memory Interface
- Arbiter Logic

The Arbiter registers are used to handle the endpoint configurations, Read address, and Write address for the endpoints. It also configures the logical transfer type required for each endpoint. The Arbiter in the EZ-PD™ PMG1-S2 MCU device family supports only mode 0.

17.2.3.1 SIE Interface Module

This module handles all the transactions with the SIE block. The SIE reads data from the SRAM memory and transmits to the host. Similarly, it writes the data received from the host to the SRAM memory. These requests are registered in the SIE Interface module and are handled by this block.

17.2.3.2 Memory Interface

The memory interface is used to control the interface between the USB block and the SRAM memory unit. The maximum memory size supported is 512 bytes organized as 256 x a 16-bit memory unit. This is a dedicated memory for the USB. The memory access can be requested by the SIE or by the CPU. The SIE Interface block and the CPU Interface block handle these requests.

17.2.3.3 Arbiter Logic

This is the main block of the Arbiter. It is responsible for arbitrations for all the transactions that happen in the Arbiter. It arbitrates the CPU and SIE access to the memory unit and the registers. This block also handles the memory management. The memory management is either “Manual” or “Automatic”. In Manual memory management, the read and write address manipulations are done by the firmware. This block also handles the interrupt requests for each endpoint.

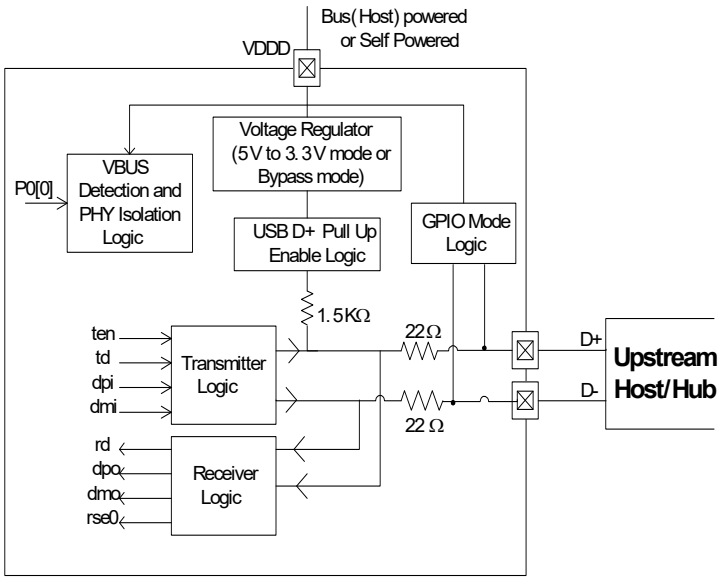
17.3 How it Works

17.3.1 USB Physical Layer (USB PHY)

The USB block includes the transmitter and the receiver (transceiver), which corresponds to the USB PHY.

Figure 17-2 shows the PHY architecture. The USB PHY also includes the pull-up resistor on the D+ line to identity the device as full-speed type to the host. The signal between the USB device and the host is a differential signal. The receiver receives the differential signal from the host and converts it to a single-ended signal for processing by the SIE. The transmitter converts the single-ended signal from the SIE to the differential signal, and transmits it to the host. The differential signal is given to the upstream devices at a nominal voltage range of 0 V to 3.3 V.

Figure 17-2. USB PHY Architecture



17.3.1.1 Power Scheme

The USB PHY is powered by the VDDD power pad of the EZ-PD™ PMG1-S2 MCU device. The USB PHY needs a nominal voltage of 3.3 V for its communication with the host. The REG_ENABLE bit in the USB_CR1 register controls the operation of the internal voltage regulator. EZ-PD™ PMG1-S2 MCU's VDDD rail is derived from a regulated output from VBUS or VSYS. For applications using VBUS as the primary supply (bus-powered applications), the regulator inside the USB PHY must be bypassed by setting REG_ENABLE bit to '0'. In applications using VSYS as the primary supply (self-powered applications), the regulator should be enabled when the VSYS is outside the 3.3 V (3.15 V to 3.45 V) range. When the VSYS supply voltage is in the 5-V range (4.35 V to 5.5 V), the REG_ENABLE bit must be set high to enable the internal regulator.

17.3.1.2 VBUS Detection

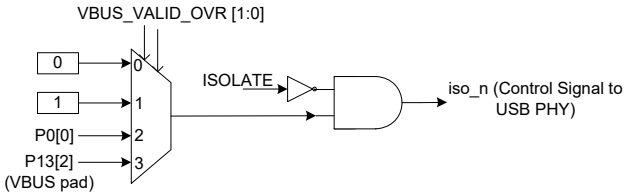
USB devices have the option of deriving power from the VBUS power signal from the host (bus-powered configuration) or having an external power supply independent of host VBUS (self-powered configuration). The EZ-PD™ PMG1-S2 MCU VBUS regulator is inside the USBPD subsystem. The VBUS detect interrupt and status registers from the USBPD subsystem must be used.

The VBUS detect is mapped to the INT_WAKEUP interrupt of the USBPD subsystem. The corresponding VBUS status can be read from NCELL_STATUS register by reading the VBUS_C_STATUS field. After VBUS is detected through the interrupt and status register, the CPU can enable the pull-ups in the device to signal attach to the host.

17.3.1.3 USB PHY Isolation Logic

The USB PHY has an isolation logic to disable the PHY output signals when the host VBUS is not present.

Figure 17-3. USB PHY Output Isolation Logic



The PHY is designed to operate from a VBUS supply independent from any other supplies such as VDDD and VCCD, and contains the appropriate isolation logic. Figure 17-3 shows the isolation logic signal (iso_n) generation. iso_n is an active low signal that is used to isolate the output of the PHY (generated in the VBUS power domain) from the rest of the chip operating in other independent power domain. The device firmware can set the ISOLATE bit field in the USB_POWER_CTRL register to isolate the PHY outputs irrespective of the state of VBUS.

The EZ-PD™ PMG1-S2 MCU PHY does not require any dynamic isolation control by firmware because it operates from a common VDDD power rail. So the VBUS_VALID_OVR[1:0] bit fields in the USB_POWER_CTRL register should be selected as 1 or 3 from the following selection:

1: Force vbus_valid=1

3: Use vbus_valid signal from PHY detector.

- The device firmware should enable the USB block operation only after the host VBUS is present - detected by reading the status of VBUS from the NCELL_STATUS register of USBPD (VBUS_C_STATUS field)
- When VBUS is present, the device firmware should clear the ISOLATE bit field in the USB_POWER_CTRL register after a delay of at least 2 µs. Thereafter, the signal selected by VBUS_VALID_OVR[1:0] will take care of dynamic PHY output isolation. When the USB block is stopped in the device firmware, the ISOLATE bit field should be set as part of the stop sequence

17.3.1.4 USB D+ pin Pull-up Enable Logic

When a USB device is self-powered, the USB specification warrants that the device enable the pull-up resistor on its D+ pin to identify itself as a full-speed device to the host. When the host VBUS is removed, the device should disable the pull-up resistor on the D+ line so as to not back power the host. The USB PHY block includes an internal 1.5-k pull- up resistor on the D+ line to indicate to the host that the EZ-PD™ PMG1-S2 MCU is a full-speed device. The pull-up resistor can be enabled or disabled by configuring the USBPUEN bit in the USBIO_CR1 register. The firmware should read VBUS status to enable/disable the pull-up resistor on the D+ pin.

17.3.1.5 Transmitter and Receiver Logic

The transmitter logic takes care of differential transmission to the USB host.

The transmitter can be manually forced to transmit signals. The USB_USBIO_CR0 register is used to manually transmit the signals. Examples are as follows:

- When the manual transmission is enabled, the register can be configured to transmit Single-Ended Zero signal (that is, D+ and D– are low).
- Configurable to transmit the USB signals. The USB signals can be two types:
  - D+ low and D– high = J
  - D+ high and D– low = K
- The register also has a bit, which is used to read the received signal levels. The bit can show if D+ < D– or D+ > D–.

17.3.2 Endpoints

The SIE and Arbiter support eight data endpoints (EP1 to EP8) and one control endpoint (EP0). The data endpoints share the SRAM memory area of 512 bytes. The endpoint memory management can be either Manual or Automatic. The endpoints are configured for direction and other configuration using the SIE and arbiter registers. The endpoint read address and write address registers are accessed through the Arbiter.

The endpoints can be individually made active. In the Auto Management mode, the register USB_EP_ACTIVE is written to control the active state of the endpoint. The endpoint activation cannot be dynamically changed during runtime. In Manual Memory Management mode, the firmware decides the memory allocation, so it is not required to specify the active endpoints. The USB_EP_ACTIVE register is ignored during the manual memory management mode. The USB_EP_TYPE register is used to control the transfer direction (IN, OUT) for the endpoints. The control endpoint has separate eight bytes for its data (USB_EP0_DR registers).

17.3.3 Transfer Types

The EZ-PD™ PMG1-S2 MCU USB supports full-speed transfers and is compliant with the USB 2.0 Specification. It supports four types of transfers:

- Interrupt Transfer
- Bulk Transfer
- Isochronous Transfer
- Control Transfer

For further details about these transfers, refer to the USB Specification 2.0.

17.3.4 Interrupts

The USB block has three general-purpose interrupt lines – INTR_LO, INTR_MED, and INTR_HI. Refer to the Interrupts chapter on page 29 for the vector numbers of these interrupt lines. The USB block has a predefined set of 12 interrupt trigger events that can be mapped to either one of the three interrupts. The interrupt line to which each of the trigger event is mapped is configured through the USB_INTR_LVL_SEL register. Each of these three interrupt lines has a status register to identify the event that cause of the interrupt. These are the USB_INTR_CAUSE_LO, USB_INTR_CAUSE_MED, and USB_INTR_CAUSE_HI status registers.

The following 12 events can be used to generate an interrupt on one of the three interrupt lines.

- USB Start of Frame (SOF) Event
- USB Bus Reset Event
- Eight Data Endpoint (EP1 – EP8) interrupt events
- Control Endpoint (EP0) interrupt event
- Arbiter Interrupt event

The following sections provide details on each of these interrupt events.

17.3.4.1 Data Endpoint Interrupt Events

These are eight interrupt events corresponding to each data endpoint (EP1-EP8). Each of the endpoint interrupt events can be enabled/disabled by using the corresponding bit in the USB_SIE_EP_INT_EN register. The interrupt status of each endpoint can be known by reading the USB_SIE_EP_INT_SR status register. An endpoint whose interrupt is enabled can trigger

the interrupt on the following events:

- Successful completion of an IN/OUT transfer
- NAK-ed IN/OUT transaction if the corresponding NAK_INT_EN bit in the SIE_EPx_CR0 register is set
- When there is an error in the transaction, the bit ERR_IN_TXN in the SIE_EPx_CR0 register is set and interrupt is generated. Refer to the register description for details on the conditions under which this bit can be set.
- If the STALL bit in SIE_EPx_CR0 is set, then stall events can generate interrupts. This stall event can happen if an OUT packet is received for an endpoint whose mode bits in SIE_EPx_CR0 are set to ACK_OUT or if an IN packet is received with mode bits set to ACK_IN.

17.3.4.2 Control Endpoint (EP0) Interrupt Event

The interrupt event corresponding to the control endpoint (EP0) is generated under the following events:

- Successful completion of an IN/OUT transfer
- When a SETUP packet is received on the control endpoint

17.3.4.3 Arbiter Interrupt Event

The arbiter generates an interrupt event for the endpoints during the following events - the final arbiter interrupt event is the logical OR of these events.

IN endpoint local buffer full:

This event status is set on different conditions in Store and Forward mode.

- Store and Forward Mode: This status is set when the entire packet data has been transferred to the local memory. The check is that data written for the particular endpoint is equal to the programmed Byte Count for that endpoint in the USB.SIE_EPx.CNT0 and USB.SIE_EPx.CNT1 registers.

17.3.4.4 USB Start of Frame (SOF) Event

- Generated whenever the SOF is received. The SOF interrupt is enabled by setting the SOF_INTR_MASK bit in the USB_INTR_SIE_MASK register. The SOF interrupt status is reflected in the SOF_INTR status bit in the USB_INTR_SIE status register.
- The SOF interrupt status is also available in the SOF_INTR_MASKED bit of the USB_INTR_SIE_MASKED register – this bit is the logical AND of corresponding SOF bits in the USB_INTR_SIE_MASK register and the USB_INTR_SIE register.
- The USB packet decoder in the SIE decodes the SOF PID and asserts a pulse of width one 12-MHz clock cycle. This pulse is synchronized to the system clock (SYSCLK) and can be routed as a digital signal through the Digital System Interconnect (DSI) interface. This routing feature is in addition to the interrupt feature.

17.3.4.5 USB Bus Reset Event

- Generated whenever a USB bus reset condition occurs. The bus reset interrupt is enabled by setting the BUS_RESET_INTR_MASK bit in the USB_INTR_SIE_MASK register. The bus reset interrupt status is reflected in the BUS_RESET_INTR status bit in the USB_INTR_SIE status register.
- The bus reset interrupt status is also available in the BUS_RESET_INTR_MASKED bit of the USB_INTR_SIE_MASKED register – this bit is the logical AND of corresponding bus reset bits in the USB_INTR_SIE_MASK register and the USB_INTR_SIE register.
- The 32-kHz low-frequency clock (LFCLK) is used to detect the USB bus reset condition. It is required that the LFCLK is enabled in the application firmware. The SIE logic triggers on counter running on LFCLK when a SE0 condition is detected on the USB bus. When the counter reaches the count value configured in the USB_BUS_RST_CNT register, the bus reset interrupt is triggered. It is recommended to set the count value in the USB_BUS_RST_CNT register to three to detect the bus reset condition.

17.4 Logical Transfer Modes

The USB block in EZ-PD™ PMG1-S2 MCU devices supports only mode 1, which is store and forward logical transfer mode. Table 17-1 describes the details of Store and Forward transfer mode.

Table 17-1. USB Transfer Modes

Feature	Store and Forward Mode
SRAM Memory Usage	Requires more memory
SRAM Memory Management	Manual
SRAM Memory Sharing	512 bytes of SRAM shared between endpoints. Sharing is done by firmware.
IN Command	Entire packet present in SRAM memory before the IN command is received.
OUT Command	Entire packet is written to SRAM memory on OUT command. After entire data is available, it is copied from SRAM memory to the USB device.
Transfer of Data	Data is transferred when all bytes are written to the memory.
Supported Transfer Types	Best suited for Interrupt and Bulk transfers

The logical transfer can be configured using the register setting for each endpoint. Any of the logical transfer methods can be adapted to support the three types of data transfers (Interrupt, Bulk, and Isochronous) mentioned in the USB 2.0 Specification. The control transfer is mandatory in any USB device.

Every endpoint has a set of registers that need to be handled during the modes of operation, as detailed in Table 17-2.

Table 17-2. Endpoint Registers

Register	Comment	Content	Usage
ARB_RWx_WA	Endpoint Write Address register	Address of the SRAM	This register indicates the SRAM location to which the data in the Data register is to be written.
ARB_RWx_RA	Endpoint Read Address register	Address of the SRAM	This register indicates the SRAM location from which the data must be read and stored to the Data register.
ARB_RWx_DR	Endpoint Data Register	8-Bit Data	Data register is read/ written to perform any transaction. IN command: Data written to the Data register is copied to the SRAM location specified by the WA register. After write, the WA value is automatically incremented to point to the next memory location. OUT command: Data available in the SRAM location pointed by the RA register is read and stored to the DR. When the DR is read, the value of RA is automatically incremented to point to the next SRAM memory location that must be read.
SIE_EPx_CNT0 and SIE_EPx_CNT1	Endpoint Byte Count Register	Number of Bytes	Holds the number of bytes that can be transferred. IN command: Holds the number of bytes to be transferred to host. OUT command: Holds the maximum number of bytes that can be received. The firmware programs the maximum number of bytes that can be received for that endpoint. The SIE updates the register with the number of bytes received for the endpoint.
“Mode” bits in SIE_EPx_CR0	Mode Values	Response to the Host	Controls how the USB device responds to the USB traffic and the USB host. Some examples of mode include ACK, NAK, STALL, etc. See Table 17-1 for additional details.

In Manual Memory Management, the endpoint read and endpoint write address registers are updated by the firmware. So the memory allocation can be done as required by the user and the memory allocation decides which endpoints are active; that is, the user can decide to share the 512 bytes for all the eight endpoints or a lesser number of endpoints.

In Automatic memory management, the endpoint read and endpoint write address registers are updated by the USB block. The block assigns memory to the endpoints that are activated using the USB_EP_ACTIVE register. The size of memory allocated depends on the value in the USB_BUF_SIZE register. The remaining memory, after allocation, is called the “Common Area” memory and used for the transfer of data.

In the following text, the algorithm for the IN and OUT transaction for each mode is discussed. An IN transaction is when the data is read by the USB host (for example, PC). An OUT transaction is when the data is written by the USB host to the USB device.

17.4.1 Store and Forward Mode

17.4.1.1 No DMA Access

This is the Manual Memory Management mode with no DMA access.

IN Transaction (CPU Write, SIE Read): The steps for an IN transaction on an IN endpoint are shown in Figure 17-4. OUT Transaction (CPU Read, SIE Write): The steps for an OUT transaction on an OUT endpoint are shown in Figure 17-5.

Figure 17-4. No DMA Access IN Transaction

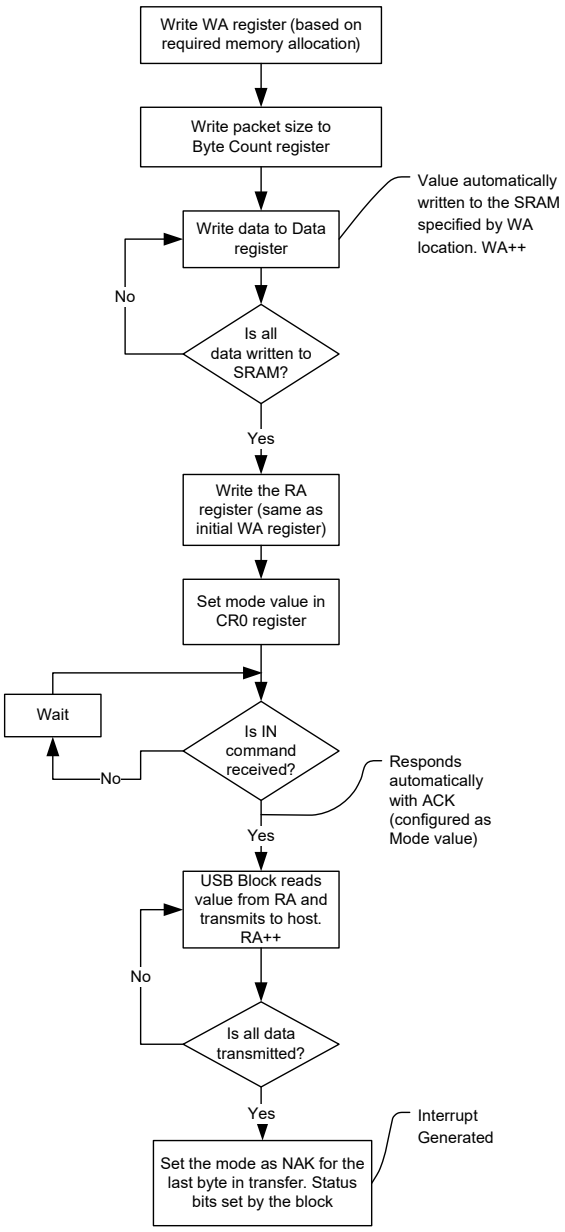
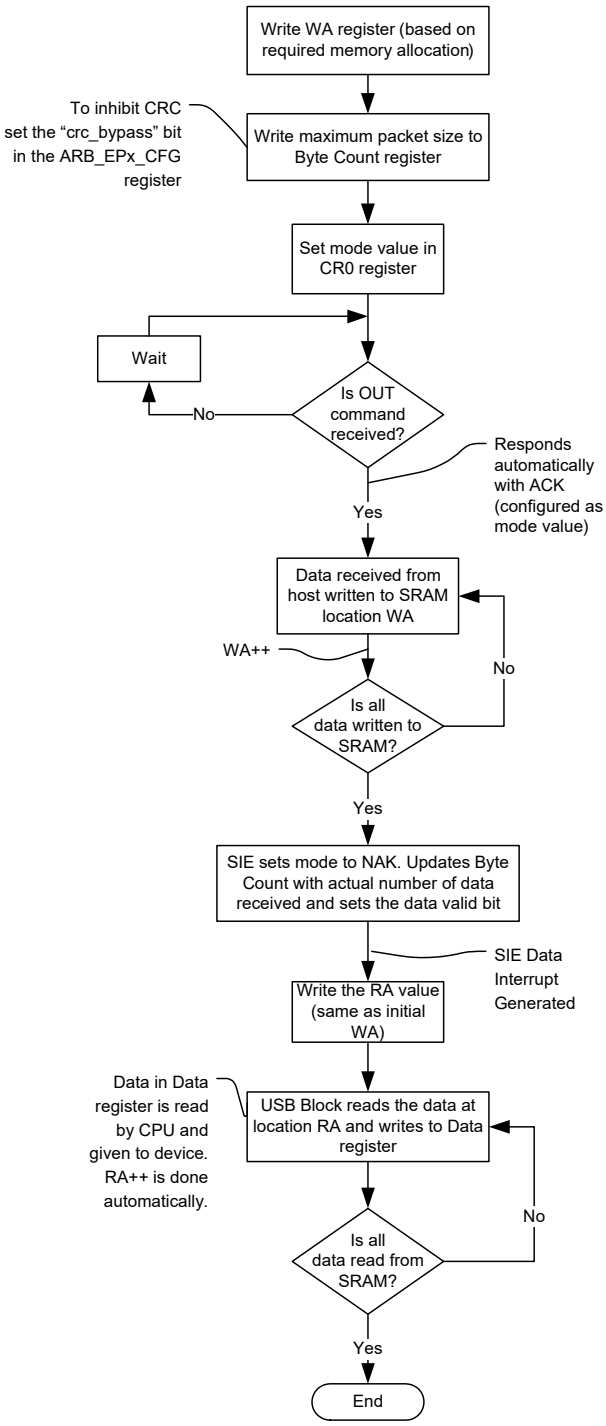


Figure 17-5. No DMA Access OUT Transaction



17.4.1.2 *Manual DMA Access/CutThrough Mode*

EZ-PD™ PMG1-S2 MCU does not support DMA modes (manual or cut-through) as there is no DMA engine in EZ-PD™ PMG1-S2 MCU devices.

17.4.2 Control Endpoint Logical Transfer

The control endpoint has a special logical transfer mode. It does not share the 512 bytes of memory. Instead, it has a dedicated 8-byte register buffer (USB_EP0_DRx registers). The IN and OUT transaction for the control endpoint is detailed in the following figures.

Figure 17-6. Control Endpoint IN Transaction

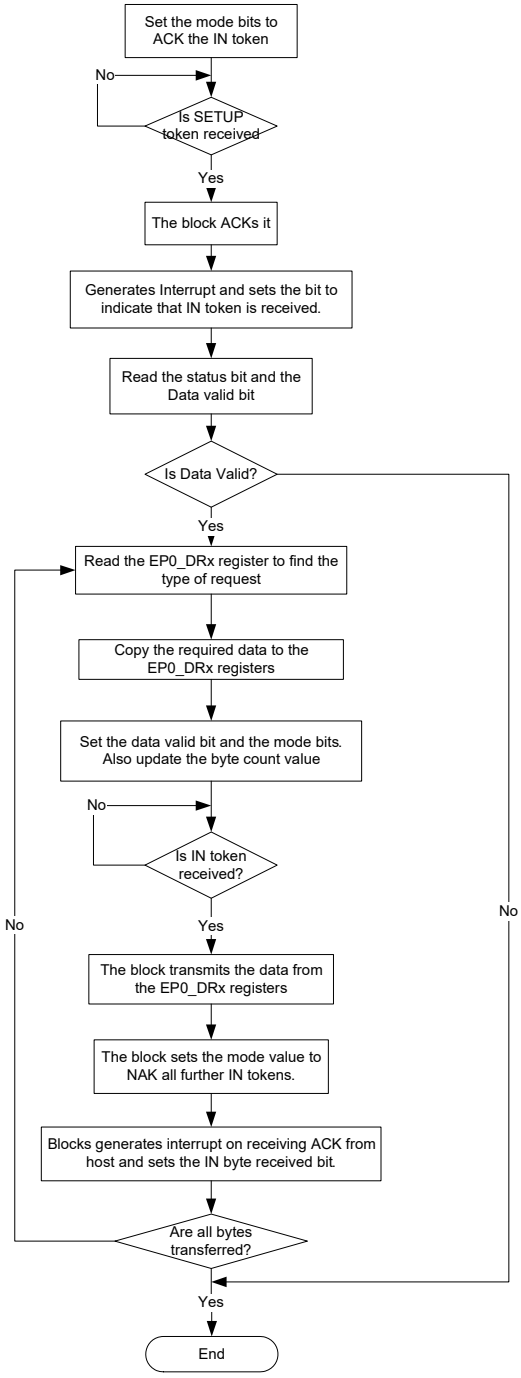
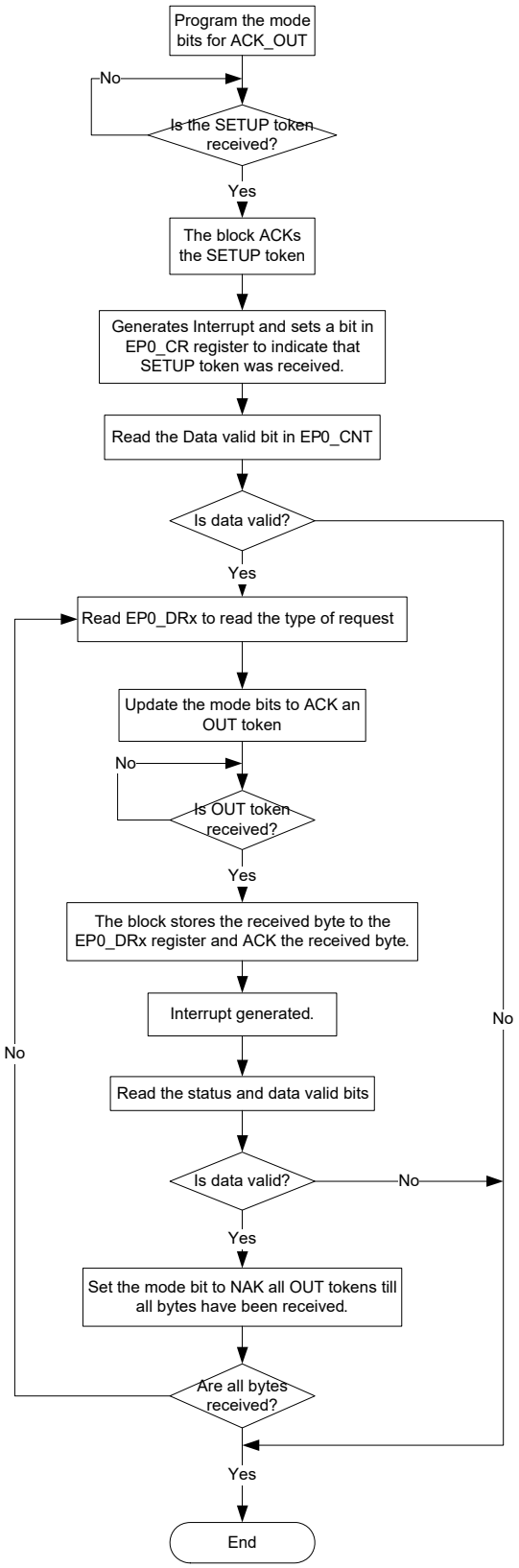


Figure 17-7. Control Endpoint OUT Transaction



17.5 Register Summary

Register Name	Description
USBDEVv2_EP0_DRx	Control Endpoint EP0 Data Register. 'x' can be 0–7 for the 8-byte EP0 data buffer
USBDEVv2_CR0	USB Control 0 Register. Register to enable the USB device and specify the 7-bit device address
USBDEVv2_CR1	USB Control 1 Register. Register to configure the USB block regulator, read bus activity status, and enable the internal oscillator to the USB traffic
USBDEVv2_SIE_EP_INT_EN	USB SIE Data Endpoints Interrupt Enable Register
USBDEVv2_SIE_EP_INT_SR	USB SIE Data Endpoint Interrupt Status
USBDEVv2_SIE_EPx_CNT0	Non-control Endpoint Byte Count Register. 'x' can be 1–8 corresponding to one of the eight data endpoints. This register stores the most significant 3-bits of the 11-bit byte counter, the toggle state of the data packet, and data valid status.
USBDEVv2_SIE_EPx_CNT1	Non-control Endpoint Byte Count Register. 'x' can be 1–8 corresponding to one of the eight data endpoints. This register stores the lower eight bits of the 11-bit byte count value.
USBDEVv2_SIE_EPx_CR0	Non-control Endpoint Control Register. 'x' can be 1–8 corresponding to one of the eight data endpoints. This register contains the endpoint operating mode, error status, stall control, and the NAK interrupt generation.
USBDEVv2_USBIO_CR0	USBIO Control 0 Register. This register contains the control and configuration bits for the USB I/Os (D+ and D– pins respectively) for single-ended and differential mode operation.
USBDEVv2_USBIO_CR1	USBIO Control 1 Register. This register contains the control and configuration bits for the USB I/Os (D+ and D– pins respectively) for selecting the USB operating mode, reading the single ended USBIO receiver outputs, and enabling pull-up resistor on the D+ line.
USBDEVv2_DYN_RECONFIG	USB Dynamic Reconfiguration Register. This register is used to control the status of dynamic reconfiguration of an endpoint.
USBDEVv2_SOF0	Start Of Frame Register. This register contains the lower eight bits [7:0] of the SOF frame number.
USBDEVv2_SOF1	Start Of Frame Register. This register contains the upper three bits [10:8] of the SOF frame number.
USBDEVv2_OSCLK_DR0	Oscillator Lock Data Register 0. This register contains the lower eight bits of the oscillator locking circuit's adder output.
USBDEVv2_OSCLK_DR1	Oscillator Lock Data Register 1. This register contains the upper seven bits of the oscillator locking circuit's adder output.
USBDEVv2_EP0_CR	Endpoint0 Control Register. This register contains operating mode of the control endpoint, and the status bits for different packet conditions on the control endpoint.
USBDEVv2_EP0_CNT	Endpoint0 Count Register. This register stores the 4-bit byte counter, the toggle state of the data packet, and data valid status.
USBDEVv2_ARB_EPx_CFG	Endpoint Configuration Register. 'x' can be 1–8 corresponding to one of the eight data endpoints. This register contains the settings to reset the endpoint buffer pointers, CRC bypass feature, manual DMA request, and the data ready status.
USBDEVv2_ARB_EPx_INT_EN	Endpoint Interrupt Enable Register. 'x' can be 1–8 corresponding to one of the eight data endpoints. Register to configure the conditions under which an interrupt should be generated for an endpoint.
USBDEVv2_ARB_EPx_SR	Endpoint Interrupt Status Register. 'x' can be 1–8 corresponding to one of the eight data endpoints. Register to read the interrupt status of an endpoint.
USBDEVv2_ARB_RWx_WA	Endpoint Write Address value. 'x' can be 1–8 corresponding to one of the eight data endpoints. Lower eight bits of the 9-bit write address pointer.
USBDEVv2_ARB_RWx_WA_MSB	Endpoint Write Address value. 'x' can be 1–8 corresponding to one of the eight data endpoints. Most significant bit of the 9-bit write address pointer.
USBDEVv2_ARB_RWx_RA	Endpoint Read Address value. 'x' can be 1–8 corresponding to one of the eight data endpoints. Lower eight bits of the 9-bit read address pointer.
USBDEVv2_ARB_RWx_RA_MSB	Endpoint Read Address value. 'x' can be 1–8 corresponding to one of the eight data endpoints. Most significant bit of the 9-bit read address pointer.
USBDEVv2_ARB_RWx_DR	Endpoint Data Register. 'x' can be 1–8 corresponding to one of the eight data endpoints.
USBDEVv2_BUF_SIZE	Dedicated Endpoint Buffer Size Register
USBDEVv2_EP_ACTIVE	Endpoint Active Indication Register

Register Name	Description
USBDEVv2_EP_TYPE	Endpoint Type (IN/OUT) Indication register
USBDEVv2_ARB_CFG	Arbiter Configuration Register. This register is used to configure the buffer management mode of the USB block.
USBDEVv2_USB_CLK_EN	USB Block Clock Enable Register
USBDEVv2_ARB_INT_EN	Arbiter Interrupt Enable Register. This register contains the configuration to enable arbiter interrupt generation for each endpoint.
USBDEVv2_ARB_INT_SR	Arbiter Interrupt Status Register. This register contains the status of arbiter interrupt generation due to each endpoint.
USBDEVv2_CWA	Common Area Write Address Register. This register contains the lower eight bits of the 9-bit write address pointer for the common area.
USBDEVv2_CWA_MSB	Common Area Write Address Register. This register contains the most significant bit of the 9-bit write address pointer for the common area.
USBDEVv2_DMA_THRES	DMA Burst/Threshold Configuration Register. This register contains the lower eight bits of the 9-bit threshold byte count value.
USBDEVv2_DMA_THRES_MSB	DMA Burst/Threshold Configuration Register. This register contains the most significant bit of the 9-bit threshold byte count value.
USBDEVv2_BUS_RST_CNT	Bus Reset Count Register. This register contains the number of clock cycles of LFCLK that should elapse to detect a bus reset condition.
USBDEVv2_MEM_DATAx	This is the 512-byte data buffer for storing the non-control endpoint data. 'x' can be 0 to 511.
USBDEVv2_SOF16	16-bit version of the Start Of Frame Register
USBDEVv2_OSCLK_DR16	16-bit version of the Oscillator Lock Data Register
USBDEVv2_ARB_RWx_WA16	16-bit version of the Endpoint Write Address Value Register. 'x' can be 1 to 8.
USBDEVv2_ARB_RWx_RA16	16-bit version of the Endpoint Read Address Value Register. 'x' can be 1 to 8.
USBDEVv2_CWA16	16-bit version of the Common Area Write Address Register
USBDEVv2_ARB_RWx_DR16	16-bit version of the Endpoint Data Register. 'x' can be 1 to 8
USBDEVv2_DMA_THRES16	16-bit version of the DMA Burst/Threshold Configuration Register
USBDEVv2_USB_POWER_CTRL	Power Control Register
USBDEVv2_USB_CHGDET_CTRL	Charger Detection Control Register
USBDEVv2_USB_USBIO_CTRL	USB I/O Control Register
USBDEVv2_USB_FLOW_CTRL	Flow Control Register
USBDEVv2_USB_LPM_CTRL	LPM Control Register
USBDEVv2_USB_LPM_STAT	LPM Status Register
USBDEVv2_USB_INTR_SIE	USB SOF, BUS RESET, and EP0 Interrupt Status Register
USBDEVv2_USB_INTR_SIE_SET	USB SOF, BUS RESET, and EP0 Interrupt Set Register
USBDEVv2_USB_INTR_SIE_MASK	USB SOF, BUS RESET, and EP0 Interrupt Mask Register
USBDEVv2_USB_INTR_SIE_MASKED	USB SOF, BUS RESET, and EP0 Interrupt Masked Register
USBDEVv2_USB_INTR_LVL_SEL	USB Interrupt Level Select Register
USBDEVv2_USB_INTR_CAUSE_HI	High Priority Interrupt Cause Register
USBDEVv2_USB_INTR_CAUSE_MED	Medium Priority Interrupt Cause Register
USBDEVv2_USB_INTR_CAUSE_LO	Low Priority Interrupt Cause Register
USBDEVv2_USB_PHY_TRIM0	PHY Trim Control Register
USBDEVv2_USB_PHY_TRIM1	PHY Trim Control Register
USBDEVv2_USB_PHY_TRIM2	PHY Trim Control Register
USBDEVv2_USB_PHY_TRIM3	PHY Trim Control Register
USBDEVv2_USB_CHGDET_TRIM	Charger Detect Trim Values Register
USBDEVv2_USB_TRIM	Trim Values Register
USBDEVv2_USB_USBIO_TRIM	Trim Values for I/Os Register

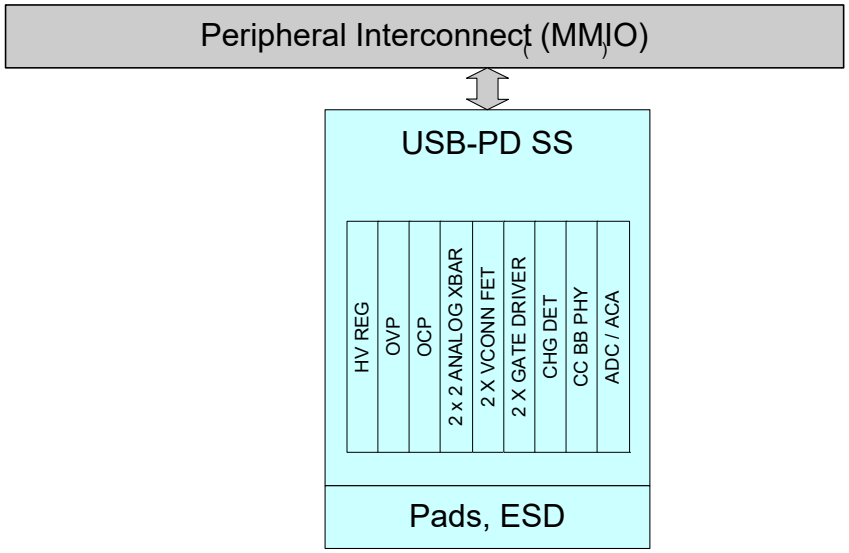
Section F: USB Power Delivery

This section encompasses the following chapter:

- USB Power Delivery chapter on page 160

Top Level Architecture

USB Power Delivery Block Diagram



18. USB Power Delivery

EZ-PD™ PMG1-S2 MCU uses a built-in USB Power Delivery (USB PD) baseband PHY. This PHY supports all the features as per the USB Power Delivery 3.0 specification and USB Type-C Cable and Connector specification rev 1.3.

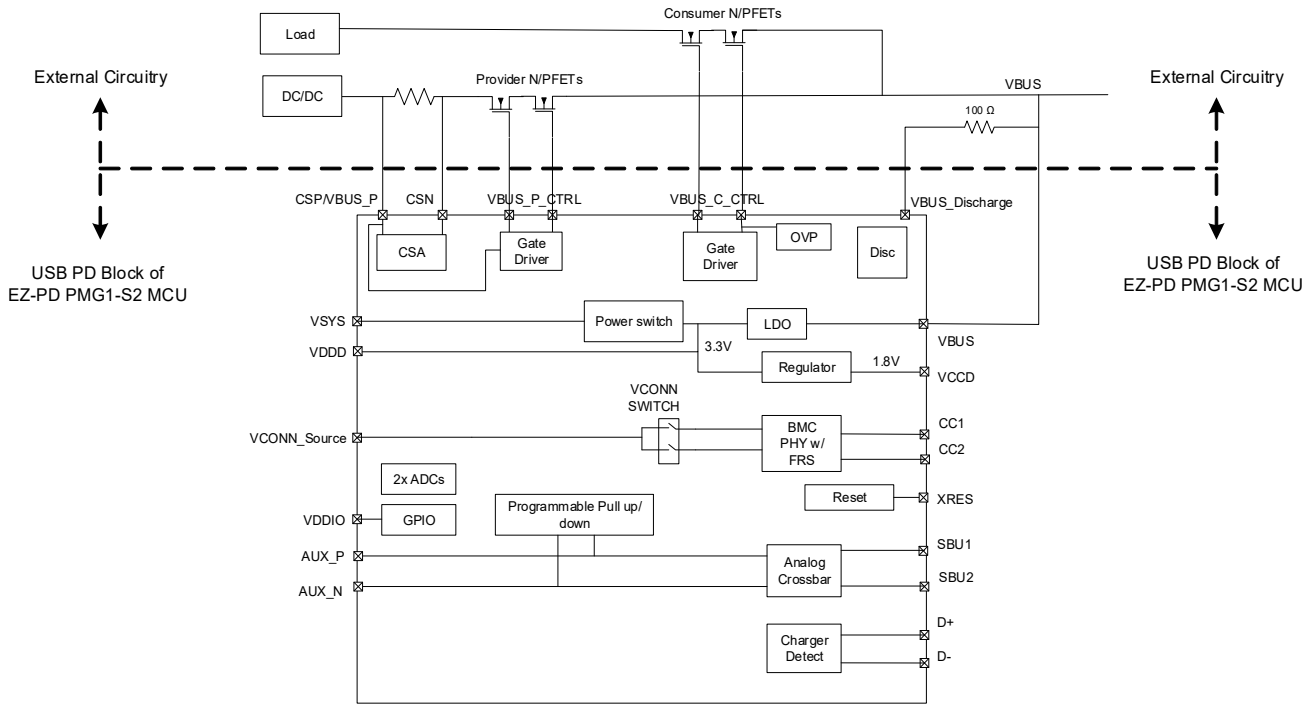
The USB PD block implements the PD communication using a dedicated Communication Channel (CC) using Bi-phase Mark Coding (BMC). The block contains a dedicated SRAM to store USB PD RX and TX header and messages coming as a part of USB PD negotiations. In addition, it includes all termination resistors required to implement various Type-C roles such as DFP, UFP and DRP. This block includes everything necessary to support UFP, DFP, and DRP for the USB Type-C connector. The baseband transceiver is used for SOP, SOP', or SOP'' communication over the CC line.

The block also implements under-voltage and over-voltage detection circuits. The current sense amplifier (CSA) is used for over-current sensing. An 8-bit SAR ADC is used for generic voltage sensing of anything connected to the analog mux (AMUX) buses, as well as voltage supply measurement and temperature sensing. The USB PD block also implements battery charger emulation and detection for BC.1.2 and Apple charging protocols. The block has bi-directional analog switches to connect the SBU1/2 pins to the AUX_P/N pins in one of two combinations.

Use the USB PD component provided by the ModusToolbox (MTB) IDE to implement the USB PD block in designs. See the *Getting Started with EZ-PD™ PMG1 MCU MCU on ModusToolbox* application note for more details.

Figure 18-1 shows the USB PD system block diagram.

Figure 18-1. USB PD Block Diagram



18.1 ADC

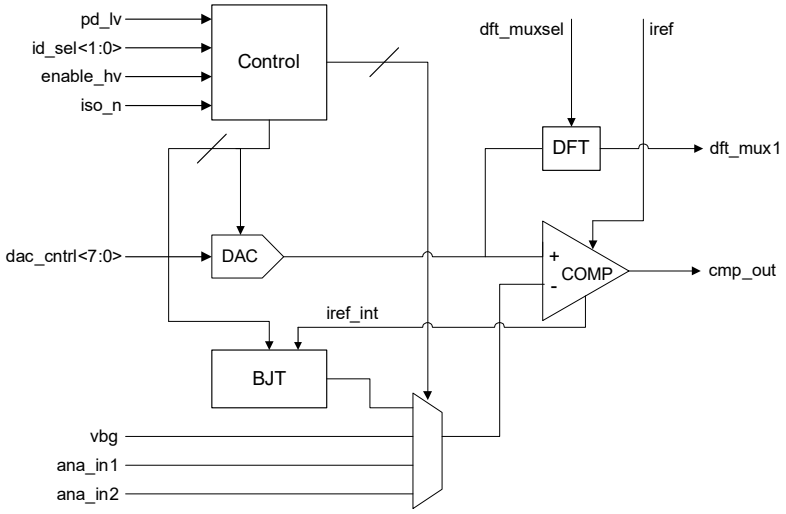
The USB PD block of EZ-PD™ PMG1-S2 MCU includes an 8-bit SAR ADC, which can also be used independently. Its contents are: control logic for level shifting and power down, 8-bit DAC, DFT switch, comparator (COMP), and a BJT NPN device. These blocks are referenced in Figure 18-2.

The DAC takes an 8-bit control signal and outputs a voltage referenced to the supply voltage. The comparator (COMP) compares the DAC voltage versus one of the following input voltages: either the amux_a or amux_b line, the bandgap voltage (vbg), or the voltage generated from sourcing current into a diode-connected BJT NPN device. If the DAC voltage potential is greater than the muxed voltage, then the comparator digital output is high.

The DFT block contains an isolating switch that connects the DAC output voltage to the chip's ADFT network. The switch is closed when the dft_muxsel signal is high.

For details on how to configure the ADC, refer to the USB PD component provided by the ModusToolbox (MTB) IDE.

Figure 18-2. Block Diagram of ADC Block



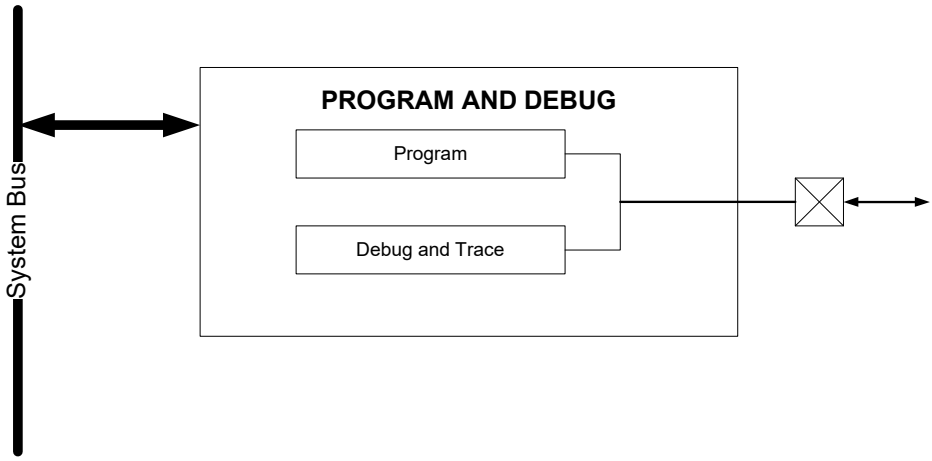
Section G: Program and Debug

This section encompasses the following chapters:

- Program and Debug Interface chapter on page 163
- Nonvolatile Memory Programming chapter on page 170

Top Level Architecture

Program and Debug Block Diagram



19. Program and Debug Interface

The EZ-PD™ PMG1-S2 MCU Program and Debug interface provides a communication gateway for an external device to perform programming or debugging. The external device can be a Cypress-supplied programmer and debugger, or a third-party device that supports EZ-PD™ PMG1-S2 MCU programming and debugging. The serial wire debug (SWD) interface is used as the communication protocol between the external device and EZ-PD™ PMG1-S2 MCU.

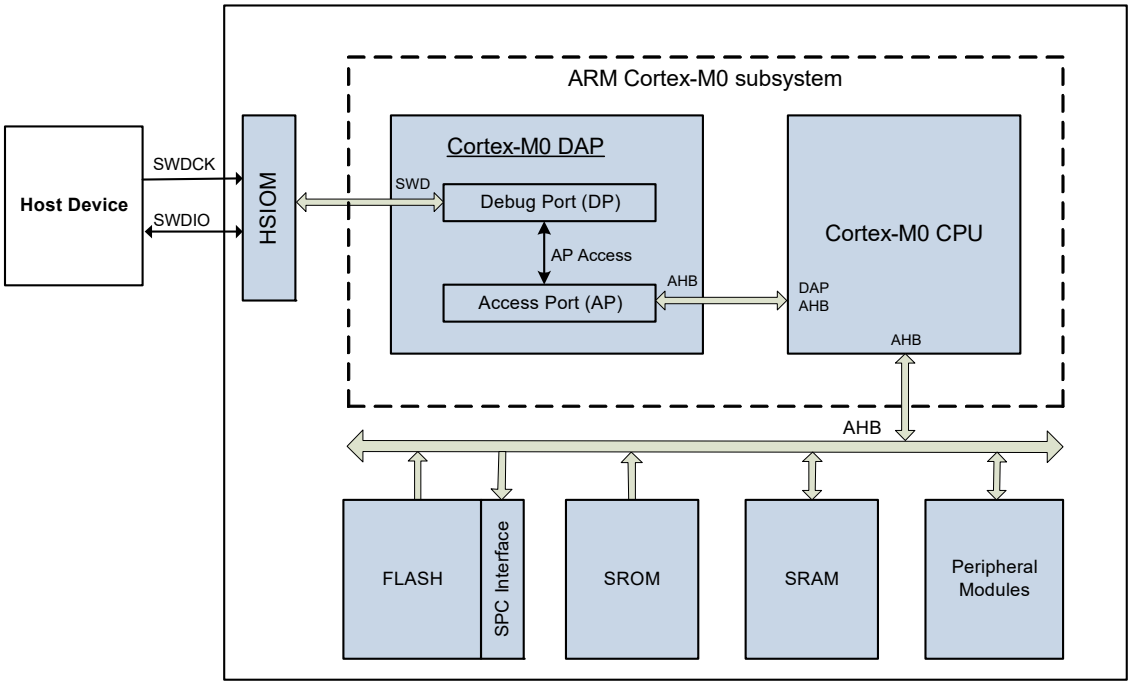
19.1 Features

- 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

19.2 Functional Description

Figure 19-1 shows the block diagram of the program and debug interface in EZ-PD™ PMG1-S2 MCU. 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 EZ-PD™ PMG1-S2 MCU "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 chapter on page 42 for details on HSIOM.

Figure 19-1. 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 EZ-PD™ PMG1-S2 MCU, which facilitates memory and peripheral register access by the AHB master. EZ-PD™ PMG1-S2 MCU has two AHB masters – Arm 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.

19.3 Serial Wire Debug (SWD) Interface

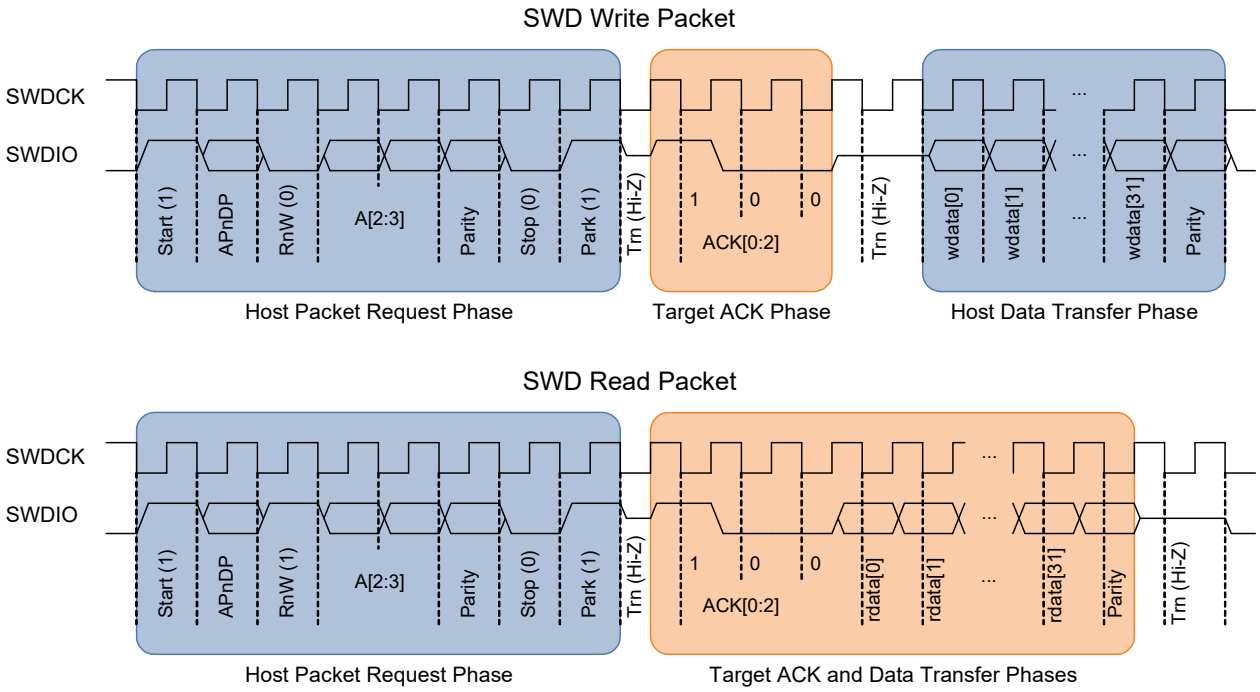
EZ-PD™ PMG1-S2 MCU's Cortex-M0 supports programming and debugging through the SWD interface. The SWD protocol is a packet-based serial transaction protocol. At the pin level, it uses a single bidirectional data signal (SWDIO) and a unidirectional clock signal (SWDCK). The host programmer always drives 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 EZ-PD™ PMG1-S2 MCU target.
- **Target Acknowledge Response Phase** – The EZ-PD™ PMG1-S2 MCU target sends an acknowledgement to the host.
- **Data Transfer Phase** – The host or target writes data to the bus, depending on the direction of the transfer.

When control of the SWDIO line passes from the host to the target, or vice versa, there is a turnaround period (Trn) where neither device drives the line and it floats in a high-impedance (Hi-Z) state. This period is either one-half or one and a half clock cycles, depending on the transition.

Figure 19-2 shows the timing diagrams of read and write SWD packets.

Figure 19-2. 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 19-3 and Table 19-4 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.

   If the parity bit is not correct, the header is ignored by EZ-PD™ PMG1-S2 MCU; there is no ACK response (ACK = 3b111). The programming operation should be aborted and retried again by following a device reset.

f. The stop bit is always logic 0.

g. The park bit is always logic 1.

2. Target Acknowledge Response Phase: SWDIO driven by the target
   a. The ACK[2:0] bits represent the target to host response, indicating failure or success, among other results. See Table 19-2 for definitions. **Note** ACK bits are transmitted with the LSB first.
3. Data Transfer Phase: SWDIO driven by either target or host depending on direction
   a. The data for read or write is written to the bus, LSB first.
   b. The data parity bit indicates the parity of the data read or written. It is an even parity; this means when XORed with the data bits, the result will be 0.

      If the parity bit indicates a data error, corrective action should be taken. For a read packet, if the host detects a parity error, it must abort the programming operation and restart. For a write packet, if the target detects a parity error, it generates a FAULT ACK response in the next packet.

According to the SWD protocol, the host can generate any number of SWDCK clock cycles between two packets with SWDIO low. 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.

19.3.1 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 19-1 and Figure 19-2 illustrate the timing of SWDIO bit writes and reads.

Table 19-1. SWDIO Bit Write and Read Timing

SWD Packet Phase	SWDIO Edge	
	Falling	Rising
Host Packet Request	Host Write	Target Read
Host Data Transfer		
Target Ack Response	Host Read	Target Write
Target Data Transfer		

19.3.2 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 19-2 shows the ACK bit-field decoding details.

Table 19-2. 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 EZ-PD™ PMG1-S2 MCU. But, the host must still send the data to be written to complete the packet. The parity bit corresponding to the data should also be sent by the host.
- For a WAIT response, it means that the EZ-PD™ PMG1-S2 MCU 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.

19.3.3 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 19-2. 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 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 EZ-PD™ PMG1-S2 MCU should drive the SWDIO line during the Trn period.

19.4 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*.

19.4.1 Debug Port (DP) Registers

Table 19-3 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 19-3. 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 0x0BB11477.
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 EZ-PD™ PMG1-S2 MCU, 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.

19.4.2 Access Port (AP) Registers

Table 19-4 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 19-4. 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 EZ-PD™ PMG1-S2 MCU 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

19.5 Programming the EZ-PD™ PMG1-S2 MCU Device

EZ-PD™ PMG1-S2 MCU is programmed using the following sequence. Refer to the EZ-PD™ PMG1 MCU Programming Specifications for complete details on the programming algorithm, timing specifications, and hardware configuration required for programming.

1. Acquire the SWD port in EZ-PD™ PMG1-S2 MCU.
2. Enter the programming mode.
3. Execute the device programming routines such as Silicon ID Check, Flash Programming, Flash Verification, and Checksum Verification.

19.5.1 SWD Port Acquisition

19.5.1.1 Primary and Secondary SWD Pin Pairs

The first step in device programming is to acquire the SWD port in EZ-PD™ PMG1-S2 MCU. Refer to the EZ-PD™ PMG1-S2 MCU device datasheet for information on SWD pins.

19.5.1.2 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 < 2 ms); //retry connection until OK ACK or timeout

if (time_elapsed >= 2 ms) return FAIL; //check for acquire time out

if (ID != CM0_ID) return FAIL; //confirm SWD ID of Cortex-M0 CPU. (0x0BB11477)
```

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 (2 ms) occurs. The SWD port is said to be in the acquired state if an OK ACK is received within the time window and the IDCODE read matches with that of the Cortex-M0 DAP.

19.5.2 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 EZ-PD™ PMG1 MCU Programming Specifications.

19.5.3 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 chapter on page 170. The exact sequence of calling the programming routines is given in the EZ-PD™ PMG1-S2 MCU Device Programming Specifications document.

19.6 EZ-PD™ PMG1-S2 MCU 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.

19.6.1 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 (CM0_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 (CM0_DFSR) – This register describes the reason a debug event has occurred. This includes debug events, which are caused by a CPU halt, breakpoint event, or watchpoint event.
- Debug Core Register Selector Register (CM0_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 (CM0_DCRDR) – This register is used to store the data to write to or read from the register selected in the CM0_DCRSR register.
- Debug Exception and Monitor Control Register (CM0_DEMCR) – This register contains the enable bits for global debug watchpoint (DWT) block enable, reset vector catch, and hard fault exception catch.

19.6.2 Breakpoint Unit (BPU)

The BPU provides breakpoint functionality on instruction fetches. The Cortex-M0 DAP in EZ-PD™ PMG1-S2 MCU 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 (CM0_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 EZ-PD™ PMG1-S2 MCU).
- Each hardware breakpoint has a Breakpoint Compare Register (CM0_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.

19.6.3 Data Watchpoint (DWT)

The DWT provides watchpoint support on a data address access or a program counter (PC) instruction address. Trace is not supported by the Cortex-M0 in EZ-PD™ PMG1-S2 MCU. 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 (CM0_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 (CM0_DWT_MASKx) registers store the ignore masks applied to the address range matching in the associated watchpoints.
- The watchpoint function (CM0_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 (CM0_DWT_PCSR) stores the current value of the program counter. This register is used for coarse, non-invasive profiling of the program counter register.

19.6.4 Debugging the EZ-PD™ PMG1-S2 MCU Device

The host debugs the target EZ-PD™ PMG1-S2 MCU device 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 (SWDP) 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 EZ-PD™ PMG1-S2 MCU device 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 chapter on page 42 to understand how to configure the SWD port pins, allowing them to be used only for SWD interface or for other functions such as 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 chapter on page 63. 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.

19.7 Registers

Table 19-5. List of Registers

Register Name	Description
CM0_DHCSR	Debug Halting Control and Status Register
CM0_DFSR	Debug Fault Status Register
CM0_DCRSR	Debug Core Register Selector Register
CM0_DCRDR	Debug Core Register Data Register
CM0_DEMCR	Debug Exception and Monitor Control Register
CM0_BP_CTRL	Breakpoint control register
CM0_BP_COMPx	Breakpoint Compare Register
CM0_DWT_COMPx	Watchpoint Compare Register
CM0_DWT_MASKx	Watchpoint Mask Register
CM0_DWT_FUNCTIONx	Watchpoint Function Register
CM0_DWT_PCSR	Watchpoint Comparator PC Sample Register

20. Nonvolatile Memory Programming

Nonvolatile memory programming refers to the programming of flash memory in the EZ-PD™ PMG1-S2 MCU device. This chapter explains the different functions that are part of device programming, such as erase, write, program, and checksum calculation. Cypress-supplied programmers and other third-party programmers can use these functions to program the EZ-PD™ PMG1-S2 MCU 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.

20.1 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

20.2 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.

EZ-PD™ PMG1-S2 MCU 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 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 EZ-PD™ PMG1-S2 MCU using the SWD protocol by sending the commands to the Debug and Access Port (DAP). The programming sequence for the EZ-PD™ PMG1-S2 MCU device with an external programmer is given in the EZ-PD™ PMG1-S2 MCU Device 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 chapter on page 61) 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.

20.3 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.

20.4 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.

20.4.1 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_SYSREG 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_SYSREG 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 20-1 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_SYSREG and CPUSS_SYSARG registers to fetch the values returned.

20.5 System Calls

Table 20-1 lists all the system calls supported in EZ-PD™ PMG1-S2 MCU along with the function description and availability in device protection modes. See the Device Security chapter on page 63 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 20-1. 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	✔	✔	–	✔
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	✔	–	–	✔
Write Row	Erases and then programs a row of flash with data in the page latch buffer	✔	–	–	✔
Program Row	Programs a row of flash with data in the page latch buffer	✔	–	–	✔
Erase All	Erases all user code in the flash array; the flash row-level protection data in the supervisory flash area	✔	–	–	
Checksum	Calculates the checksum over the entire flash memory (user and supervisory area) or checksums a single row of flash	✔	✔	–	✔
Write Protection	This programs both flash row-level protection settings and chip-level protection settings into the supervisory flash (row 0)	✔	✔	–	
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	–	–	–	✔
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	–	–	–	✔
Resume Non-Blocking	Resumes a non-blocking write row or non-blocking program row. This function is meant only for CPU access	–	–	–	✔

20.5.1 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	See the device datasheet for Silicon ID values for different part numbers
Bits [15:8]	Silicon ID Hi	
Bits [19:16]	Minor Revision Id	See the EZ-PD™ PMG1 MCU 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 0xAD for EZ-PD™ PMG1-S2 MCU
Bits [15:12]	Chip Protection	See the Device Security chapter on page 63
Bits [31:16]	0xXXXX	Not used

20.5.2 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 HF clock (clk_hf) are set to IMO at 48 MHz before 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.

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]	0xE8	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]	0x0015	Configure clock 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)

20.5.3 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 64 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 0x80 - Byte 128 of latch buffer
Bits [31:24]	Flash Macro Select	0x00 – Flash Macro 0 0x01 – Flash Macro 1 (Refer to the Cortex-M0 CPU chapter on page 24 for the number of flash macros in the device)
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)

20.5.4 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 HF clock (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.

Note that the SROM does not modify, enable, or disable any clock during any flash operation. Refer to the CLK_IMO_CONFIG register in the EZ-PD™ PMG1-S2 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 0x00FF - Row 255
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)

20.5.5 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. This 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 HF clock (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 0x00FF - Row 255
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)

20.5.6 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 HF clock (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)

20.5.7 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

20.5.8 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 HF clock (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 0x01 – Flash Macro 1
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	

20.5.9 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 171.

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.

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 HF clock (clk_hf) are set to IMO at 48 MHz.

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

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 0x00FF - Row 255
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)

20.5.10 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 171. 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 HF clock (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 0x00FF - Row 255
CPUSS_SYSARG register		

Address	Value to be Written	Description
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)

20.5.11 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 171.

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)

20.6 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 20-2. 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 API’s 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.

20.7 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.

```
#include "cy_device.h"
#include "cybsp.h"

// Flash row size
#define FLASH_ROW_SIZE (CY_FLASH_SIZEOF_ROW)

/* Keys used in SROM APIs. */
#define SROM_KEY_ONE (0xB6u)
#define SROM_KEY_TWO_OFFSET (8u)
#define SROM_PARAM_OFFSET (16u)

// Offset of Argument 1, 2 and data for SROM APIs in SRAM buffer.
#define SROM_API_ARG0_OFFSET (0x00u)
#define SROM_API_ARG1_OFFSET (0x01u)
#define SROM_API_DATA_OFFSET (0x02u)

// CPUSS SYSARG return value
#define CPUSS_SYSARG_RETURN_VALUE_MASK (0xF0000000u) // Mask
#define CPUSS_SYSARG_PASS_RETURN_VALUE (0xA0000000u) // Success
#define CPUSS_SYSARG_ERROR_RETURN_VALUE (0xF0000000u) // Error
#define ROW_NUMBER_TO_WRITE (255u)

// Buffer used for SROM APIs
uint32_t srom_arg_buf[(FLASH_ROW_SIZE / sizeof(uint32_t)) + 2];
```

```
// Data to write into flash
uint8_t data_to_write[FLASH_ROW_SIZE];
// Counter to keep track of the SPC interrupts during Non-Blocking Flash update.
static volatile uint8_t spc_intr_counter = 0;

// This example uses the simplest way to place a function in the RAM space; functions
// are mapped to .data section that is normally used for initialized variables.
// Code is tested with the GCC compiler.
void spc_interrupt_handler(void) __attribute__ ((section(".data")));
uint32_t non_blocking_flash_row_write(uint16_t row_num) __attribute__((section(".data")));
bool load_flash_bytes(uint8_t *data);
bool configure_clock(void);

//SPC interrupt configuration
const cy_stc_sysint_t spc_int_config =
{
    .intrSrc = (IRQn_Type)cpuss_interrupt_spcif_IRQn,
    .intrPriority = 0U,
};

//--------------------------------------------------------
int main()
{
    bool ret;

    // Create a dummy buffer with known data
    for(uint32_t i = 0u; i < FLASH_ROW_SIZE; i++)
    {
        data_to_write[i] = i + 2;
   }

   // Assign SPC interrupt handler
   Cy_SysInt_Init(&spc_int_config, &spc_interrupt_handler);

   // Enable interrupts
   __enable_irq();
   NVIC_EnableIRQ(spc_int_config.intrSrc);

   // Write to Flash; you should analyze each return value before proceeding to next step
   ret = load_flash_bytes(data_to_write);
   ret = configure_clock();
   ret = non_blocking_flash_row_write(ROW_NUMBER_TO_WRITE);
   /* Loop forever */
   for(;;)
   {
   }
}

//--------------------------------------------------------
// load_flash_bytes
//
// This function loads the page latch buffer with data to be programmed into a row of
// flash. Data loaded into the page latch buffer remains until a program operation
// is performed, which clears the page latch contents.
//--------------------------------------------------------
#define SROM_LOAD_FLASH_API_OPCODE (0x04u)
#define SROM_LOAD_FLASH_KEY_TWO (0xD7u)
#define SROM_LATCH_BUFFER_ADDR (0x00u)
```

```
bool load_flash_bytes(uint8_t *data)
{
   uint32_t ret;
   // Fill in the arguments for API.
   srom_arg_buf[SROM_API_ARG0_OFFSET] = (uint32_t)((SROM_LATCH_BUFFER_ADDR <<
   SROM_PARAM_OFFSET)|(SROM_LOAD_FLASH_KEY_TWO << SROM_KEY_TWO_OFFSET)|SROM_KEY_ONE);
   // Number of bytes to flash - 1 (e.g., 0x00 - 1 byte, 0xFF - 256 bytes)
   srom_arg_buf[SROM_API_ARG1_OFFSET] = FLASH_ROW_SIZE-1;
   // Copy data to write in a temp buf which is eventually passed to SROM API.
   memcpy ((void *)&srom_arg_buf[SROM_API_DATA_OFFSET], data, FLASH_ROW_SIZE);
   //Set SYSARG, SYSREQ
   CPUSS_SYSARG = (uint32_t)&srom_arg_buf[SROM_API_ARG0_OFFSET];
   CPUSS_SYSREQ = (CPUSS_SYSREQ_SYSCALL_REQ_Msk | SROM_LOAD_FLASH_API_OPCODE);
   __asm("NOP\n");
   // Read the result and provide return.
   ret = CPUSS_SYSARG;
   return ((ret & CPUSS_SYSARG_RETURN_VALUE_MASK) == CPUSS_SYSARG_PASS_RETURN_VALUE);
}

//-------------------------------------------------------------
// 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 and the HF clock
// are set to IMO at 48 MHz before calling the flash write and flash erase APIs.
//-------------------------------------------------------------
#define SROM_CONFIGURE_CLOCK_API_OPCODE (0x15u)
#define SROM_CONFIGURE_CLOCK_KEY_TWO (0xE8)
bool configure_clock(void)
{
   uint32_t ret;
   //Set SYSARG, SYSREQ
   CPUSS_SYSARG = (uint32_t)((SROM_CONFIGURE_CLOCK_KEY_TWO << SROM_KEY_TWO_OFFSET) |
   SROM_KEY_ONE);
   CPUSS_SYSREQ = (CPUSS_SYSREQ_SYSCALL_REQ_Msk | SROM_CONFIGURE_CLOCK_API_OPCODE);
   __asm("NOP\n");
   ret = CPUSS_SYSARG;
   return ((ret & CPUSS_SYSARG_RETURN_VALUE_MASK) == CPUSS_SYSARG_PASS_RETURN_VALUE);
}

//-------------------------------------------------------------
// non_blocking_flash_row_write
//
// This function writes a flash row in a non-blocking manner, so that the CPU can execute
// code from SRAM while the write operation is being done. Call the Load Flash Bytes
// function before calling this function, to load the data bytes that will be used for
// programming the row. Call the Configure Clock API before calling this function.
//-------------------------------------------------------------
#define SROM_NB_FLASH_ROW_API_OPCODE (0x07u)
#define SROM_NB_FLASH_ROW_NUM_KEY_TWO (0xDAu)
uint32_t non_blocking_flash_row_write(uint16_t row_num)
{
   uint32_t ret;
   /* This command results in three SPC interrupts. Reset the counter. */
   spc_intr_counter = 0;
   /* Arguments */
   srom_arg_buf[SROM_API_ARG0_OFFSET] = (uint32_t)((row_num << SROM_PARAM_OFFSET)|
   (SROM_NB_FLASH_ROW_NUM_KEY_TWO << SROM_KEY_TWO_OFFSET)|SROM_KEY_ONE);
```

```
   //Set SYSARG, SYSREQ
   CPUSS_SYSARG = (uint32_t)&srom_arg_buf[SROM_API_ARG0_OFFSET];
   CPUSS_SYSREQ = (CPUSS_SYSREQ_SYSCALL_REQ_Msk | SROM_NB_FLASH_ROW_API_OPCODE);
   __asm("NOP\n");
   // The non-blocking write row system call has three phases. Execute user code until
   // three SPC interrupts have occurred.
   while (spc_intr_counter < 3)
   {
      // PUT YOUR CODE THAT WILL BE EXECUTED DURING FLASH WRITE HERE
   }
   ret = CPUSS_SYSARG;
   return ((ret & CPUSS_SYSARG_RETURN_VALUE_MASK) == CPUSS_SYSARG_PASS_RETURN_VALUE);
}

// Interrupt handler that calls the Resume Non-Blocking function, which is required for
// completion of the non-blocking write operation.
// This function is executed from RAM as it is executed during a Non-Blocking System Call.
#define SROM_RESUME_NB_API_OPCODE (0x09u)
#define SROM_RESUME_NB_KEY_TWO (0xDCu)
void spc_interrupt_handler(void)
{
   // Execute Resume Non-Blocking API Syscall
   // Arguments
   srom_arg_buf[SROM_API_ARG0_OFFSET] = (uint32_t)((SROM_RESUME_NB_KEY_TWO <<
   SROM_KEY_TWO_OFFSET)|SROM_KEY_ONE);
   //Set SYSARG and SYSREQ
   CPUSS_SYSARG = (uint32_t)&srom_arg_buf[SROM_API_ARG0_OFFSET];
   CPUSS_SYSREQ = (CPUSS_SYSREQ_SYSCALL_REQ_Msk | SROM_RESUME_NB_API_OPCODE);
   spc_intr_counter++;
}
```

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 chapter on page 29 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.

Glossary

The Glossary section explains the terminology used in this technical reference manual. Glossary terms are characterized in ***bold, italic font*** throughout the text of this manual.

A

accumulator	In a CPU, a register in which intermediate results are stored. Without an accumulator, it is necessary to write the result of each calculation (addition, subtraction, shift, and so on.) to main memory and read them back. Access to main memory is slower than access to the accumulator, which usually has direct paths to and from the arithmetic and logic unit (ALU).
active high	1. A logic signal having its asserted state as the logic 1 state. 2. A logic signal having the logic 1 state as the higher voltage of the two states.
active low	1. A logic signal having its asserted state as the logic 0 state. 2. A logic signal having its logic 1 state as the lower voltage of the two states: inverted logic.
address	The label or number identifying the memory location (RAM, ROM, or register) where a unit of information is stored.
algorithm	A procedure for solving a mathematical problem in a finite number of steps that frequently involve repetition of an operation.
ambient temperature	The temperature of the air in a designated area, particularly the area surrounding the device.
analog	See ***analog signals***.
analog blocks	The basic programmable opamp circuits. These are SC (switched capacitor) and CT (continuous time) blocks. These blocks can be interconnected to provide ADCs, DACs, multi-pole filters, gain stages, and much more.
analog output	An output that is capable of driving any voltage between the supply rails, instead of just a logic 1 or logic 0.
analog signals	A signal represented in a continuous form with respect to continuous times, as contrasted with a digital signal represented in a discrete (discontinuous) form in a sequence of time.
analog-to-digital (ADC)	A device that changes an analog signal to a digital signal of corresponding magnitude. Typically, an ADC converts a voltage to a digital number. The *digital-to-analog (DAC)* converter performs the reverse operation.

AND See *Boolean Algebra*.

API (Application Programming Interface) A series of software routines that comprise an interface between a computer application and lower-level services and functions (for example, user modules and libraries). APIs serve as building blocks for programmers that create software applications.

array An array, also known as a vector or list, is one of the simplest data structures in computer programming. Arrays hold a fixed number of equally-sized data elements, generally of the same data type. Individual elements are accessed by index using a consecutive range of integers, as opposed to an associative array. Most high-level programming languages have arrays as a built-in data type. Some arrays are multi-dimensional, meaning they are indexed by a fixed number of integers; for example, by a group of two integers. One- and two-dimensional arrays are the most common. Also, an array can be a group of capacitors or resistors connected in some common form.

assembly A symbolic representation of the machine language of a specific processor. Assembly language is converted to machine code by an assembler. Usually, each line of assembly code produces one machine instruction, though the use of macros is common. Assembly languages are considered low-level languages; where as C is considered a high-level language.

asynchronous A signal whose data is acknowledged or acted upon immediately, irrespective of any clock signal.

attenuation The decrease in intensity of a signal as a result of absorption of energy and of scattering out of the path to the detector, but not including the reduction due to geometric spreading. Attenuation is usually expressed in dB.

B

bandgap reference A stable voltage reference design that matches the positive temperature coefficient of V_T with the negative temperature coefficient of V_{BE}, to produce a zero temperature coefficient (ideally) reference.

bandwidth

1. The frequency range of a message or information processing system measured in hertz.
2. The width of the spectral region over which an amplifier (or absorber) has substantial gain (or loss); it is sometimes represented more specifically as, for example, full width at half maximum.

bias

1. A systematic deviation of a value from a reference value.
2. The amount by which the average of a set of values departs from a reference value.
3. The electrical, mechanical, magnetic, or other force (field) applied to a device to establish a reference level to operate the device.

bias current The constant low-level DC current that is used to produce a stable operation in amplifiers. This current can sometimes be changed to alter the bandwidth of an amplifier.

binary The name for the base 2 numbering system. The most common numbering system is the base 10 numbering system. The base of a numbering system indicates the number of values that may exist for a particular positioning within a number for that system. For example, in base 2, binary, each position may have one of two values (0 or 1). In the base 10, decimal, numbering system, each position may have one of ten values (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9).

bit A single digit of a binary number. Therefore, a bit may only have a value of '0' or '1'. A group of 8 bits is called a byte. Because the EZ-PD™ PMG1-S2 MCU's M8CP is an 8-bit microcontroller, the EZ-PD™ PMG1-S2 MCU devices' native data chunk size is a byte.

bit rate (BR) The number of bits occurring per unit of time in a bit stream, usually expressed in bits per second (bps).

block
1. A functional unit that performs a single function, such as an oscillator.
2. A functional unit that may be configured to perform one of several functions, such as a digital EZ-PD™ PMG1-S2 MCU block or an analog EZ-PD™ PMG1-S2 MCU block.

Boolean Algebra In mathematics and computer science, Boolean algebras or Boolean lattices, are algebraic structures which "capture the essence" of the logical operations AND, OR and NOT as well as the set theoretic operations union, intersection, and complement. Boolean algebra also defines a set of theorems that describe how Boolean equations can be manipulated. For example, these theorems are used to simplify Boolean equations, which will reduce the number of logic elements needed to implement the equation.

The operators of Boolean algebra may be represented in various ways. Often they are simply written as AND, OR, and NOT. In describing circuits, NAND (NOT AND), NOR (NOT OR), XNOR (exclusive NOT OR), and XOR (exclusive OR) may also be used. Mathematicians often use + (for example, A+B) for OR and • for AND (for example, A*B) (in some ways those operations are analogous to addition and multiplication in other algebraic structures) and represent NOT by a line drawn above the expression being negated (for example, ~A, A_, !A).

break-before-make The elements involved go through a disconnected state entering ('break") before the new connected state ("make").

broadcast net A signal that is routed throughout the microcontroller and is accessible by many blocks or systems.

buffer
1. A storage area for data that is used to compensate for a speed difference, when transferring data from one device to another. Usually refers to an area reserved for I/O operations, into which data is read, or from which data is written.
2. A portion of memory set aside to store data, often before it is sent to an external device or as it is received from an external device.
3. An amplifier used to lower the output impedance of a system.

bus
1. A named connection of nets. Bundling nets together in a bus makes it easier to route nets with similar routing patterns.
2. A set of signals performing a common function and carrying similar data. Typically represented using vector notation; for example, address[7:0].
3. One or more conductors that serve as a common connection for a group of related devices.

byte A digital storage unit consisting of 8 bits.

C

C	A high-level programming language.
capacitance	A measure of the ability of two adjacent conductors, separated by an insulator, to hold a charge when a voltage differential is applied between them. Capacitance is measured in units of Farads.
capture	To extract information automatically through the use of software or hardware, as opposed to hand-entering of data into a computer file.
chaining	Connecting two or more 8-bit digital blocks to form 16-, 24-, and even 32-bit functions. Chaining allows certain signals such as Compare, Carry, Enable, Capture, and Gate to be produced from one block to another.
checksum	The checksum of a set of data is generated by adding the value of each data word to a sum. The actual checksum can simply be the result sum or a value that must be added to the sum to generate a pre-determined value.
clear	To force a bit/register to a value of logic ‘0’.
clock	The device that generates a periodic signal with a fixed frequency and duty cycle. A clock is sometimes used to synchronize different logic blocks.
clock generator	A circuit that is used to generate a clock signal.
CMOS	The logic gates constructed using *MOS* transistors connected in a complementary manner. CMOS is an acronym for complementary metal-oxide semiconductor.
comparator	An electronic circuit that produces an output voltage or current whenever two input levels simultaneously satisfy predetermined amplitude requirements.
compiler	A program that translates a high-level language, such as C, into machine language.
configuration	In a computer system, an arrangement of functional units according to their nature, number, and chief characteristics. Configuration pertains to hardware, software, firmware, and documentation. The configuration will affect system performance.
configuration space	In EZ-PD™ PMG1-S2 MCU devices, the register space accessed when the XIO bit, in the CPU_F register, is set to ‘1’.
crowbar	A type of over-voltage protection that rapidly places a low-resistance shunt (typically an SCR) from the signal to one of the power supply rails, when the output voltage exceeds a predetermined value.
CPUSS	CPU subsystem
crystal oscillator	An oscillator in which the frequency is controlled by a piezoelectric crystal. Typically a piezoelectric crystal is less sensitive to ambient temperature than other circuit components.

cyclic redundancy check (CRC)	A calculation used to detect errors in data communications, typically performed using a linear feedback shift register. Similar calculations may be used for a variety of other purposes such as data compression.

D

data bus	A bi-directional set of signals used by a computer to convey information from a memory location to the central processing unit and vice versa. More generally, a set of signals used to convey data between digital functions.
data stream	A sequence of digitally encoded signals used to represent information in transmission.
data transmission	Sending data from one place to another by means of signals over a channel.
debugger	A hardware and software system that allows the user to analyze the operation of the system under development. A debugger usually allows the developer to step through the firmware one step at a time, set break points, and analyze memory.
dead band	A period of time when neither of two or more signals are in their active state or in transition.
decimal	A base-10 numbering system, which uses the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 (called digits) together with the decimal point and the sign symbols + (plus) and - (minus) to represent numbers.
default value	Pertaining to the pre-defined initial, original, or specific setting, condition, value, or action a system will assume, use, or take in the absence of instructions from the user.
device	The device referred to in this manual is the EZ-PD™ PMG1-S2 MCU device, unless otherwise specified.
die	An non-packaged integrated circuit (IC), normally cut from a wafer.
digital	A signal or function, the amplitude of which is characterized by one of two discrete values: ‘0’ or ‘1’.
digital blocks	The 8-bit logic blocks that can act as a counter, timer, serial receiver, serial transmitter, CRC generator, pseudo-random number generator, or SPI.
digital logic	A methodology for dealing with expressions containing two-state variables that describe the behavior of a circuit or system.
digital-to-analog (DAC)	A device that changes a digital signal to an analog signal of corresponding magnitude. The *analog-to-digital (ADC)* converter performs the reverse operation.
direct access	The capability to obtain data from a storage device, or to enter data into a storage device, in a sequence independent of their relative positions by means of addresses that indicate the physical location of the data.
duty cycle	The relationship of a clock period *high time* to its *low time*, expressed as a percent.

E

External Reset (XRES_N) An active high signal that is driven into the EZ-PD™ PMG1-S2 MCU device. It causes all operation of the CPU and blocks to stop and return to a pre-defined state.

F

falling edge A transition from a logic 1 to a logic 0. Also known as a negative edge.

feedback The return of a portion of the output, or processed portion of the output, of a (usually active) device to the input.

filter A device or process by which certain frequency components of a signal are attenuated.

firmware The software that is embedded in a hardware device and executed by the CPU. The software may be executed by the end user, but it may not be modified.

flag Any of various types of indicators used for identification of a condition or event (for example, a character that signals the termination of a transmission).

Flash An electrically programmable and erasable, *volatile* technology that provides users with the programmability and data storage of EPROMs, plus in-system erasability. Nonvolatile means that the data is retained when power is off.

Flash bank A group of flash ROM blocks where flash block numbers always begin with ‘0’ in an individual flash bank. A flash bank also has its own block level protection information.

Flash block The smallest amount of flash ROM space that may be programmed at one time and the smallest amount of flash space that may be protected. A flash block holds 64 bytes.

flip-flop A device having two stable states and two input terminals (or types of input signals) each of which corresponds with one of the two states. The circuit remains in either state until it is made to change to the other state by application of the corresponding signal.

frequency The number of cycles or events per unit of time, for a periodic function.

G

gain The ratio of output current, voltage, or power to input current, voltage, or power, respectively. Gain is usually expressed in dB.

gate

1. A device having one output channel and one or more input channels, such that the output channel state is completely determined by the input channel states, except during switching transients.
2. One of many types of combinational logic elements having at least two inputs (for example, AND, OR, NAND, and NOR (also see *Boolean Algebra*)).

ground

1. The electrical neutral line having the same potential as the surrounding earth.
2. The negative side of DC power supply.
3. The reference point for an electrical system.
4. The conducting paths between an electric circuit or equipment and the earth, or some conducting body serving in place of the earth.

H

hardware

A comprehensive term for all of the physical parts of a computer or embedded system, as distinguished from the data it contains or operates on, and the software that provides instructions for the hardware to accomplish tasks.

hardware reset

A reset that is caused by a circuit, such as a POR, watchdog reset, or external reset. A hardware reset restores the state of the device as it was when it was first powered up. Therefore, all registers are set to the POR value as indicated in register tables throughout this document.

hexadecimal

A base 16 numeral system (often abbreviated and called hex), usually written using the symbols 0-9 and A-F. It is a useful system in computers because there is an easy mapping from four bits to a single hex digit. Thus, one can represent every byte as two consecutive hexadecimal digits. Compare the binary, hex, and decimal representations:

bin = hex = dec

0000b = 0x0 = 0

0001b = 0x1 = 1

0010b = 0x2 = 2

...

1001b = 0x9 = 9

1010b = 0xA = 10

1011b = 0xB = 11

...

1111b = 0xF = 15

So the decimal numeral 79 whose binary representation is 0100 1111b can be written as 4Fh in hexadecimal (0x4F).

high time

The amount of time the signal has a value of ‘1’ in one period, for a periodic digital signal.

I

I^2C

A two-wire serial computer bus by Phillips Semiconductors (now NXP Semiconductors). I^2C is an Inter-Integrated Circuit. It is used to connect low-speed peripherals in an embedded system. The original system was created in the early 1980s as a battery control interface, but it was later used as a simple internal bus system for building control electronics. I^2C uses only two bidirectional pins, clock and data, both running at +5 V and pulled high with resistors. The bus operates at 100 Kbps in standard mode and 400 Kbps in fast mode.

idle state A condition that exists whenever user messages are not being transmitted, but the service is immediately available for use.

impedance
1. The resistance to the flow of current caused by resistive, capacitive, or inductive devices in a circuit.
2. The total passive opposition offered to the flow of electric current. Note the impedance is determined by the particular combination of resistance, inductive reactance, and capacitive reactance in a given circuit.

input A point that accepts data, in a device, process, or channel.

input/output (I/O) A device that introduces data into or extracts data from a system.

instruction An expression that specifies one operation and identifies its operands, if any, in a programming language such as C or assembly.

instruction mnemonics A set of acronyms that represent the opcodes for each of the assembly-language instructions, for example, ADD, SUBB, MOV.

integrated circuit (IC) A device in which components such as resistors, capacitors, diodes, and *transistors* are formed on the surface of a single piece of semiconductor.

interface The means by which two systems or devices are connected and interact with each other.

interrupt A suspension of a process, such as the execution of a computer program, caused by an event external to that process, and performed in such a way that the process can be resumed.

interrupt service routine (ISR) A block of code that normal code execution is diverted to when the M8CP receives a hardware interrupt. Many interrupt sources may each exist with its own priority and individual ISR code block. Each ISR code block ends with the RETI instruction, returning the device to the point in the program where it left normal program execution.

J

jitter
1. A misplacement of the timing of a transition from its ideal position. A typical form of corruption that occurs on serial data streams.
2. The abrupt and unwanted variations of one or more signal characteristics, such as the interval between successive pulses, the amplitude of successive cycles, or the frequency or phase of successive cycles.

L

latency The time or delay that it takes for a signal to pass through a given circuit or network.

least significant bit (LSb) The binary digit, or bit, in a binary number that represents the least significant value (typically the right-hand bit). The bit versus byte distinction is made by using a lower case “b” for bit in LSb.

least significant byte (LSB) The byte in a multi-byte word that represents the least significant values (typically the right-hand byte). The byte versus bit distinction is made by using an upper case “B” for byte in LSB.

Linear Feedback Shift Register (LFSR) A shift register whose data input is generated as an *XOR* of two or more elements in the register chain.

load The electrical demand of a process expressed as power (watts), current (amps), or resistance (ohms).

logic function A mathematical function that performs a digital operation on digital data and returns a digital value.

lookup table (LUT) A logic block that implements several logic functions. The logic function is selected by means of select lines and is applied to the inputs of the block. For example: A 2 input LUT with four select lines can be used to perform any one of 16 logic functions on the two inputs resulting in a single logic output. The LUT is a combinational device; therefore, the input/output relationship is continuous, that is, not sampled.

low time The amount of time the signal has a value of ‘0’ in one period, for a periodic digital signal.

low-voltage detect (LVD) A circuit that senses V_{DDD} and provides an interrupt to the system when V_{DDD} falls below a selected threshold.

M

M8CP An 8-bit Harvard Architecture microprocessor. The microprocessor coordinates all activity inside a EZ-PD™ PMG1-S2 MCU device by interfacing to the flash, SRAM, and register space.

macro A programming language macro is an abstraction, whereby a certain textual pattern is replaced according to a defined set of rules. The interpreter or compiler automatically replaces the macro instance with the macro contents when an instance of the macro is encountered. Therefore, if a macro is used five times and the macro definition required 10 bytes of code space, 50 bytes of code space will be needed in total.

mask

1. To obscure, hide, or otherwise prevent information from being derived from a signal. It is usually the result of interaction with another signal, such as noise, static, jamming, or other forms of interference.
2. A pattern of bits that can be used to retain or suppress segments of another pattern of bits, in computing and data processing systems.

master device	A device that controls the timing for data exchanges between two devices. Or when devices are cascaded in width, the master device is the one that controls the timing for data exchanges between the cascaded devices and an external interface. The controlled device is called the *slave device*.
microcontroller	An integrated circuit device that is designed primarily for control systems and products. In addition to a CPU, a microcontroller typically includes memory, timing circuits, and I/O circuitry. The reason for this is to permit the realization of a controller with a minimal quantity of devices, thus achieving maximal possible miniaturization. This in turn, will reduce the volume and the cost of the controller. The microcontroller is normally not used for general-purpose computation as is a microprocessor.
mnemonic	A tool intended to assist the memory. Mnemonics rely on not only repetition to remember facts, but also on creating associations between easy-to-remember constructs and lists of data. A two to four character string representing a microprocessor instruction.
mode	A distinct method of operation for software or hardware. For example, the Digital EZ-PD™ PMG1-S2 MCU block may be in either counter mode or timer mode.
modulation	A range of techniques for encoding information on a carrier signal, typically a sine-wave signal. A device that performs modulation is known as a modulator.
Modulator	A device that imposes a signal on a carrier.
MOS	An acronym for metal-oxide semiconductor.
most significant bit (MSb)	The binary digit, or bit, in a binary number that represents the most significant value (typically the left-hand bit). The bit versus byte distinction is made by using a lower case “b” for bit in MSb.
most significant byte (MSB)	The byte in a multi-byte word that represents the most significant values (typically the left-hand byte). The byte versus bit distinction is made by using an upper case “B” for byte in MSB.
multiplexer (mux)	1. A logic function that uses a binary value, or address, to select between a number of inputs and conveys the data from the selected input to the output. 2. A technique which allows different input (or output) signals to use the same lines at different times, controlled by an external signal. Multiplexing is used to save on wiring and I/O ports.

N

NAND	See *Boolean Algebra*.
negative edge	A transition from a logic 1 to a logic 0. Also known as a falling edge.
net	The routing between devices.
nibble	A group of four bits, which is one-half of a byte.
noise	1. A disturbance that affects a signal and that may distort the information carried by the signal. 2. The random variations of one or more characteristics of any entity such as voltage, current, or data.

NOR See *Boolean Algebra*.

NOT See *Boolean Algebra*.

O

OR See *Boolean Algebra*.

oscillator A circuit that may be crystal controlled and is used to generate a clock frequency.

output The electrical signal or signals which are produced by an analog or digital block.

P

parallel The means of communication in which digital data is sent multiple bits at a time, with each simultaneous bit being sent over a separate line.

parameter Characteristics for a given block that have either been characterized or may be defined by the designer.

parameter block A location in memory where parameters for the SSC instruction are placed before execution.

parity A technique for testing transmitting data. Typically, a binary digit is added to the data to make the sum of all the digits of the binary data either always even (even parity) or always odd (odd parity).

path
1. The logical sequence of instructions executed by a computer.
2. The flow of an electrical signal through a circuit.

pending interrupts An interrupt that is triggered but not serviced, either because the processor is busy servicing another interrupt or global interrupts are disabled.

phase The relationship between two signals, usually the same frequency, that determines the delay between them. This delay between signals is either measured by time or angle (degrees).

pin A terminal on a hardware component. Also called lead.

pinouts The pin number assignment: the relation between the logical inputs and outputs of the EZ-PD™ PMG1-S2 MCU device and their physical counterparts in the printed circuit board (PCB) package. Pinouts will involve pin numbers as a link between schematic and PCB design (both being computer generated files) and may also involve pin names.

port A group of pins, usually eight.

positive edge A transition from a logic 0 to a logic 1. Also known as a rising edge.

posted interrupts An interrupt that is detected by the hardware but may or may not be enabled by its mask bit. Posted interrupts that are not masked become pending interrupts.

Power On Reset (POR)	A circuit that forces the EZ-PD™ PMG1-S2 MCU device to reset when the voltage is below a pre-set level. This is one type of *hardware reset*.
program counter	The instruction pointer (also called the program counter) is a register in a computer processor that indicates where in memory the CPU is executing instructions. Depending on the details of the particular machine, it holds either the address of the instruction being executed, or the address of the next instruction to be executed.
protocol	A set of rules. Particularly the rules that govern networked communications.
pulse	A rapid change in some characteristic of a signal (for example, phase or frequency), from a base-line value to a higher or lower value, followed by a rapid return to the baseline value.
pulse width modulator (PWM)	An output in the form of duty cycle which varies as a function of the applied measure.

R

RAM	An acronym for random access memory. A data-storage device from which data can be read out and new data can be written in.
register	A storage device with a specific capacity, such as a bit or byte.
reset	A means of bringing a system back to a know state. See *hardware reset* and *software reset*.
resistance	The resistance to the flow of electric current measured in ohms for a conductor.
revision ID	A unique identifier of the EZ-PD™ PMG1-S2 MCU device.
ripple divider	An asynchronous ripple counter constructed of flip-flops. The clock is fed to the first stage of the counter. An n-bit binary counter consisting of n flip-flops that can count in binary from 0 to 2^n - 1.
rising edge	*See positive edge*.
ROM	An acronym for read only memory. A data-storage device from which data can be read out, but new data cannot be written in.
routine	A block of code, called by another block of code, that may have some general or frequent use.
routing	Physically connecting objects in a design according to design rules set in the reference library.
runt pulses	In digital circuits, narrow pulses that, due to non-zero rise and fall times of the signal, do not reach a valid high or low level. For example, a runt pulse may occur when switching between asynchronous clocks or as the result of a race condition in which a signal takes two separate paths through a circuit. These race conditions may have different delays and are then recombined to form a glitch or when the output of a flip-flop becomes metastable.

S

sampling	The process of converting an analog signal into a series of digital values or reversed.
schematic	A diagram, drawing, or sketch that details the elements of a system, such as the elements of an electrical circuit or the elements of a logic diagram for a computer.
seed value	An initial value loaded into a linear feedback shift register or random number generator.
serial	1. Pertaining to a process in which all events occur one after the other. 2. Pertaining to the sequential or consecutive occurrence of two or more related activities in a single device or channel.
set	To force a bit/register to a value of logic 1.
settling time	The time it takes for an output signal or value to stabilize after the input has changed from one value to another.
shift	The movement of each bit in a word one position to either the left or right. For example, if the hex value 0x24 is shifted one place to the left, it becomes 0x48. If the hex value 0x24 is shifted one place to the right, it becomes 0x12.
shift register	A memory storage device that sequentially shifts a word either left or right to output a stream of serial data.
sign bit	The most significant binary digit, or bit, of a signed binary number. If set to a logic 1, this bit represents a negative quantity.
signal	A detectable transmitted energy that can be used to carry information. As applied to electronics, any transmitted electrical impulse.
silicon ID	A unique identifier of the EZ-PD™ PMG1-S2 MCU silicon.
skew	The difference in arrival time of bits transmitted at the same time, in parallel transmission.
slave device	A device that allows another device to control the timing for data exchanges between two devices. Or when devices are cascaded in width, the slave device is the one that allows another device to control the timing of data exchanges between the cascaded devices and an external interface. The controlling device is called the master device.
software	A set of computer programs, procedures, and associated documentation about the operation of a data processing system (for example, compilers, library routines, manuals, and circuit diagrams). Software is often written first as source code, and then converted to a binary format that is specific to the device on which the code will be executed.
software reset	A partial reset executed by software to bring part of the system back to a known state. A software reset will restore the M8CP to a know state but not EZ-PD™ PMG1-S2 MCU blocks, systems, peripherals, or registers. For a software reset, the CPU registers (CPU_A, CPU_F, CPU_PC, CPU_SP, and CPU_X) are set to 0x00. Therefore, code execution will begin at flash address 0x0000.

SRAM	An acronym for static random access memory. A memory device allowing users to store and retrieve data at a high rate of speed. The term static is used because, when a value is loaded into an SRAM cell, it will remain unchanged until it is explicitly altered or until power is removed from the device.
SROM	An acronym for supervisory read only memory. The SROM holds code that is used to boot the device, calibrate circuitry, and perform flash operations. The functions of the SROM may be accessed in normal user code, operating from flash.
stack	A stack is a data structure that works on the principle of Last In First Out (LIFO). This means that the last item put on the stack is the first item that can be taken off.
stack pointer	A stack may be represented in a computer's inside blocks of memory cells, with the bottom at a fixed location and a variable stack pointer to the current top cell.
state machine	The actual implementation (in hardware or software) of a function that can be considered to consist of a set of states through which it sequences.
sticky	A bit in a register that maintains its value past the time of the event that caused its transition, has passed.
stop bit	A signal following a character or block that prepares the receiving device to receive the next character or block.
switching	The controlling or routing of signals in circuits to execute logical or arithmetic operations, or to transmit data between specific points in a network.
switch phasing	The clock that controls a given switch, PHI1 or PHI2, in respect to the switch capacitor (SC) blocks. The EZ-PD™ PMG1-S2 MCU SC blocks have two groups of switches. One group of these switches is normally closed during PHI1 and open during PHI2. The other group is open during PHI1 and closed during PHI2. These switches can be controlled in the normal operation, or in reverse mode if the PHI1 and PHI2 clocks are reversed.
synchronous	1. A signal whose data is not acknowledged or acted upon until the next active edge of a clock signal. 2. A system whose operation is synchronized by a clock signal.

T

tap	The connection between two blocks of a device created by connecting several blocks/components in a series, such as a shift register or resistive voltage divider.
terminal count	The state at which a counter is counted down to zero.
threshold	The minimum value of a signal that can be detected by the system or sensor under consideration.

Thumb-2 The Thumb-2 instruction set is a highly efficient and powerful instruction set that delivers significant benefits in terms of ease of use, code size, and performance. The Thumb-2 instruction set is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions.

transistors The transistor is a solid-state semiconductor device used for amplification and switching, and has three terminals: a small current or voltage applied to one terminal controls the current through the other two. It is the key component in all modern electronics. In digital circuits, transistors are used as very fast electrical switches, and arrangements of transistors can function as logic gates, RAM-type memory, and other devices. In analog circuits, transistors are essentially used as amplifiers.

tristate A function whose output can adopt three states: 0, 1, and Z (high impedance). The function does not drive any value in the Z state and, in many respects, may be considered to be disconnected from the rest of the circuit, allowing another output to drive the same *net*.

U

UART A UART or universal asynchronous receiver-transmitter translates between parallel bits of data and serial bits.

user The person using the EZ-PD™ PMG1-S2 MCU device and reading this manual.

user modules Pre-build, pre-tested hardware/firmware peripheral functions that take care of managing and configuring the lower level Analog and Digital EZ-PD™ PMG1-S2 MCU Blocks. User Modules also provide high level *API (Application Programming Interface)* for the peripheral function.

user space The bank 0 space of the register map. The registers in this bank are more likely to be modified during normal program execution and not just during initialization. Registers in bank 1 are most likely to be modified only during the initialization phase of the program.

V

V_{DDD} A name for a power net meaning "voltage drain." The most positive power supply signal. Usually 5 or 3.3 volts.

volatile Not guaranteed to stay the same value or level when not in scope.

V_{SS} A name for a power net meaning "voltage source." The most negative power supply signal.

W

watchdog timer	A timer that must be serviced periodically. If it is not serviced, the CPU will reset after a specified period of time.
waveform	The representation of a signal as a plot of amplitude versus time.

X

XOR	See *Boolean Algebra*.