ESP32-C3

Technical Reference Manual

Pre-release v0.1
Espressif Systems

About This Manual

The **ESP32-C3 Technical Reference Manual** is addressed to application developers. The manual provides detailed and complete information on how to use the ESP32-C3 memory and peripherals.

For pin definition, electrical characteristics, and package information, please see *ESP32-C3 Datasheet*.

Document Updates

Please always refer to the latest version on https://www.espressif.com/en/support/download/documents.

Revision History

For revision history of this document, please refer to the last page.

Documentation Change Notification

Espressif provides email notifications to keep customers updated on changes to technical documentation. Please subscribe at www.espressif.com/en/subscribe.

Certification

Download certificates for Espressif products from www.espressif.com/en/certificates.

Contents

List of Tables

List of Figures

1 Reset and Clock

1.1 Reset

1.1.1 Overview

ESP32-C3 provides four types of reset that occur at different levels, namely CPU Reset, Core Reset, System Reset, and Chip Reset. All reset types mentioned above (except Chip Reset) maintain the data stored in internal memory. Figure 1-1 shows the scope of affected subsystems by each type of reset.

1.1.2 Architectural Overview

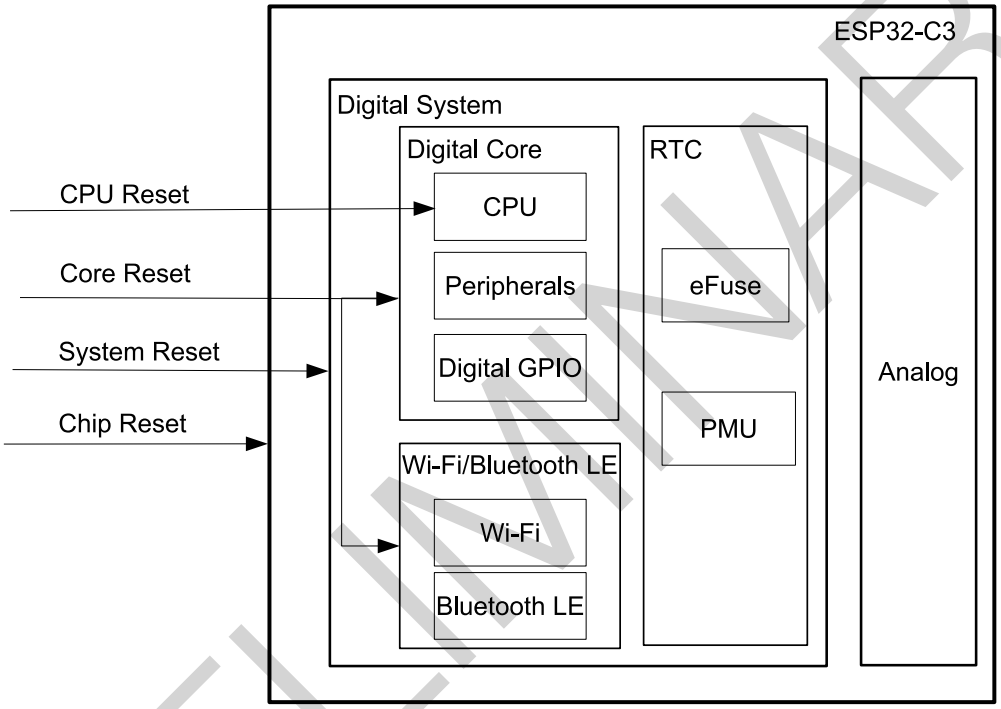


Figure 1-1. Reset Types

1.1.3 Features

- Support four reset levels:
  - CPU Reset: Only resets CPU core. Once such reset is triggered, the instructions from the CPU reset vector will be executed.
  - Core Reset: Resets the whole digital system except RTC, including CPU, peripherals, Wi-Fi, Bluetooth® LE, and digital GPIOs.
  - System Reset: Resets the whole digital system, including RTC.
  - Chip Reset: Resets the whole chip.
- Support software reset and hardware reset:
  - Software Reset: the CPU can trigger a software reset by configuring the corresponding registers.
  - Hardware Reset: Hardware reset is directly triggered by the circuit.

Note:
If CPU is reset, SENSITIVE registers will be reset, too.

1.1.4 Functional Description

CPU will be reset immediately when any of the reset above occurs. Users can get reset source codes by reading register RTC_CNTL_RESET_CAUSE_PROCPU after the reset is released.

Table 1-1 lists possible reset sources and the types of reset they trigger.

Table 1-1. Reset Sources

Code	Source	Reset Type	Comments
0x01	Chip reset	Chip Reset	See the note[1] below
0x0F	Brown-out system reset	Chip Reset or System Reset	Triggered by brown-out detector, see the note[2] below
0x10	RWDT system reset	System Reset	
0x13	CLK GLITCH reset	System Reset	
0x12	Super Watchdog reset	System Reset	
0x03	Software system reset	Core Reset	Triggered by configuring RTC_CNTL_SW_SYS_RST
0x05	Deep-sleep reset	Core Reset	
0x14	eFuse reset	Core Reset	Triggered by eFuse CRC error
0x17	Power glitch reset	Core Reset	Triggered by power glitch
0x07	MWDT0 core reset	Core Reset	
0x08	MWDT1 core reset	Core Reset	
0x09	RWDT core reset	Core Reset	
0x0B	MWDT0 CPU reset	CPU Reset	
0x0C	Software CPU reset	CPU Reset	Triggered by configuring RTC_CNTL_SW_PROCPU_RST
0x0D	RWDT CPU reset	CPU Reset	
0x11	MWDT1 CPU reset	CPU Reset	

Note:

1. Chip Reset can be triggered by the following two sources:
   - Triggered by chip power-on.
   - Triggered by brown-out detector.
2. Once brown-out status is detected, the detector will trigger System Reset or Chip Reset, depending on register configuration.

1.2 Clock

1.2.1 Overview

ESP32-C3 clocks are mainly sourced from oscillator (OSC), RC, and PLL circuit, and then processed by the dividers or selectors, which allows most functional modules to select their working clock according to their power consumption and performance requirements. Figure 1-2 shows the system clock structure.

1.2.2 Architectural Overview

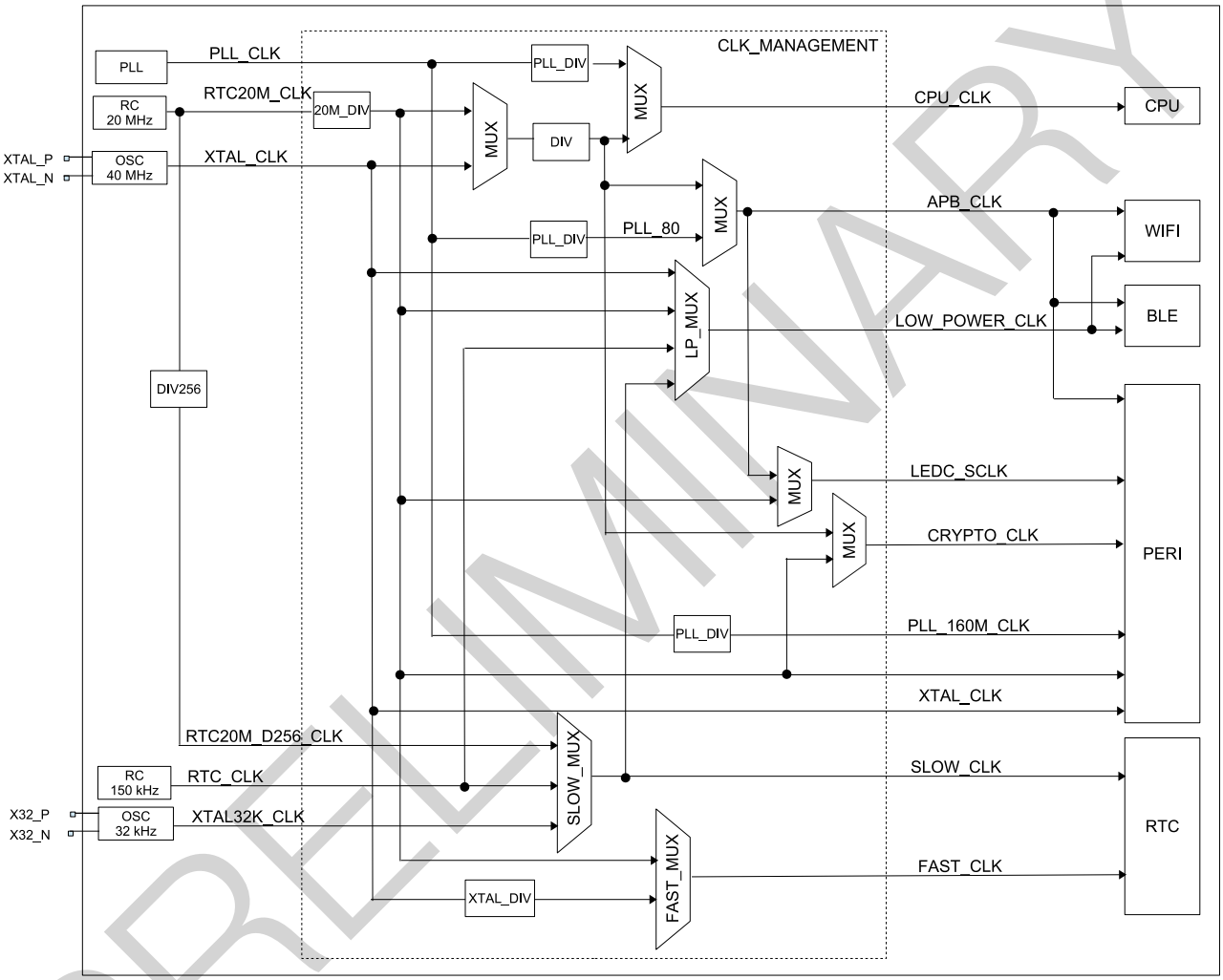


Figure 1-2. System Clock

1.2.3 Features

ESP32-C3 clocks can be classified in two types depending on their frequencies:

- High speed clocks for devices working at a higher frequency, such as CPU and digital peripherals
  - PLL_CLK (320 MHz or 480 MHz): internal PLL clock
  - XTAL_CLK (40 MHz): external crystal clock
- Slow speed clocks for low-power devices, such as RTC module and low-power peripherals

- XTAL32K_CLK (32 kHz): external crystal clock
- RTC20M_CLK (20 MHz by default): internal oscillator with adjustable frequency
- RTC20M_D256_CLK (78.125 kHz by default): internal clock derived from RTC20M_CLK divided by 256
- RTC_CLK (150 kHz by default): internal low power clock with adjustable frequency

1.2.4 Functional Description

1.2.4.1 CPU Clock

As Figure 1-2 shows, CPU_CLK is the master clock for CPU and it can be as high as 160 MHz when CPU works in high performance mode. Alternatively, CPU can run at lower frequencies, such as at 2 MHz, to lower power consumption. Users can set PLL_CLK, RTC20M_CLK or XTAL_CLK as CPU_CLK clock source by configuring register SYSTEM_SOC_CLK_SEL, see Table 1-2 and Table 1-3. By default, the CPU clock is sourced from XTAL_CLK with a divider of 2, i.e. the CPU clock is 20 MHz.

Table 1-2. CPU_CLK Clock Source

SYSTEM_SOC_CLK_SEL Value	CPU Clock Source
0	XTAL_CLK
1	PLL_CLK
2	RTC20M_CLK

Table 1-3. CPU Clock Frequency

CPU Clock Source	SEL_0*	SEL_1*	SEL_2*	CPU Clock Frequency
XTAL_CLK	0	-	-	CPU_CLK = XTAL_CLK/(SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT ranges from 0 ~ 1023. Default is 1
PLL_CLK (480 MHz)	1	1	0	CPU_CLK = PLL_CLK/6 CPU_CLK frequency is 80 MHz
PLL_CLK (480 MHz)	1	1	1	CPU_CLK = PLL_CLK/3 CPU_CLK frequency is 160 MHz
PLL_CLK (320 MHz)	1	0	0	CPU_CLK = PLL_CLK/4 CPU_CLK frequency is 80 MHz
PLL_CLK (320 MHz)	1	0	1	CPU_CLK = PLL_CLK/2 CPU_CLK frequency is 160 MHz
RTC20M_CLK	2	-	-	CPU_CLK = RTC20M_CLK/(SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT ranges from 0 ~ 1023. Default is 1

* The value of register SYSTEM_SOC_CLK_SEL.
* The value of register SYSTEM_PLL_FREQ_SEL.
* The value of register SYSTEM_CPUPERIOD_SEL.

1.2.4.2 Peripheral Clock

Peripheral clocks include APB_CLK, CRYPTO_CLK, PLL_160M_CLK, LEDC_SCLK, XTAL_CLK, and RTC20M_CLK. Table 1-4 shows which clock can be used by each peripheral.

Table 1-4. Peripheral Clocks

Peripheral	XTAL_CLK	APB_CLK	PLL_160M_CLK	(RTC) FAST_CLK	RTC20M_CLK	CRYPTO_CLK	LEDC_SCLK
TIMG	Y	Y					
I2S	Y		Y				
UHCI		Y					
UART	Y	Y			Y		
RMT	Y	Y			Y		
I2C	Y				Y		
SPI	Y	Y					
eFuse Controller				Y			
SARADC		Y					
Temperature Sensor	Y				Y		
USB		Y					
CRYPTO						Y	
TWAI Controller		Y					
LEDC	Y	Y	Y		Y		Y
SYS_TIMER	Y	Y					

APB_CLK

The frequency of APB_CLK is determined by the clock source of CPU_CLK as shown in Table 1-5.

Table 1-5. APB_CLK Clock Frequency

CPU_CLK Source	APB_CLK Frequency
PLL_CLK	80 MHz
XTAL_CLK	CPU_CLK
RTC20M_CLK	CPU_CLK

CRYPTO_CLK

The frequency of CRYPTO_CLK is determined by the CPU_CLK source, as shown in Table 1-6.

Table 1-6. CRYPTO_CLK Frequency

CPU_CLK Source	CRYPTO_CLK Frequency
PLL_CLK	160 MHz
XTAL_CLK	CPU_CLK
RTC20M_CLK	CPU_CLK

PLL_160M_CLK

PLL_160M_CLK is divided from PLL_CLK according to current PLL frequency.

LEDC_SCLK

LEDC module uses RTC20M_CLK as clock source when APB_CLK is disabled. In other words, when the system is in low-power mode, most peripherals will be halted (as APB_CLK is turned off), but LEDC can still work normally via RTC20M_CLK.

1.2.4.3 Wi-Fi and Bluetooth® LE Clock

Wi-Fi and Bluetooth LE can only work when CPU_CLK uses PLL_CLK as its clock source. Suspending PLL_CLK requires that Wi-Fi and Bluetooth LE have entered low-power mode first.

LOW_POWER_CLK uses XTAL32K_CLK, XTAL_CLK, RTC20M_CLK or SLOW_CLK (the low clock selected by RTC) as its clock source for Wi-Fi and Bluetooth LE in low-power mode.

1.2.4.4 RTC Clock

The clock sources for SLOW_CLK and FAST_CLK are low-frequency clocks. RTC module can operate when most other clocks are stopped. SLOW_CLK derived from RTC_CLK, XTAL32K_CLK or RTC20M_D256_CLK is used to clock Power Management module. FAST_CLK is used to clock On-chip Sensor module. It can be sourced from a divided XTAL_CLK or from a divided RTC20M_CLK.

2 Random Number Generator

2.1 Introduction

The ESP32-C3 contains a true random number generator, which generates 32-bit random numbers that can be used for cryptographical operations, among other things.

2.2 Features

The random number generator in ESP32-C3 generates true random numbers, which means random number generated from a physical process, rather than by means of an algorithm. No number generated within the specified range is more or less likely to appear than any other number.

2.3 Functional Description

Every 32-bit value that the system reads from the RNG_DATA_REG register of the random number generator is a true random number. These true random numbers are generated based on the **thermal noise** in the system and the **asynchronous clock mismatch**.

- **Thermal noise** comes from the high-speed ADC or SAR ADC or both. Whenever the high-speed ADC or SAR ADC is enabled, bit streams will be generated and fed into the random number generator through an XOR logic gate as random seeds.
- RTC20M_CLK is an **asynchronous clock** source and it increases the RNG entropy by introducing circuit metastability.

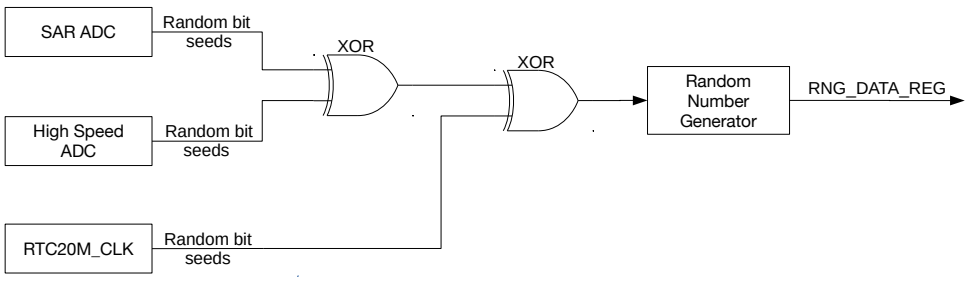


Figure 2-1. Noise Source

When there is noise coming from the SAR ADC, the random number generator is fed with a 2-bit entropy in one clock cycle of RTC20M_CLK (20 MHz), which is generated from an internal RC oscillator (see Chapter 1 *Reset and Clock* for details). Thus, it is advisable to read the RNG_DATA_REG register at a maximum rate of 1 MHz to obtain the maximum entropy.

When there is noise coming from the high-speed ADC, the random number generator is fed with a 2-bit entropy in one APB clock cycle, which is normally 80 MHz. Thus, it is advisable to read the RNG_DATA_REG register at a maximum rate of 5 MHz to obtain the maximum entropy.

A data sample of 2 GB, which is read from the random number generator at a rate of 5 MHz with only the high-speed ADC being enabled, has been tested using the Dieharder Random Number Testsuite (version 3.31.1). The sample passed all tests.

2.4 Programming Procedure

When using the random number generator, make sure at least either the SAR ADC, high-speed ADC[1], or RTC20M_CLK[2] is enabled. Otherwise, pseudo-random numbers will be returned.

- SAR ADC can be enabled by using the DIG ADC controller.
- High-speed ADC is enabled automatically when the Wi-Fi or Bluetooth modules is enabled.
- RTC20M_CLK is enabled by setting the RTC_CNTL_DIG_CLK20M_EN bit in the RTC_CNTL_CLK_CONF_REG register.

Note:

1. Note that, when the Wi-Fi module is enabled, the value read from the high-speed ADC can be saturated in some extreme cases, which lowers the entropy. Thus, it is advisable to also enable the SAR ADC as the noise source for the random number generator for such cases.
2. Enabling RTC20M_CLK increases the RNG entropy. However, to ensure maximum entropy, it's recommended to always enable an ADC source as well.

When using the random number generator, read the RNG_DATA_REG register multiple times until sufficient random numbers have been generated. Ensure the rate at which the register is read does not exceed the frequencies described in section 2.3 above.

2.5 Register Summary

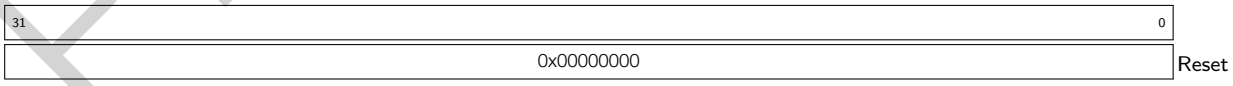
The address in the following table is relative to the random number generator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Name	Description	Address	Access
RNG_DATA_REG	Random number data	0x00B0	RO

2.6 Register

The address in this section is relative to the random number generator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Register 2.1. RNG_DATA_REG (0x00B0)



RNG_DATA Random number source. (RO)

3 System and Memory

3.1 Overview

The ESP32-C3 is an ultra-low-power and highly-integrated system with a 32-bit RISC-V single-core processor with a four-stage pipeline that operates at up to 160 MHz. All internal memory, external memory, and peripherals are located on the CPU buses.

3.2 Features

- **Address Space**
  - 792 KB of internal memory address space accessed from the instruction bus
  - 552 KB of internal memory address space accessed from the data bus
  - 836 KB of peripheral address space
  - 8 MB of external memory virtual address space accessed from the instruction bus
  - 8 MB of external memory virtual address space accessed from the data bus
  - 384 KB of internal DMA address space
- **Internal Memory**
  - 384 KB of Internal ROM
  - 400 KB of Internal SRAM
  - 8 KB of RTC Memory
- **External Memory**
  - Supports up to 16 MB external flash
- **Peripheral Space**
  - 35 modules/peripherals in total
- **GDMA**
  - 7 GDMA-supported modules/peripherals

Figure 3-1 illustrates the system structure and address mapping.

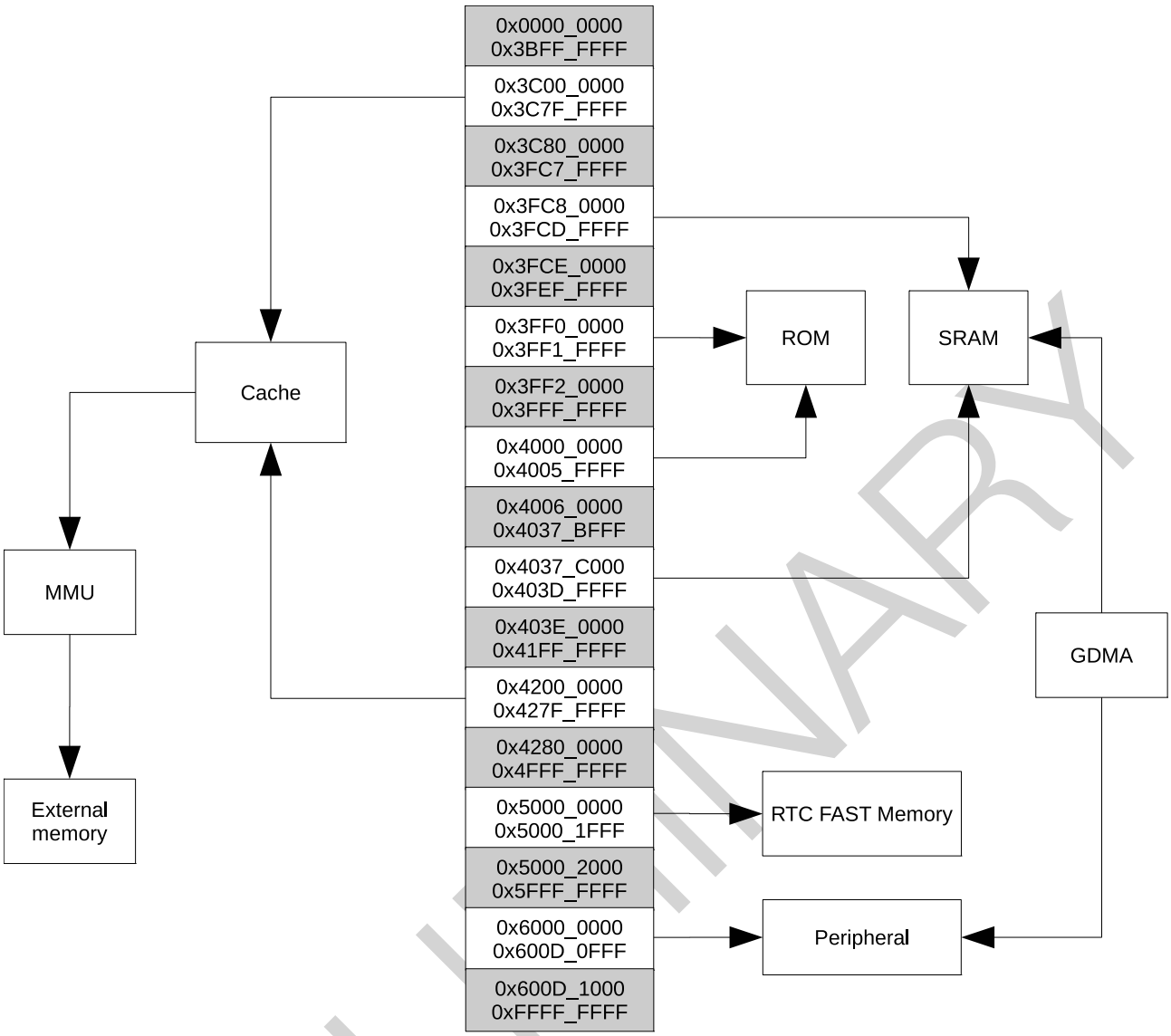


Figure 3-1. System Structure and Address Mapping

Note:

- The address space with gray background is not available to users.
- The range of addresses available in the address space may be larger than the actual available memory of a particular type.

3.3 Functional Description

3.3.1 Address Mapping

Addresses below 0x4000_0000 are accessed using the data bus. Addresses in the range of 0x4000_0000 ~ 0x4FFF_FFFF are accessed using the instruction bus. Addresses over and including 0x5000_0000 are shared by the data bus and the instruction bus.

Both data bus and instruction bus are little-endian. The CPU can access data via the data bus using single-byte, double-byte, 4-byte alignment. The CPU can also access data via the instruction bus, but only in 4-byte aligned

manner.

The CPU can:

- directly access the internal memory via both data bus and instruction bus;
- access the external memory which is mapped into the virtual address space via cache;
- directly access modules/peripherals via data bus.

Table 3-1 lists the address ranges on the data bus and instruction bus and their corresponding target memory.

Some internal and external memory can be accessed via both data bus and instruction bus. In such cases, the CPU can access the same memory using multiple addresses.

Table 3-1. Address Mapping

Bus Type	Boundary Address		Size	Target
	Low Address	High Address		
	0x0000_0000	0x3BFF_FFFF		Reserved
Data bus	0x3C00_0000	0x3C7F_FFFF	8 MB	External memory
	0x3C80_0000	0x3FC7_FFFF		Reserved
Data bus	0x3FC8_0000	0x3FCD_FFFF	384 KB	Internal memory
	0x3FCE_0000	0x3FEF_FFFF		Reserved
Data bus	0x3FF0_0000	0x3FF1_FFFF	128 KB	Internal memory
	0x3FF2_0000	0x3FFF_FFFF		Reserved
Instruction bus	0x4000_0000	0x4005_FFFF	384 KB	Internal memory
	0x4006_0000	0x4037_BFFF		Reserved
Instruction bus	0x4037_C000	0x403D_FFFF	400 KB	Internal memory
	0x403E_0000	0x41FF_FFFF		Reserved
Instruction bus	0x4200_0000	0x427F_FFFF	8 MB	External memory
	0x4280_0000	0x4FFF_FFFF		Reserved
Data/Instruction bus	0x5000_0000	0x5000_1FFF	8 KB	Internal memory
	0x5000_2000	0x5FFF_FFFF		Reserved
Data/Instruction bus	0x6000_0000	0x600D_0FFF	836 KB	Peripherals
	0x600D_1000	0xFFFF_FFFF		Reserved

3.3.2 Internal Memory

The ESP32-C3 consists of the following three types of internal memory:

- Internal ROM (384 KB): The Internal ROM of the ESP32-C3 is a Mask ROM, meaning it is strictly read-only and cannot be reprogrammed. Internal ROM contains the ROM code (software instructions and some software read-only data) of some low level system software.
- Internal SRAM (400 KB): The Internal Static RAM (SRAM) is a volatile memory that can be quickly accessed by the CPU (generally within a single CPU clock cycle).
  - A part of the SRAM can be configured to operate as a cache for external memory access.
  - Some parts of the SRAM can only be accessed via the CPU's instruction bus.

  - Some parts of the SRAM can be accessed via both the CPU's instruction bus and the CPU's data bus.

- RTC Memory (8 KB): The RTC (Real Time Clock) memory implemented as Static RAM (SRAM) thus is volatile. However, RTC memory has the added feature of being persistent in deep sleep (i.e., the RTC memory retains its values throughout deep sleep).
  - RTC FAST Memory (8 KB): RTC FAST memory can only be accessed by the CPU and can be generally used to store instructions and data that needs to persist across a deep sleep.

Based on the three different types of internal memory described above, the internal memory of the ESP32-C3 is split into three segments: Internal ROM (384 KB), Internal SRAM (400 KB), RTC FAST Memory (8 KB).

However, within each segment, there may be different bus access restrictions (e.g., some parts of the segment may only be accessible by the CPU's Data bus). Therefore, each some segments are also further divided into parts. Table 3-2 describes each part of internal memory and their address ranges on the data bus and/or instruction bus.

Table 3-2. Internal Memory Address Mapping

Bus Type	Boundary Address		Size	Target
	Low Address	High Address		
Data bus	0x3FF0_0000	0x3FF1_FFFF	128 KB	Internal ROM 1
	0x3FC8_0000	0x3FCD_FFFF	384 KB	Internal SRAM 1
Instruction bus	0x4000_0000	0x4003_FFFF	256 KB	Internal ROM 0
	0x4004_0000	0x4005_FFFF	128 KB	Internal ROM 1
	0x4037_C000	0x4037_FFFF	16 KB	Internal SRAM 0
	0x4038_0000	0x403D_FFFF	384 KB	Internal SRAM 1
Data/Instruction bus	0x5000_0000	0x5000_1FFF	8 KB	RTC FAST Memory

Note:

All of the internal memories are managed by Permission Control module. An internal memory can only be accessed when it is allowed by Permission Control, then the internal memory can be available to the CPU.

1. Internal ROM 0

Internal ROM 0 is a 256 KB, read-only memory space, addressed by the CPU only through the instruction bus via 0x4000_0000 ~ 0x4003_FFFF, as shown in Table 3-2.

2. Internal ROM 1

Internal ROM 1 is a 128 KB, read-only memory space, addressed by the CPU through the instruction bus via 0x4004_0000 ~ 0x4005_FFFF or through the data bus via 0x3FF0_0000 ~ 0x3FF1_FFFF in the same order, as shown in Table 3-2.

This means, for example, address 04004_0000 and 0x3FF0_0000 correspond to the same word, 0x4004_0004 and 0x3FF0_0004 correspond to the same word, 0x4004_0008 and 0x3FF0_0008 correspond to the same word, etc (the same ordering applies for Internal SRAM 1).

3. Internal SRAM 0

Internal SRAM 0 is a 16 KB, read-and-write memory space, addressed by the CPU through the instruction bus via the range described in Table 3-2.

This memory managed by Permission Control, can be configured as instruction cache to store cache instructions or read-only data of the external memory. In this case, the memory cannot be accessed by the CPU.

4. Internal SRAM 1

Internal SRAM 1 is a 384 KB, read-and-write memory space, addressed by the CPU through the data bus or instruction bus, in the same order, via the ranges described in Table 3-2.

5. RTC FAST Memory

RTC FAST Memory is a 8 KB, read-and-write SRAM, addressed by the CPU through the data/instruction bus via the shared address 0x5000_0000 ~ 0x5000_1FFF, as described in Table 3-2.

3.3.3 External Memory

ESP32-C3 supports SPI, Dual SPI, Quad SPI, and QPI interfaces that allow connection to multiple external flash. It supports hardware manual encryption and automatic decryption based on XTS_AES to protect user programs and data in the external flash.

3.3.3.1 External Memory Address Mapping

The CPU accesses the external memory via the cache. According to the MMU (Memory Management Unit) settings, the cache maps the CPU's address to the external memory's physical address. Due to this address mapping, the ESP32-C3 can address up to 16 MB external flash.

Using the cache, ESP32-C3 is able to support the following address space mappings. Note that the instruction bus address space (8MB) and the data bus address space (8 MB) is always shared.

- Up to 8 MB instruction bus address space can be mapped into the external flash. The mapped address space is organized as individual 64-KB blocks.
- Up to 8 MB data bus (read-only) address space can be mapped into the external flash. The mapped address space is organized as individual 64-KB blocks.

Table 3-3 lists the mapping between the cache and the corresponding address ranges on the data bus and instruction bus.

Table 3-3. External Memory Address Mapping

Bus Type	Boundary Address		Size	Target
	Low Address	High Address		
Data bus (read-only)	0x3C00_0000	0x3C7F_FFFF	8 MB	Uniform Cache
Instruction bus	0x4200_0000	0x427F_FFFF	8 MB	Uniform Cache

Note:

Only if the CPU obtains permission for accessing the external memory, can it be responded for memory access.

3.3.3.2 Cache

As shown in Figure 3-2, ESP32-C3 has a read-only uniform cache which is eight-way set-associative, its size is 16 KB and its block size is 32 bytes. When cache is active, some internal memory space will be occupied by

cache (see Internal SRAM 0 in Section 3.3.2).

The uniform cache is accessible by the instruction bus and the data bus at the same time, but can only respond to one of them at a time. When a cache miss occurs, the cache controller will initiate a request to the external memory.

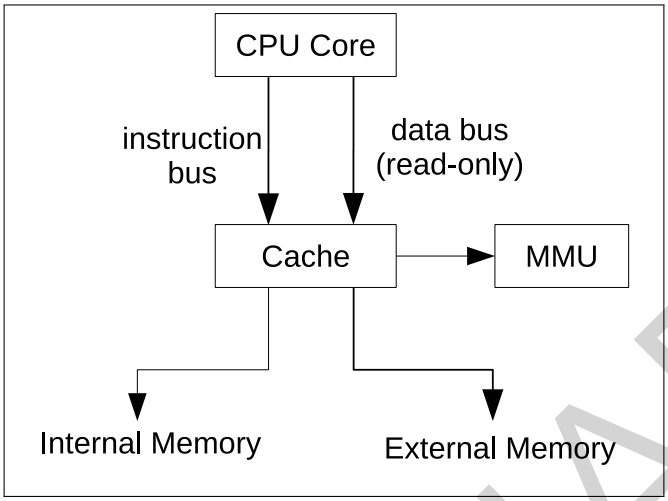


Figure 3-2. Cache Structure

3.3.3.3 Cache Operations

ESP32-C3 cache support the following operations:

1. **Invalidate**: This operation is used to clear valid data in the cache. After this operation is completed, the data will only be stored in the external memory. The CPU needs to access the external memory in order to read this data. There are two types of invalidate-operation: automatic invalidation (Auto-Invalidate) and manual invalidation (Manual-Invalidate). Manual-Invalidate is performed only on data in the specified area in the cache, while Auto-Invalidate is performed on all data in the cache.
2. **Preload**: This operation is used to load instructions and data into the cache in advance. The minimum unit of preload-operation is one block. There are two types of preload-operation: manual preload (Manual-Preload) and automatic preload (Auto-Preload). Manual-Preload means that the hardware prefetches a piece of continuous data according to the virtual address specified by the software. Auto-Preload means the hardware prefetches a piece of continuous data according to the current address where the cache hits or misses (depending on configuration).
3. **Lock/Unlock**: The lock operation is used to prevent the data in the cache from being easily replaced. There are two types of lock: prelock and manual lock. When prelock is enabled, the cache locks the data in the specified area when filling the missing data to cache memory, while the data outside the specified area will not be locked. When manual lock is enabled, the cache checks the data that is already in the cache memory and only locks the data in the specified area, and leaves the data outside the specified area unlocked. When there are missing data, the cache will replace the data in the unlocked way first, so the data in the locked way is always stored in the cache and will not be replaced. But when all ways within the cache are locked, the cache will replace data, as if it was not locked. Unlocking is the reverse of locking, except that it only can be done manually.

   Please note that the Manual-Invalidate operations will only work on the unlocked data. If you expect to perform such operation on the locked data, please unlock them first.

3.3.4 GDMA Address Space

The GDMA (General Direct Memory Access) peripheral in ESP32-C3 can provide DMA (Direct Memory Access) services including:

- Data transfers between different locations of internal memory;
- Data transfers between modules/peripherals and internal memory.

GDMA uses the same addresses as the data bus to read and write Internal SRAM 1. Specifically, GDMA uses address range 0x3FC8_0000 ~ 0x3FCD_FFFF to access Internal SRAM 1. Note that GDMA cannot access the internal memory occupied by the cache.

There are 7 peripherals/modules that can work together with GDMA. As shown in Figure 3-3, these 7 vertical lines in turn correspond to these 7 peripherals/modules with GDMA function, the horizontal line represents a certain channel of GDMA (can be any channel), and the intersection of the vertical line and the horizontal line indicates that a peripheral/module has the ability to access the corresponding channel of GDMA. If there are multiple intersections on the same line, it means that these peripherals/modules cannot enable the GDMA function at the same time.

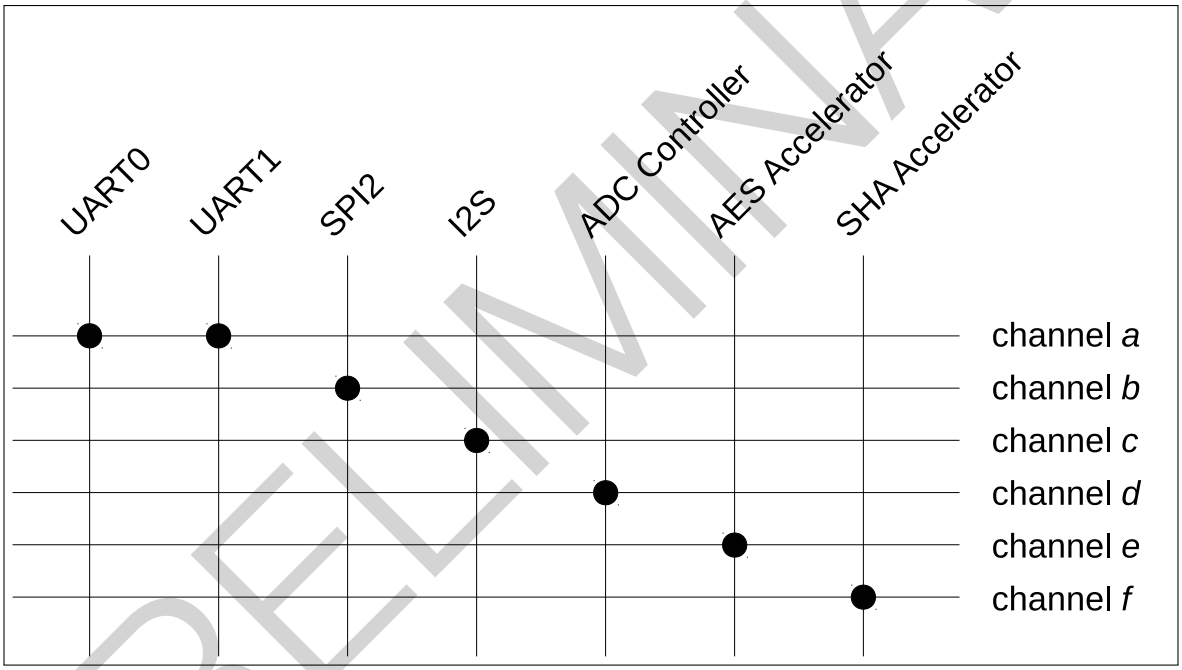


Figure 3-3. Peripherals/modules that can work with GDMA

These peripherals/modules can access any memory available to GDMA.

Note:

When accessing a memory via GDMA, a corresponding access permission is needed, otherwise this access may fail.

3.3.5 Modules/Peripherals

The CPU can access modules/peripherals via 0x6000_0000 ~ 0x600D_0FFF shared by the data/instruction bus.

3.3.5.1 Module/Peripheral Address Mapping

Table 3-4 lists all the modules/peripherals and their respective address ranges. Note that the address space of specific modules/peripherals is defined by "Boundary Address" (including both Low Address and High Address).

Table 3-4. Module/Peripheral Address Mapping

Target	Boundary Address		Size	Notes
	Low Address	High Address		
UART Controller 0	0x6000_0000	0x6000_0FFF	4 KB	
Reserved	0x6000_1000	0x6000_1FFF		
SPI Controller 1	0x6000_2000	0x6000_2FFF	4 KB	
SPI Controller 0	0x6000_3000	0x6000_3FFF	4 KB	
GPIO	0x6000_4000	0x6000_4FFF	4 KB	
Reserved	0x6000_5000	0x6000_6FFF		
TIMER	0x6000_7000	0x6000_7FFF	4 KB	
Low-Power Management	0x6000_8000	0x6000_8FFF	4 KB	
IO MUX	0x6000_9000	0x6000_9FFF	4 KB	
Reserved	0x6000_A000	0x6000_FFFF		
UART Controller 1	0x6001_0000	0x6001_0FFF	4 KB	
Reserved	0x6001_1000	0x6001_2FFF		
I2C Controller	0x6001_3000	0x6001_3FFF	4 KB	
UHCI0	0x6001_4000	0x6001_4FFF	4 KB	
Reserved	0x6001_5000	0x6001_5FFF		
Remote Control Peripheral	0x6001_6000	0x6001_6FFF	4 KB	
Reserved	0x6001_7000	0x6001_8FFF		
LED Control PWM	0x6001_9000	0x6001_9FFF	4 KB	
eFuse Controller	0x6001_A000	0x6001_AFFF	4 KB	
Reserved	0x6001_B000	0x6001_EFFF		
Timer Group 0	0x6001_F000	0x6001_FFFF	4 KB	
Timer Group 1	0x6002_0000	0x6002_0FFF	4 KB	
Reserved	0x6002_1000	0x6002_2FFF		
System Timer	0x6002_3000	0x6002_3FFF	4 KB	
SPI Controller 2	0x6002_4000	0x6002_4FFF	4 KB	
Reserved	0x6002_5000	0x6002_5FFF		
APB Controller	0x6002_6000	0x6002_6FFF	4 KB	
Reserved	0x6002_7000	0x6002_AFFF		
Two-wire Automotive Interface	0x6002_B000	0x6002_BFFF	4 KB	
Reserved	0x6002_C000	0x6002_CFFF		
I2S Controller	0x6002_D000	0x6002_DFFF	4 KB	
Reserved	0x6002_E000	0x6003_9FFF		
AES Accelerator	0x6003_A000	0x6003_AFFF	4 KB	
SHA Accelerator	0x6003_B000	0x6003_BFFF	4 KB	
RSA Accelerator	0x6003_C000	0x6003_CFFF	4 KB	
Digital Signature	0x6003_D000	0x6003_DFFF	4 KB	

Target	Boundary Address		Size	Notes
	Low Address	High Address		
HMAC Accelerator	0x6003_E000	0x6003_EFFF	4 KB	
GDMA Controller	0x6003_F000	0x6003_FFFF	4 KB	
ADC Controller	0x6004_0000	0x6004_0FFF	4 KB	
Reserved	0x6004_1000	0x6002_FFFF		
USB Serial/JTAG Controller	0x6004_3000	0x6004_3FFF	4 KB	
Reserved	0x6004_4000	0x600B_FFFF		
System Registers	0x600C_0000	0x600C_0FFF	4 KB	
Sensitive Register	0x600C_1000	0x600C_1FFF	4 KB	
Interrupt Matrix	0x600C_2000	0x600C_2FFF	4 KB	
Reserved	0x600C_3000	0x600C_3FFF		
Configure Cache	0x600C_4000	0x600C_BFFF	32 KB	
External Memory Encryption and Decryption	0x600C_C000	0x600C_CFFF	4 KB	
Reserved	0x600C_D000	0x600C_DFFF		
Assist Debug	0x600C_E000	0x600C_EFFF	4 KB	
Reserved	0x600C_F000	0x600C_FFFF		
World Controller	0x600D_0000	0x600D_0FFF	4 KB	

4 IO MUX and GPIO Matrix (GPIO, IO_MUX)

4.1 Overview

The ESP32-C3 chip features 22 physical GPIO pins. Each pin can be used as a general-purpose I/O, or be connected to an internal peripheral signal. Through GPIO matrix and IO MUX, peripheral input signals can be from any IO pins, and peripheral output signals can be routed to any IO pins. Together these modules provide highly configurable I/O.

Note that the GPIO pins are numbered from 0 ~ 21.

4.2 Features

GPIO Matrix Features

- A full-switching matrix between the peripheral input/output signals and the pins. Control signals: DRV, IE, OE, WPU, WPD.
- 49 peripheral input signals can be sourced from the input of any GPIO pins. Control signals: SIG_IN_SEL, IE, etc.
- The output of any GPIO pins can be from any of the 125 peripheral output signals. Control signals: SIG_OUT_SEL, OE, etc.
- Support signal synchronization for peripheral inputs based on APB clock bus.
- Provide input signal filter.
- Support sigma delta modulated output.
- Support GPIO simple input and output.

IO MUX Features

- Provide one configuration register IO_MUX_GPIO*n*_REG for each GPIO pin. The pin can be configured to
  - perform GPIO function routed by GPIO matrix;
  - or perform direct connection bypassing GPIO matrix.
- Support some high-speed digital signals (SPI, JTAG, UART) bypassing GPIO matrix for better high-frequency digital performance. In this case, IO MUX is used to connect these pins directly to peripherals.

4.3 Architectural Overview

This section provides an overview to the architecture of IO MUX and GPIO matrix with the following figures:

- Figure 4-1 shows the general work flow of IO MUX and GPIO matix.
- Figure 4-2 shows in details how IO MUX and GPIO matrix route signals from pins to peripherals, and from peripherals to pins.
- Figure 4-3 shows the interface logic for a GPIO pin.

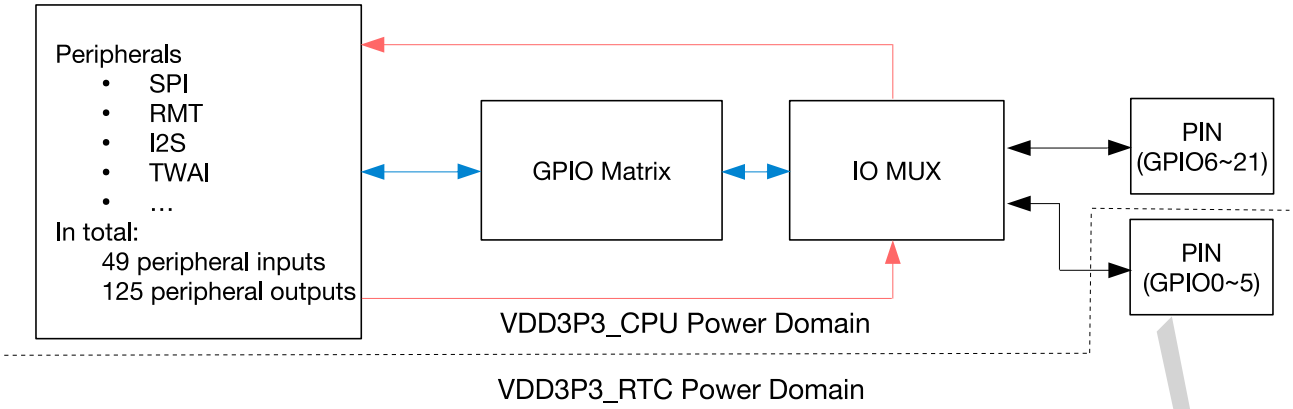


Figure 4-1. Diagram of IO MUX and GPIO Matrix

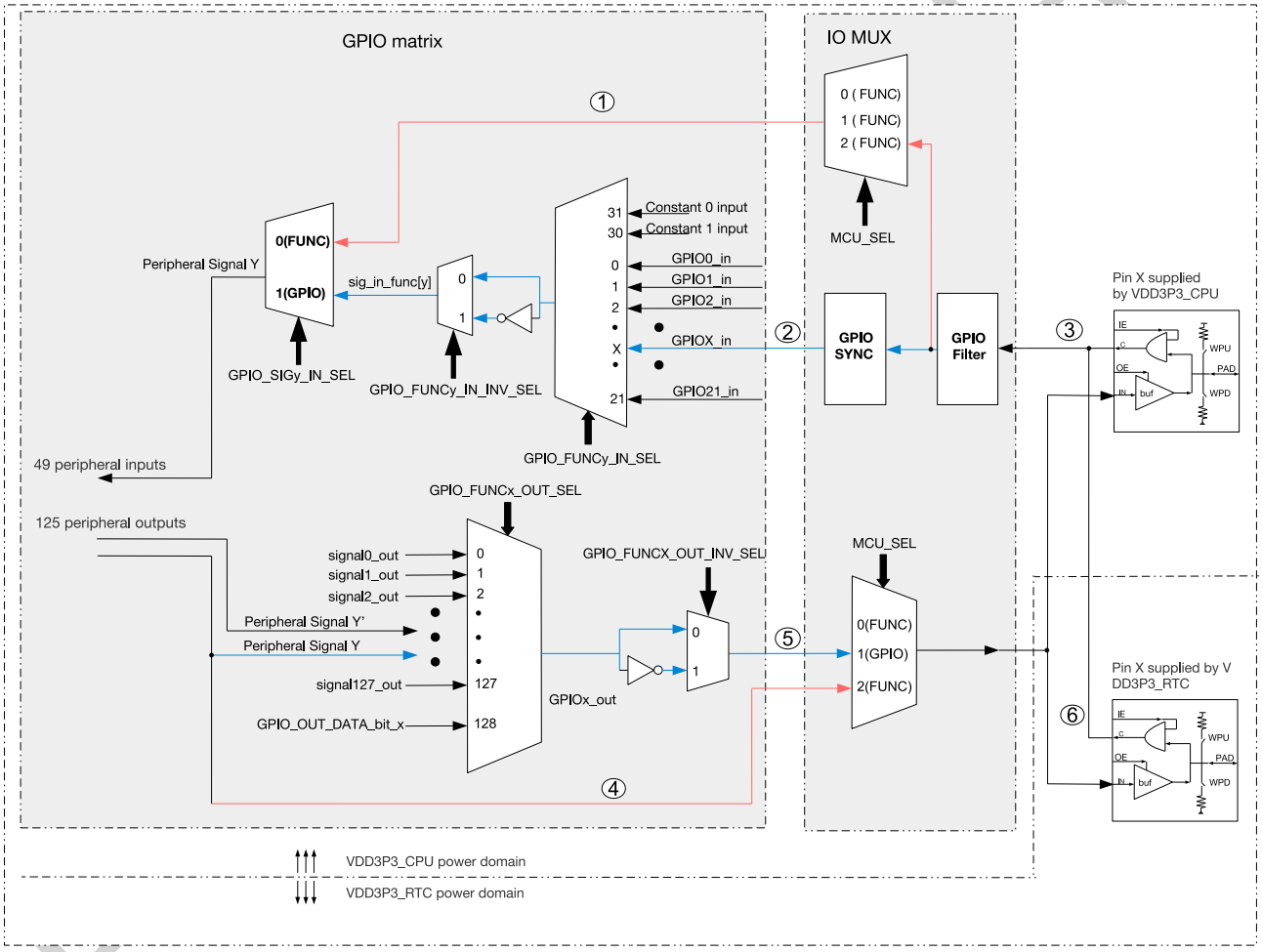


Figure 4-2. Architecture of IO MUX and GPIO Matirx

1. Only part of peripheral input signals (Y: 0 ~ 3, 6 ~ 7, 9 ~ 10, 63 ~ 68) can bypass GPIO matrix. The other input signals can only be routed to peripherals via GPIO matrix.
2. There are only 22 inputs from GPIO SYNC to GPIO matrix, since ESP32-C3 provides 22 GPIO pins in total.
3. The pins supplied by VDD3P3_CPU are controlled by the signals: IE, OE, WPU, and WPD.
4. Only part of peripheral outputs (0 ~ 59, 63~ 127) can be routed to pins bypassing GPIO matrix. See Table 4-1.

5. There are only 22 outputs (GPIO pin X: 0 ~ 21) from GPIO matrix to IO MUX.
6. The pins supplied by VDD3P3_RTC are controlled by the signals: IE, OE, WPU, and WPD.

Figure 4-3 shows the internal structure of a pad, which is an electrical interface between the chip logic and the GPIO pin.

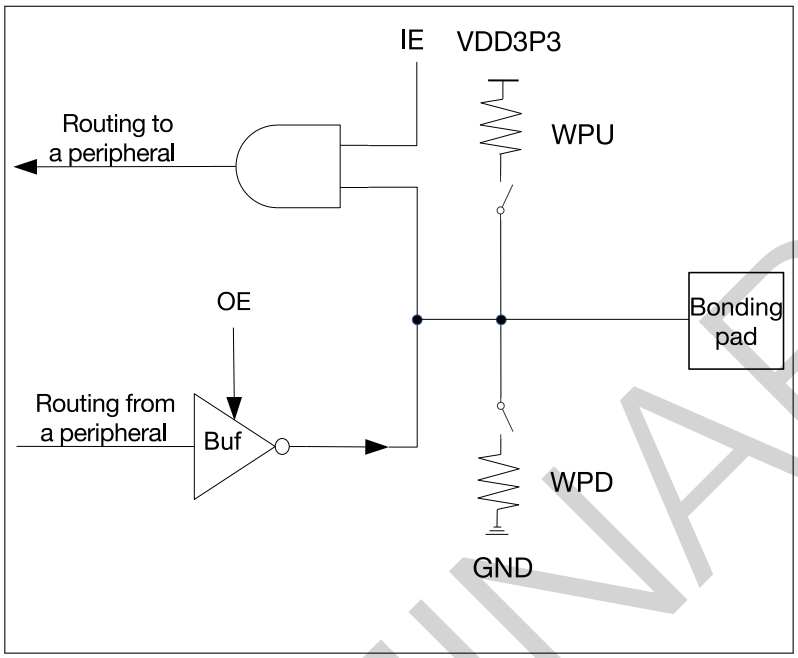


Figure 4-3. Internal Structure of a Pad

- IE: input enable
- OE: output enable
- WPU: internal weak pull-up
- WPD: internal weak pull-down

4.4 Peripheral Input via GPIO Matrix

4.4.1 Overview

To receive a peripheral input signal via GPIO matrix, the matrix is configured to source the peripheral input signal from one of the 22 GPIOs (0 ~ 21), see Table 4-1. Meanwhile, register corresponding to the peripheral signal should be set to receive input signal via GPIO matrix.

4.4.2 Signal Synchronization

When signals are directed from pins using GPIO matrix, the signals will be synchronized to the APB bus clock by GPIO SYNC hardware, then go to GPIO matrix. This synchronization applies to all GPIO matrix signals but does not apply when using the IO MUX, see Figure 4-2.

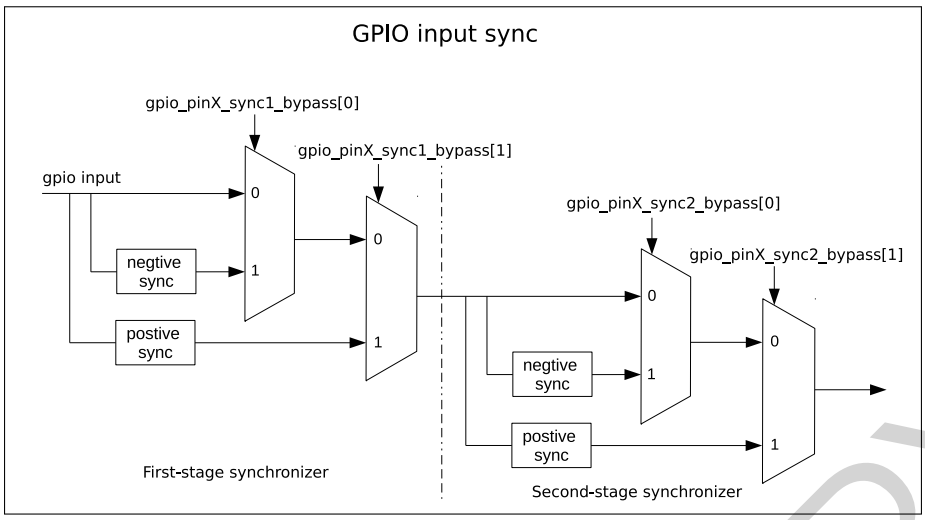


Figure 4-4. GPIO Input Synchronized on APB Clock Rising Edge or on Falling Edge

Figure 4-4 shows the functionality of GPIO SYNC. In the figure, negative sync and positive sync mean GPIO input is synchronized on APB clock falling edge and on APB clock rising edge, respectively.

4.4.3 Functional Description

To read GPIO pin *X*[1] into peripheral signal *Y*, follow the steps below:

1. Configure register GPIO_FUNC*y*_IN_SEL_CFG_REG corresponding to peripheral signal *Y* in GPIO matrix:
   - Set GPIO_SIG*y*_IN_SEL to enable peripheral signal input via GPIO matrix.
   - Set GPIO_FUNC*y*_IN_SEL to the desired GPIO pin, i.e. *X* here.

   Note that some peripheral signals have no valid GPIO_SIG*y*_IN_SEL bit, namely, these peripherals can only receive input signals via GPIO matrix.
2. Optionally enable the filter for pin input signals by setting the register IO_MUX_GPIO*n*_FILTER_EN. Only the signals with a valid width of more than two clock cycles can be sampled, see Figure 4-5.

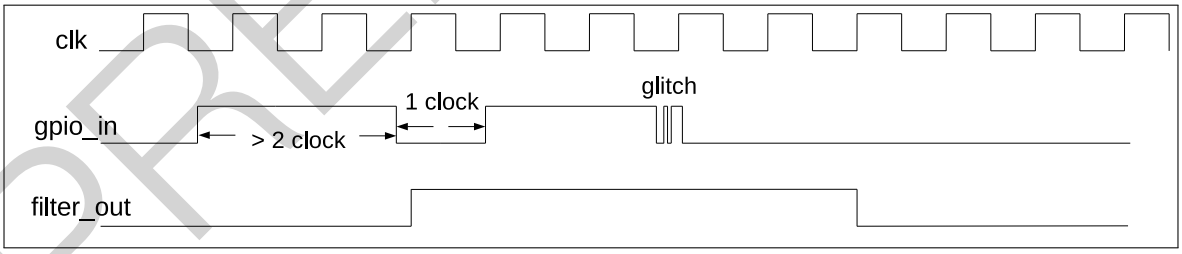


Figure 4-5. Filter Timing of GPIO Input Signals

3. Synchronize GPIO input. To do so, please set GPIO_PIN*x*_REG corresponding to GPIO pin *X* as follows:
   - Set GPIO_PIN*x*_SYNC1_BYPASS to enable input signal synchronized on rising edge or on falling edge in the first clock, see Figure 4-4.
   - Set GPIO_PIN*x*_SYNC2_BYPASS to enable input signal synchronized on rising edge or on falling edge in the second clock, see Figure 4-4.

4. Configure IO MUX register to enable pin input. For this end, please set IO_MUX_GPIO*x*_REG corresponding to GPIO pin*x* as follows:
   - Set IO_MUX_GPIO*x*_FUN_IE to enable input[2].
   - Set or clear IO_MUX_GPIO*x*_FUN_WPU and IO_MUX_GPIO*x*_FUN_WPD, as desired, to enable or disable pull-up and pull-down resistors.

For example, to connect I2S MCLK input signal[3] (I2S_MCLK_in, signal index 12) to GPIO7, please follow the steps below. Note that GPIO7 is also named as MTDO pin.

1. Set GPIO_SIG12_IN_SEL in register GPIO_FUNC12_IN_SEL_CFG_REG to enable peripheral signal input via GPIO matrix.
2. Set GPIO_FUNC12_IN_SEL in register GPIO_FUNC12_IN_SEL_CFG_REG to 7.
3. Set IO_MUX_GPIO7_FUN_IE in register IO_MUX_GPIO7_REG to enable pin input.

Note:

1. One input pin can be connected to multiple peripheral input signals.
2. The input signal can be inverted by configuring GPIO_FUNC*y*_IN_INV_SEL.
3. It is possible to have a peripheral read a constantly low or constantly high input value without connecting this input to a pin. This can be done by selecting a special GPIO_FUNC*y*_IN_SEL input, instead of a GPIO number:
   - When GPIO_FUNC*y*_IN_SEL is set to 0x1F, input signal is always 0.
   - When GPIO_FUNC*y*_IN_SEL is set to 0x1E, input signal is always 1.

4.4.4 Simple GPIO Input

GPIO_IN_REG holds the input values of each GPIO pin. The input value of any GPIO pin can be read at any time without configuring GPIO matrix for a particular peripheral signal. However, it is necessary to enable the input via IO MUX by setting IO_MUX_GPIO*x*_FUN_IE bit in register IO_MUX_GPIO*x*_REG corresponding to pin *X*, as mentioned in Section 4.4.2.

4.5 Peripheral Output via GPIO Matrix

4.5.1 Overview

To output a signal from a peripheral via GPIO matrix, the matrix is configured to route peripheral output signals (0 ~ 59, 63~ 127) to one of the 22 GPIOs (0 ~ 21). See Table 4-1.

The output signal is routed from the peripheral into GPIO matrix and then into IO MUX. IO MUX must be configured to set the chosen pin to GPIO function. This enables the output GPIO signal to be connected to the pin.

Note:

There is a range of peripheral output signals (97 ~ 100) which are not connected to any peripheral, but to the input signals (97 ~ 100) directly. These can be used to input a signal from one GPIO pin and output directly to another GPIO pin.

4.5.2 Functional Description

Some of the 125 output signals (0 ~ 59, 63~ 127) can be set to go through GPIO matrix into IO MUX and then to a pin. Figure 4-2 illustrates the configuration.

To output peripheral signal *Y* to a particular GPIO pin *X*[1], follow these steps:

1. Configure register GPIO_FUNC*x*_OUT_SEL_CFG_REG and GPIO_ENABLE_REG[*x*] corresponding to GPIO pin *X* in GPIO matrix. Recommended operation: use corresponding W1TS (write 1 to set) and W1TC (write 1 to clear) registers to set or clear GPIO_ENABLE_REG.
   - Set the GPIO_FUNC*x*_OUT_SEL field in register GPIO_FUNC*x*_OUT_SEL_CFG_REG to the index of the desired peripheral output signal *Y*.
   - If the signal should always be enabled as an output, set the GPIO_FUNC*x*_OEN_SEL bit in register GPIO_FUNC*x*_OUT_SEL_CFG_REG and the bit in register GPIO_ENABLE_W1TS_REG, corresponding to GPIO pin *X*. To have the output enable signal decided by internal logic (for example, the SPIQ_oe in column "Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0" in Table 4-1), clear GPIO_FUNC*x*
     _OEN_SEL bit instead.
   - Clear the corresponding bit in register GPIO_ENABLE_W1TC_REG to disable the output from the GPIO pin.
2. For an open drain output, set the GPIO_PIN*x*_PAD_DRIVER bit in register GPIO_PIN*x*_REG corresponding to GPIO pin *X*.
3. Configure IO MUX register to enable output via GPIO matrix. Set the IO_MUX_GPIO*x*_REG corresponding to GPIO pin *X* as follows:
   - Set the field IO_MUX_GPIO*x*_MCU_SEL to desired IO MUX function corresponding to GPIO pin*X*. This is Function 1 (GPIO function), numeric value 1, for all pins.
   - Set the IO_MUX_GPIO*x*_FUN_DRV field to the desired value for output strength (0 ~ 3). The higher the driver strength, the more current can be sourced/sunk from the pin.
     - 0: ~5 mA
     - 1: ~10 mA
     - 2: ~20 mA (default value)
     - 3: ~40 mA
   - If using open drain mode, set/clear the IO_MUX_GPIO*x*_FUN_WPU and IO_MUX_GPIO*x*_FUN_WPD bits to enable/disable the internal pull-up/pull-down resistors.

Note:

1. The output signal from a single peripheral can be sent to multiple pins simultaneously.
2. The output signal can be inverted by setting GPIO_FUNC*x*_OUT_INV_SEL bit.

4.5.3 Simple GPIO Output

GPIO matrix can also be used for simple GPIO output. This can be done as below:

- Set GPIO matrix GPIO_FUNC*n*_OUT_SEL with a special peripheral index 128 (0x80);
- Set the corresponding bit in GPIO_OUT_REG register to the desired GPIO output value.

Note:

- GPIO_OUT_REG[0] ~ GPIO_OUT_REG[21] correspond to GPIO0 ~ GPIO21, and GPIO_OUT_REG[25:22] are invalid.
- Recommended operation: use corresponding W1TS and W1TC registers, such as GPIO_OUT_W1TS/GPIO_OUT_W1TC to set or clear the registers GPIO_OUT_REG.

4.5.4 Sigma Delta Modulated Output (SDM)

4.5.4.1 Functional Description

Four out of the 125 peripheral outputs (output index: 55 ~ 58) support 1-bit second-order sigma delta modulation. By default output is enabled for these four channels. This modulator can also output PDM (pulse density modulation) signal with configurable duty cycle. The transfer function of this second-order SDM modulator is:

$$H(z) = \text{X(z)}z^{-1} + \text{E(z)(1-}z^{-1})^2$$

E(z) is quantization error and X(z) is the input.
Sigma Delta modulator supports scaling down of APB_CLK by divider 1 ~ 256:

- Set GPIOSD_FUNCTION_CLK_EN to enable the modulator clock.
- Configure register GPIOSD_SD*n*_PRESCALE (*n* is 0 ~ 3 for four channels).

After scaling, the clock cycle is equal to one pulse output cycle from the modulator.

GPIOSD_SD*n*_IN is a signed number with a range of [-128, 127] and is used to control the duty cycle [1] of PDM output signal.

- GPIOSD_SD*n*_IN = -128, the duty cycle of the output signal is 0%.
- GPIOSD_SD*n*_IN = 0, the duty cycle of the output signal is near 50%.
- GPIOSD_SD*n*_IN = 127, the duty cycle of the output signal is close to 100%.

The formula for calculating PDM signal duty cycle is shown as below:

$$Duty\_Cycle = \frac{GPIOSD\_SDn\_IN + 128}{256}$$

Note:
For PDM signals, duty cycle refers to the percentage of high level cycles to the whole statistical period (several pulse cycles, for example 256 pulse cycles).

4.5.4.2 SDM Configuration

The configuration of SDM is shown below:

- Route one of SDM outputs to a pin via GPIO matrix, see Section 4.5.2.
- Enable the modulator clock by setting the register GPIOSD_FUNCTION_CLK_EN.
- Configure the divider value by setting the register GPIOSD_SD*n*_PRESCALE.
- Configure the duty cycle of SDM output signal by setting the register GPIOSD_SD*n*_IN.

4.6 Direct Input and Output via IO MUX

4.6.1 Overview

Some high-speed signals (SPI and JTAG) can bypass GPIO matrix for better high-frequency digital performance. In this case, IO MUX is used to connect these pins directly to peripherals.

This option is less flexible than routing signals via GPIO matrix, as the IO MUX register for each GPIO pin can only select from a limited number of functions, but high-frequency digital performance can be improved.

4.6.2 Functional Description

Two registers must be configured in order to bypass GPIO matrix for peripheral input signals:

1. IO_MUX_GPIO*n*_MCU_SEL for the GPIO pin must be set to the required pin function. For the list of pin functions, please refer to Section 4.11.
2. Clear GPIO_SIG*n*_IN_SEL to route the input directly to the peripheral.

To bypass GPIO matrix for peripheral output signals, IO_MUX_GPIO*n*_MCU_SEL for the GPIO pin must be set to the required pin function. For the list of pin functions, please refer to Section 4.11.

Note:
Not all signals can be directly connected to peripheral via IO MUX. Some input/output signals can only be connected to peripheral via GPIO matrix.

4.7 Analog Functions of GPIO Pins

Some GPIO pins in ESP32-C3 provide analog functions. When the pin is used for analog purpose, make sure that pull-up and pull-down resistors are disabled by following configuration:

- Set IO_MUX_GPIO*n*_MCU_SEL to 1, and clear IO_MUX_GPIO*n*_FUN_IE, IO_MUX_GPIO*n*_FUN_WPU, IO_MUX_GPIO*n*_FUN_WPD.
- Write 1 to GPIO_ENABLE_W1TC*[n]*, to clear output enable.

See Table 4-4 for analog functions of ESP32-C3 pins.

4.8 Pin Hold Feature

Each GPIO pin (including the RTC pins: GPIO0 ~ GPIO5) has an individual hold function controlled by a RTC register. When the pin is set to hold, the state is latched at that moment and will not change no matter how the internal signals change or how the IO MUX/GPIO configuration is modified. Users can use the hold function for the pins to retain the pin state through a core reset and system reset triggered by watchdog time-out or Deep-sleep events.

Note:

- For digital pins (GPIO6 ~21), to maintain pin input/output status in Deep-sleep mode, users can set RTC_CNTL_DIG_PAD_HOLD*n* in register RTC_CNTL_DIG_PAD_HOLD_REG to 1 before powering down. To disable the hold function after the chip is woken up, users can set RTC_CNTL_DIG_PAD_HOLD*n* to 0.
- For RTC pins (GPIO0 ~5), the input and output values are controlled by the corresponding bits of register RTC_CNTL_RTC_PAD_HOLD_REG, and users can set it to 1 to hold the value or set it to 0 to unhold the value.

4.9 Power Supplies and Management of GPIO Pins

4.9.1 Power Supplies of GPIO Pins

For more information on the power supply for IO pins, please refer to Pin Definition in *ESP32-C3 Datasheet*. All the pins can be used to wake up the chip from Light-sleep mode, but only the pins (GPIO0 ~ GPIO5) in VDD3P3_RTC domain can be used to wake up the chip from Deep-sleep mode.

4.9.2 Power Supply Management

Each ESP32-C3 pin is connected to one of the two different power domains.

- VDD3P3_RTC: the input power supply for both RTC and CPU
- VDD3P3_CPU: the input power supply for CPU

4.10 Peripheral Signal List

Table 4-1 shows the peripheral input/output signals via GPIO matrix.

Please pay attention to the configuration of the bit GPIO_FUNC*n*_OEN_SEL:

- GPIO_FUNC*n*_OEN_SEL = 1: the output enable is controlled by the corresponding bit *n* of GPIO_ENABLE_REG:
  - GPIO_ENABLE_REG = 0: output is disabled;
  - GPIO_ENABLE_REG = 1: output is enabled;
- GPIO_FUNC*n*_OEN_SEL = 0: use the output enable signal from peripheral, for example SPIQ_oe in the column "Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0" of Table 4-1. Note that the signals such as SPIQ_oe can be 1 (1'd1) or 0 (1'd0), depending on the configuration of corresponding peripherals. If it's 1'd1 in the "Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0", it indicates that once the register GPIO_FUNC*n*_OEN_SEL is cleared, the output signal is always enabled by default.

Note:

Signals are numbered consecutively, but not all signals are valid.

- For input signals, only 0 ~ 3, 6 ~ 19, 28 ~ 35, 40 ~ 42, 45, 51 ~ 54, 63 ~ 68, 74, 77 ~ 80, 97 ~ 100 are valid.
- For output signals, only 0 ~ 59, 63 ~ 127 are valid.

Table 4-1. Peripheral Signals via GPIO Matrix

Signal No.	Input Signal	Default value	Direct Input through IO MUX	Output Signal	Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0	Direct Output through IO MUX
0	SPIQ_in	0	yes	SPIQ_out	SPIQ_oe	yes
1	SPID_in	0	yes	SPID_out	SPID_oe	yes
2	SPIHD_in	0	yes	SPIHD_out	SPIHD_oe	yes
3	SPIWP_in	0	yes	SPIWP_out	SPIWP_oe	yes
4	-	-	-	SPICLK_out_mux	SPICLK_oe	yes
5	-	-	-	SPICS0_out	SPICS0_oe	yes
6	U0RXD_in	0	yes	U0TXD_out	1'd1	yes
7	U0CTS_in	0	yes	U0RTS_out	1'd1	no
8	U0DSR_in	0	no	U0DTR_out	1'd1	no
9	U1RXD_in	0	yes	U1TXD_out	1'd1	no
10	U1CTS_in	0	yes	U1RTS_out	1'd1	no
11	U1DSR_in	0	no	U1DTR_out	1'd1	no
12	I2S_MCLK_in	0	no	I2S_MCLK_out	1'd1	no
13	I2SO_BCK_in	0	no	I2SO_BCK_out	1'd1	no
13	I2SO_WS_in	0	no	I2SO_WS_out	1'd1	no
15	I2SI_SD_in	0	no	I2SO_SD_out	1'd1	no
16	I2SI_BCK_in	0	no	I2SI_BCK_out	1'd1	no
17	I2SI_WS_in	0	no	I2SI_WS_out	1'd1	no
18	gpio_bt_priority	0	no	gpio_wlan_prio	1'd1	no
19	gpio_bt_active	0	no	gpio_wlan_active	1'd1	no
20	-	-	-	cpu_test_bu0	1'd1	no
21	-	-	-	cpu_test_bu1	1'd1	no
22	-	-	-	cpu_test_bu2	1'd1	no
23	-	-	-	cpu_test_bu3	1'd1	no

Signal No.	Input Signal	Default value	Direct Input through IO_MUX	Output Signal	Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0	Direct Output through IO_MUX
24	-	-	-	cpu_test_bu4	1'd1	no
25	-	-	-	cpu_test_bu5	1'd1	no
26	-	-	-	cpu_test_bu6	1'd1	no
27	-	-	-	cpu_test_bu7	1'd1	no
28	cpu_gpio_in0	0	no	cpu_gpio_out0	cpu_gpio_out_oen0	no
29	cpu_gpio_in1	0	no	cpu_gpio_out1	cpu_gpio_out_oen1	no
30	cpu_gpio_in2	0	no	cpu_gpio_out2	cpu_gpio_out_oen2	no
31	cpu_gpio_in3	0	no	cpu_gpio_out3	cpu_gpio_out_oen3	no
32	cpu_gpio_in4	0	no	cpu_gpio_out4	cpu_gpio_out_oen4	no
33	cpu_gpio_in5	0	no	cpu_gpio_out5	cpu_gpio_out_oen5	no
34	cpu_gpio_in6	0	no	cpu_gpio_out6	cpu_gpio_out_oen6	no
35	cpu_gpio_in7	0	no	cpu_gpio_out7	cpu_gpio_out_oen7	no
36	-	-	-	usb_jtag_tck	1'd1	no
37	-	-	-	usb_jtag_tms	1'd1	no
38	-	-	-	usb_jtag_tdi	1'd1	no
39	-	-	-	usb_jtag_tdo	1'd1	no
40	usb_extphy_vp	0	no	usb_extphy_oen	1'd1	no
41	usb_extphy_vm	0	no	usb_extphy_speed	1'd1	no
42	usb_extphy_rcv	0	no	usb_extphy_vpo	1'd1	no
43	-	-	-	usb_extphy_vmo	1'd1	no
44	-	-	-	usb_extphy_suspnd	1'd1	no
45	ext_adc_start	0	no	ledc_ls_sig_out0	1'd1	no
46	-	-	-	ledc_ls_sig_out1	1'd1	no
47	-	-	-	ledc_ls_sig_out2	1'd1	no
48	-	-	-	ledc_ls_sig_out3	1'd1	no
49	-	-	-	ledc_ls_sig_out4	1'd1	no

Signal No.	Input Signal	Default value	Direct Input through IO_MUX	Output Signal	Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0	Direct Output through IO_MUX
50	-	-	-	ledc_ls_sig_out5	1'd1	no
51	rmt_sig_in0	0	no	rmt_sig_out0	1'd1	no
52	rmt_sig_in1	0	no	rmt_sig_out1	1'd1	no
53	I2CEXT0_SCL_in	1	no	I2CEXT0_SCL_out	I2CEXT0_SCL_oe	no
54	I2CEXT0_SDA_in	1	no	I2CEXT0_SDA_out	I2CEXT0_SDA_oe	no
55	-	-	-	gpio_sd0_out	1'd1	no
56	-	-	-	gpio_sd1_out	1'd1	no
57	-	-	-	gpio_sd2_out	1'd1	no
58	-	-	-	gpio_sd3_out	1'd1	no
59	-	-	-	I2SO_SD1_out	1'd1	no
60	-	-	-	-	1'd1	-
61	-	-	-	-	1'd1	-
62	-	-	-	-	1'd1	-
63	FSPICLK_in	0	yes	FSPICLK_out_mux	FSPICLK_oe	yes
64	FSPIQ_in	0	yes	FSPIQ_out	FSPIQ_oe	yes
65	FSPID_in	0	yes	FSPID_out	FSPID_oe	yes
66	FSPIHD_in	0	yes	FSPIHD_out	FSPIHD_oe	yes
67	FSPIWP_in	0	yes	FSPIWP_out	FSPIWP_oe	yes
68	FSPICS0_in	0	yes	FSPICS0_out	FSPICS0_oe	yes
69	-	-	-	FSPICS1_out	FSPICS1_oe	no
70	-	-	-	FSPICS2_out	FSPICS2_oe	no
71	-	-	-	FSPICS3_out	FSPICS3_oe	no
72	-	-	-	FSPICS4_out	FSPICS4_oe	no
73	-	-	-	FSPICS5_out	FSPICS5_oe	no
74	twai_rx	1	no	twai_tx	1'd1	no
75	-	-	-	twai_bus_off_on	1'd1	no

Signal No.	Input Signal	Default value	Direct Input through IO_MUX	Output Signal	Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0	Direct Output through IO_MUX
76	-	-	-	twai_clkout	1'd1	no
77	pcmfsync_in	0	no	bt_audio0_irq	1'd1	no
78	pcmclk_in	0	no	bt_audio1_irq	1'd1	no
79	pcmdin	0	no	bt_audio2_irq	1'd1	no
80	rw_wakeup_req	0	no	ble_audio0_irq	1'd1	no
81	-	-	-	ble_audio1_irq	1'd1	no
82	-	-	-	ble_audio2_irq	1'd1	no
83	-	-	-	pcmfsync_out	pcmfsync_en	no
84	-	-	-	pcmclk_out	pcmclk_en	no
85	-	-	-	pcmdout	pcmdout_en	no
86	-	-	-	ble_audio_sync0_p	1'd1	no
87	-	-	-	ble_audio_sync1_p	1'd1	no
88	-	-	-	ble_audio_sync2_p	1'd1	no
89	-	-	-	ant_sel0	1'd1	no
90	-	-	-	ant_sel1	1'd1	no
91	-	-	-	ant_sel2	1'd1	no
92	-	-	-	ant_sel3	1'd1	no
93	-	-	-	ant_sel4	1'd1	no
94	-	-	-	ant_sel5	1'd1	no
95	-	-	-	ant_sel6	1'd1	no
96	-	-	-	ant_sel7	1'd1	no
97	sig_in_func_97	0	no	sig_in_func97	1'd1	no
98	sig_in_func_98	0	no	sig_in_func98	1'd1	no
99	sig_in_func_99	0	no	sig_in_func99	1'd1	no
100	sig_in_func_100	0	no	sig_in_func100	1'd1	no
101	-	-	-	syncerr	!efuse_dis_btlc_gpio1	no

Signal No.	Input Signal	Default value	Direct Input through IO_MUX	Output Signal	Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0	Direct Output through IO_MUX
102	-	-	-	syncfound_flag	!efuse_dis_btlc_gpio1	no
103	-	-	-	evt_cntl_immediate_abort	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
104	-	-	-	linklbl	!efuse_dis_btlc_gpio1&!efuse_dis_btlc_gpio0	no
105	-	-	-	data_en	!efuse_dis_btlc_gpio1&!efuse_dis_btlc_gpio0	no
106	-	-	-	data	!efuse_dis_btlc_gpio1&!efuse_dis_btlc_gpio0	no
107	-	-	-	pkt_tx_on	!efuse_dis_btlc_gpio1	no
108	-	-	-	pkt_rx_on	!efuse_dis_btlc_gpio1	no
109	-	-	-	rw_tx_on	!efuse_dis_btlc_gpio1	no
110	-	-	-	rw_rx_on	!efuse_dis_btlc_gpio1	no
111	-	-	-	evt_req_p	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
112	-	-	-	evt_stop_p	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
113	-	-	-	bt_mode_on	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
114	-	-	-	gpio_lc_diag0	!efuse_dis_btlc_gpio1	no
115	-	-	-	gpio_lc_diag1	!efuse_dis_btlc_gpio1	no
116	-	-	-	gpio_lc_diag2	!efuse_dis_btlc_gpio1	no
117	-	-	-	ch_idx	!efuse_dis_btlc_gpio1&!efuse_dis_btlc_gpio0	no
118	-	-	-	rx_window	!efuse_dis_btlc_gpio1	no
119	-	-	-	update_rx	!efuse_dis_btlc_gpio1	no
120	-	-	-	rx_status	!efuse_dis_btlc_gpio1	no
121	-	-	-	clk_gpio	!efuse_dis_btlc_gpio1	no
122	-	-	-	nbt_ble	!(efuse_dis_btlc_gpio1&efuse_dis_btlc_gpio0)	no
123	-	-	-	CLK_OUT_out1	1'd1	no
124	-	-	-	CLK_OUT_out2	1'd1	no
125	-	-	-	CLK_OUT_out3	1'd1	no
126	-	-	-	SPICS1_out	1'd1	no
127	-	-	-	usb_jtag_trst	1'd1	no

4.11 IO MUX Functions List

Table 4-2 shows the IO MUX functions of each pin.

Table 4-2. IO MUX Pin Functions

GPIO	Pin Name	Function 0	Function 1	Function 2	Function 3	Function 4	Reset	Notes
0	XTAL_32K_P	GPIO0	GPIO0	-	-	-	0	R
1	XTAL_32K_N	GPIO1	GPIO1	-	-	-	0	R
2	GPIO2	GPIO2	GPIO2	FSPIQ	-	-	1	R
3	GPIO3	GPIO3	GPIO3	-	-	-	1	R
4	MTMS	MTMS	GPIO4	FSPIHD	-	-	1	R
5	MTDI	MTDI	GPIO5	FSPIWP	-	-	1	R
6	MTCK	MTCK	GPIO6	FSPICLK	-	-	1*	G
7	MTDO	MTDO	GPIO7	FSPID	-	-	1	G
8	GPIO8	GPIO8	GPIO8	-	-	-	1	-
9	GPIO9	GPIO9	GPIO9	-	-	-	3	-
10	GPIO10	GPIO10	GPIO10	FSPICS0	-	-	1	G
11	VDD_SPI	GPIO11	GPIO11	-	-	-	0	-
12	SPIHD	SPIHD	GPIO12	-	-	-	3	-
13	SPIWP	SPIWP	GPIO13	-	-	-	3	-
14	SPICS0	SPICS0	GPIO14	-	-	-	3	-
15	SPICLK	SPICLK	GPIO15	-	-	-	3	-
16	SPID	SPID	GPIO16	-	-	-	3	-
17	SPIQ	SPIQ	GPIO17	-	-	-	3	-
18	GPIO18	GPIO18	GPIO18	-	-	-	0	USB, G
19	GPIO19	GPIO19	GPIO19	-	-	-	0*	USB
20	U0RXD	U0RXD	GPIO20	-	-	-	1	G
21	U0TXD	U0TXD	GPIO21	-	-	-	1	-

Reset Configurations

"Reset" column shows the default configuration of each pin after reset:

- **0** - IE = 0 (input disabled)
- **1** - IE = 1 (input enabled)
- **2** - IE = 1, WPD = 1 (input enabled, pull-down resistor enabled)
- **3** - IE = 1, WPU = 1 (input enabled, pull-up resistor enabled)
- **0*** - IE = 0, WPU = 0. The USB pull-up value of GPIO19 is 1 by default, therefore, the pin's pull-up resistor is enabled. For more information, see the note below.
- **1*** - If eFuse bit EFUSE_DIS_PAD_JTAG = 1, the pin MTCK is left floating after reset, i.e. IE = 1. If eFuse bit EFUSE_DIS_PAD_JTAG = 0, the pin MTCK is connected to internal pull-up resistor, i.e. IE = 1, WPU = 1.

Note:

- **R** - Pins in VDD3P3_RTC domain, and part of them have analog functions, see Table 4-4.
- **USB** - GPIO18 and GPIO19 are USB pins. The pull-up value of the two pins are controlled by the pins' pull-up value together with USB pull-up value. If any one of the pull-up value is 1, the pin's pull-up resistor will be enabled. The pull-up resistors of USB pins are controlled by USB_SERIAL_JTAG_DP_PULLUP.
- **G** - These pins have glitches during power-up. See details in Table 4-3.

Table 4-3. Power-Up Glitches on Pins

Pin	Glitch	Typical Time Period (ns)
MTCK	Low-level glitch	5
MTDO	Low-level glitch	5
GPIO10	Low-level glitch	5
U0RXD	Low-level glitch	5
GPIO18	Pull-up glitch	50000

4.12 Analog Functions List

Table 4-4 shows the IO MUX pins with analog functions.

Table 4-4. Analog Functions of IO MUX Pins

GPIO Num	Pin Name	Analog Function 0	Analog Function 1
0	XTAL_32K_P	XTAL_32K_P	ADC0
1	XTAL_32K_N	XTAL_32K_N	ADC1
2	GPIO2	-	ADC2
3	GPIO3	-	ADC3

Note:

1. The pin VDD_SPI can be configured as either power supply or normal GPIO.
2. The pins GPIO18 and GPIO19 can be configured as USB pins.

4.13 Register Summary

The addresses in this section are relative to GPIO Matrix, IO MUX and SDM base addresses provided in Table 3-4 in Chapter 3 *System and Memory*.

4.13.1 GPIO Matrix Register Summary

Name	Description	Address	Access
Configuration Registers			
GPIO_BT_SELECT_REG	GPIO bit select register	0x0000	R/W
GPIO_OUT_REG	GPIO output register	0x0004	R/W/SS
GPIO_OUT_W1TS_REG	GPIO output set register	0x0008	WT
GPIO_OUT_W1TC_REG	GPIO output clear register	0x000C	WT

Name	Description	Address	Access
GPIO_ENABLE_REG	GPIO output enable register	0x0020	R/W/SS
GPIO_ENABLE_W1TS_REG	GPIO output enable set register	0x0024	WT
GPIO_ENABLE_W1TC_REG	GPIO output enable clear register	0x0028	WT
GPIO_STRAP_REG	pin strapping register	0x0038	RO
GPIO_IN_REG	GPIO input register	0x003C	RO
GPIO_STATUS_REG	GPIO interrupt status register	0x0044	R/W/SS
GPIO_STATUS_W1TS_REG	GPIO interrupt status set register	0x0048	WT
GPIO_STATUS_W1TC_REG	GPIO interrupt status clear register	0x004C	WT
GPIO_PCPU_INT_REG	GPIO PRO_CPU interrupt status register	0x005C	RO
GPIO_PCPU_NMI_INT_REG	GPIO PRO_CPU (non-maskable) interrupt status register	0x0060	RO
GPIO_STATUS_NEXT_REG	GPIO interrupt source register	0x014C	RO
Pin Configuration Registers			
GPIO_PIN0_REG	GPIO pin0 configuration register	0x0074	R/W
GPIO_PIN1_REG	GPIO pin1 configuration register	0x0078	R/W
GPIO_PIN2_REG	GPIO pin2 configuration register	0x007C	R/W
GPIO_PIN3_REG	GPIO pin3 configuration register	0x0080	R/W
GPIO_PIN4_REG	GPIO pin4 configuration register	0x0084	R/W
GPIO_PIN5_REG	GPIO pin5 configuration register	0x0088	R/W
GPIO_PIN6_REG	GPIO pin6 configuration register	0x008C	R/W
GPIO_PIN7_REG	GPIO pin7 configuration register	0x0090	R/W
GPIO_PIN8_REG	GPIO pin8 configuration register	0x0094	R/W
GPIO_PIN9_REG	GPIO pin9 configuration register	0x0098	R/W
GPIO_PIN10_REG	GPIO pin10 configuration register	0x009C	R/W
GPIO_PIN11_REG	GPIO pin11 configuration register	0x00A0	R/W
GPIO_PIN12_REG	GPIO pin12 configuration register	0x00A4	R/W
GPIO_PIN13_REG	GPIO pin13 configuration register	0x00A8	R/W
GPIO_PIN14_REG	GPIO pin14 configuration register	0x00AC	R/W
GPIO_PIN15_REG	GPIO pin15 configuration register	0x00B0	R/W
GPIO_PIN16_REG	GPIO pin16 configuration register	0x00B4	R/W
GPIO_PIN17_REG	GPIO pin17 configuration register	0x00B8	R/W
GPIO_PIN18_REG	GPIO pin18 configuration register	0x00BC	R/W
GPIO_PIN19_REG	GPIO pin19 configuration register	0x00C0	R/W
GPIO_PIN20_REG	GPIO pin20 configuration register	0x00C4	R/W
GPIO_PIN21_REG	GPIO pin21 configuration register	0x00C8	R/W
Input Function Configuration Registers			
GPIO_FUNC0_IN_SEL_CFG_REG	Configuration register for input signal 0	0x0154	R/W
GPIO_FUNC1_IN_SEL_CFG_REG	Configuration register for input signal 1	0x0158	R/W
...	...	...	...
GPIO_FUNC126_IN_SEL_CFG_REG	Configuration register for input signal 126	0x034C	R/W
GPIO_FUNC127_IN_SEL_CFG_REG	Configuration register for input signal 127	0x0350	R/W
Output Function Configuration Registers			
GPIO_FUNC0_OUT_SEL_CFG_REG	Configuration register for GPIO0 output	0x0554	R/W

Name	Description	Address	Access
GPIO_FUNC1_OUT_SEL_CFG_REG	Configuration register for GPIO1 output	0x0558	R/W
GPIO_FUNC2_OUT_SEL_CFG_REG	Configuration register for GPIO2 output	0x055C	R/W
GPIO_FUNC3_OUT_SEL_CFG_REG	Configuration register for GPIO3 output	0x0560	R/W
GPIO_FUNC4_OUT_SEL_CFG_REG	Configuration register for GPIO4 output	0x0564	R/W
GPIO_FUNC5_OUT_SEL_CFG_REG	Configuration register for GPIO5 output	0x0568	R/W
GPIO_FUNC6_OUT_SEL_CFG_REG	Configuration register for GPIO6 output	0x056C	R/W
GPIO_FUNC7_OUT_SEL_CFG_REG	Configuration register for GPIO7 output	0x0570	R/W
GPIO_FUNC8_OUT_SEL_CFG_REG	Configuration register for GPIO8 output	0x0574	R/W
GPIO_FUNC9_OUT_SEL_CFG_REG	Configuration register for GPIO9 output	0x0578	R/W
GPIO_FUNC10_OUT_SEL_CFG_REG	Configuration register for GPIO10 output	0x057C	R/W
GPIO_FUNC11_OUT_SEL_CFG_REG	Configuration register for GPIO11 output	0x0580	R/W
GPIO_FUNC12_OUT_SEL_CFG_REG	Configuration register for GPIO12 output	0x0584	R/W
GPIO_FUNC13_OUT_SEL_CFG_REG	Configuration register for GPIO13 output	0x0588	R/W
GPIO_FUNC14_OUT_SEL_CFG_REG	Configuration register for GPIO14 output	0x058C	R/W
GPIO_FUNC15_OUT_SEL_CFG_REG	Configuration register for GPIO15 output	0x0590	R/W
GPIO_FUNC16_OUT_SEL_CFG_REG	Configuration register for GPIO16 output	0x0594	R/W
GPIO_FUNC17_OUT_SEL_CFG_REG	Configuration register for GPIO17 output	0x0598	R/W
GPIO_FUNC18_OUT_SEL_CFG_REG	Configuration register for GPIO18 output	0x059C	R/W
GPIO_FUNC19_OUT_SEL_CFG_REG	Configuration register for GPIO19 output	0x05A0	R/W
GPIO_FUNC20_OUT_SEL_CFG_REG	Configuration register for GPIO20 output	0x05A4	R/W
GPIO_FUNC21_OUT_SEL_CFG_REG	Configuration register for GPIO21 output	0x05A8	R/W
Version Register			
GPIO_DATE_REG	GPIO version register	0x06FC	R/W
Clock Gate Register			
GPIO_CLOCK_GATE_REG	GPIO clock gate register	0x062C	R/W

4.13.2 IO MUX Register Summary

Name	Description	Address	Access
Configuration Registers			
IO_MUX_PIN_CTRL_REG	Clock output configuration Register	0x0000	R/W
IO_MUX_GPIO0_REG	IO MUX configuration register for pin XTAL_32K_P	0x0004	R/W
IO_MUX_GPIO1_REG	IO MUX configuration register for pin XTAL_32K_N	0x0008	R/W
IO_MUX_GPIO2_REG	IO MUX configuration register for pin GPIO2	0x000C	R/W
IO_MUX_GPIO3_REG	IO MUX configuration register for pin GPIO3	0x0010	R/W
IO_MUX_GPIO4_REG	IO MUX configuration register for pin MTMS	0x0014	R/W
IO_MUX_GPIO5_REG	IO MUX configuration register for pin MTDI	0x0018	R/W
IO_MUX_GPIO6_REG	IO MUX configuration register for pin MTCK	0x001C	R/W
IO_MUX_GPIO7_REG	IO MUX configuration register for pin MTDO	0x0020	R/W
IO_MUX_GPIO8_REG	IO MUX configuration register for pin GPIO8	0x0024	R/W
IO_MUX_GPIO9_REG	IO MUX configuration register for pin GPIO9	0x0028	R/W

Name	Description	Address	Access
IO_MUX_GPIO10_REG	IO MUX configuration register for pin GPIO10	0x002C	R/W
IO_MUX_GPIO11_REG	IO MUX configuration register for pin VDD_SPI	0x0030	R/W
IO_MUX_GPIO12_REG	IO MUX configuration register for pin SPIHD	0x0034	R/W
IO_MUX_GPIO13_REG	IO MUX configuration register for pin SPIWP	0x0038	R/W
IO_MUX_GPIO14_REG	IO MUX configuration register for pin SPICS0	0x003C	R/W
IO_MUX_GPIO15_REG	IO MUX configuration register for pin SPICLK	0x0040	R/W
IO_MUX_GPIO16_REG	IO MUX configuration register for pin SPID	0x0044	R/W
IO_MUX_GPIO17_REG	IO MUX configuration register for pin SPIQ	0x0048	R/W
IO_MUX_GPIO18_REG	IO MUX configuration register for pin GPIO18	0x004C	R/W
IO_MUX_GPIO19_REG	IO MUX configuration register for pin GPIO19	0x0050	R/W
IO_MUX_GPIO20_REG	IO MUX configuration register for pin U0RXD	0x0054	R/W
IO_MUX_GPIO21_REG	IO MUX configuration register for pin U0TXD	0x0058	R/W
Version Register			
IO_MUX_DATE_REG	IO MUX Version Control Register	0x00FC	R/W

4.13.3 SDM Register Summary

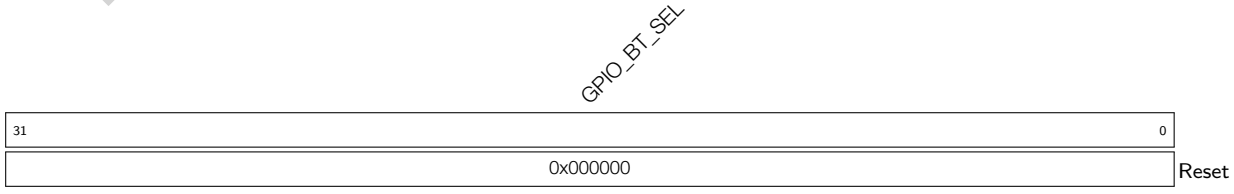
Name	Description	Address	Access
Configuration registers			
GPIOSD_SIGMADELTA0_REG	Duty Cycle Configuration Register of SDM0	0x0000	R/W
GPIOSD_SIGMADELTA1_REG	Duty Cycle Configuration Register of SDM1	0x0004	R/W
GPIOSD_SIGMADELTA2_REG	Duty Cycle Configuration Register of SDM2	0x0008	R/W
GPIOSD_SIGMADELTA3_REG	Duty Cycle Configuration Register of SDM3	0x000C	R/W
GPIOSD_SIGMADELTA_CG_REG	Clock Gating Configuration Register	0x0020	R/W
GPIOSD_SIGMADELTA_MISC_REG	MISC Register	0x0024	R/W
Version register			
GPIOSD_SIGMADELTA_VERSION_REG	Version Control Register	0x0028	R/W

4.14 Registers

The addresses in this section are relative to GPIO Matrix, IO MUX and SDM base addresses provided in Table 3-4 in Chapter 3 *System and Memory*.

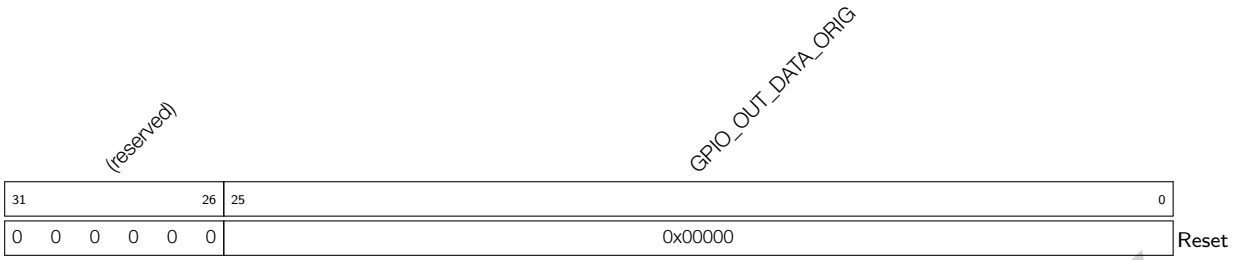
4.14.1 GPIO Matrix Registers

Register 4.1. GPIO_BT_SELECT_REG (0x0000)



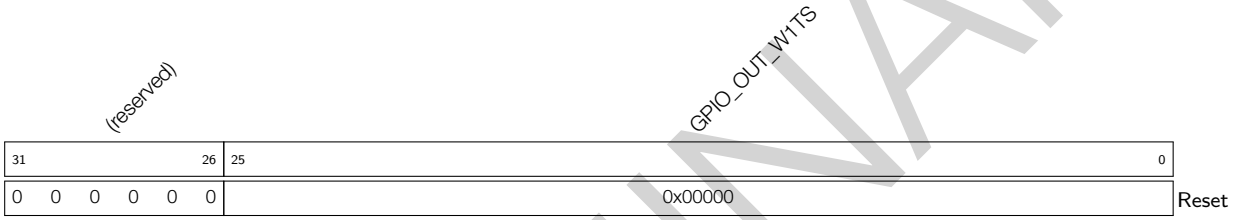
GPIO_BT_SEL Reserved (R/W)

Register 4.2. GPIO_OUT_REG (0x0004)



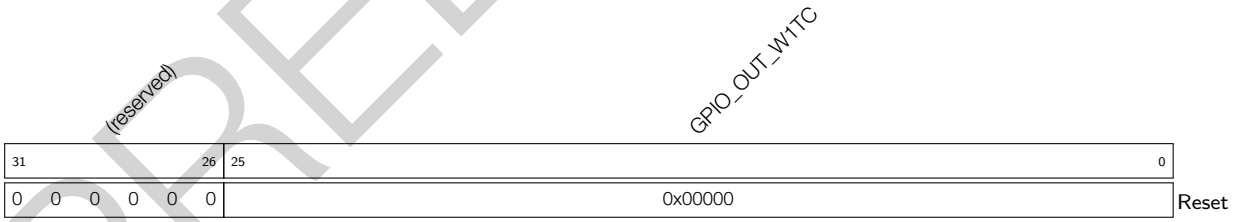
GPIO_OUT_DATA_ORIG GPIO0 ~ 21 output value in simple GPIO output mode. The values of bit0 ~ bit21 correspond to the output value of GPIO0 ~ GPIO21 respectively, and bit22 ~ bit25 are invalid. (R/W/SS)

Register 4.3. GPIO_OUT_W1TS_REG (0x0008)



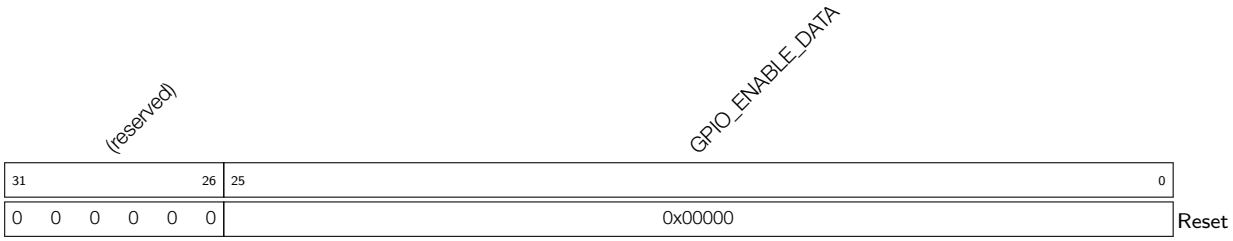
GPIO_OUT_W1TS GPIO0 ~ 21 output set register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. If the value 1 is written to a bit here, the corresponding bit in GPIO_OUT_REG will be set to 1. Recommended operation: use this register to set GPIO_OUT_REG. (WT)

Register 4.4. GPIO_OUT_W1TC_REG (0x000C)



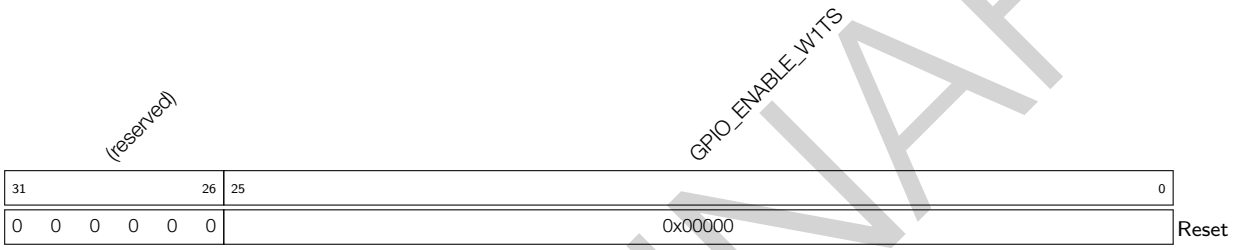
GPIO_OUT_W1TC GPIO0 ~ 21 output clear register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. If the value 1 is written to a bit here, the corresponding bit in GPIO_OUT_REG will be cleared. Recommended operation: use this register to clear GPIO_OUT_REG. (WT)

Register 4.5. GPIO_ENABLE_REG (0x0020)



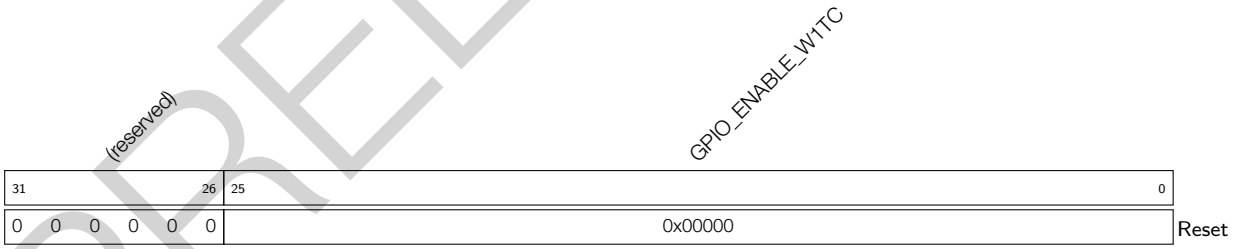
GPIO_ENABLE_DATA GPIO output enable register for GPIO0 ~ 21. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. (R/W/SS)

Register 4.6. GPIO_ENABLE_W1TS_REG (0x0024)



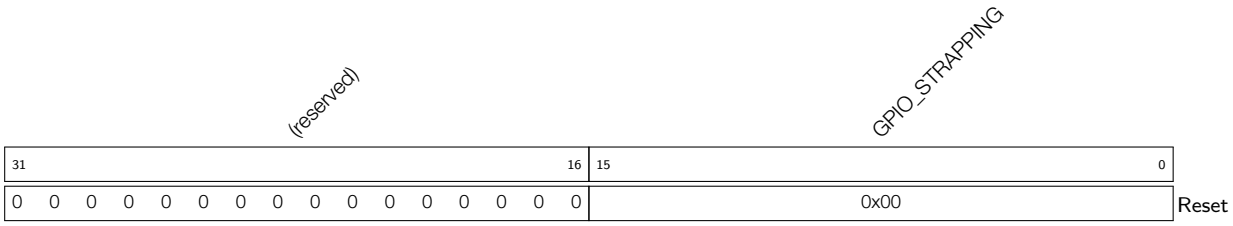
GPIO_ENABLE_W1TS GPIO0 ~ 21 output enable set register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. If the value 1 is written to a bit here, the corresponding bit in GPIO_ENABLE_REG will be set to 1. Recommended operation: use this register to set GPIO_ENABLE_REG. (WT)

Register 4.7. GPIO_ENABLE_W1TC_REG (0x0028)



GPIO_ENABLE_W1TC GPIO0 ~ 21 output enable clear register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. If the value 1 is written to a bit here, the corresponding bit in GPIO_ENABLE_REG will be cleared. Recommended operation: use this register to clear GPIO_ENABLE_REG. (WT)

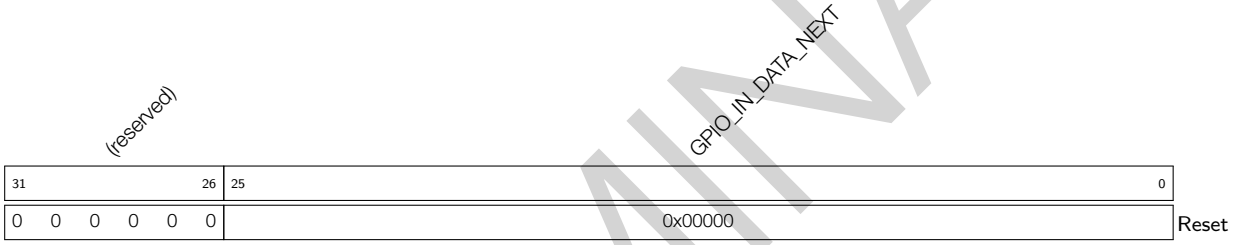
Register 4.8. GPIO_STRAP_REG (0x0038)



GPIO_STRAPPING GPIO strapping values. (RO)

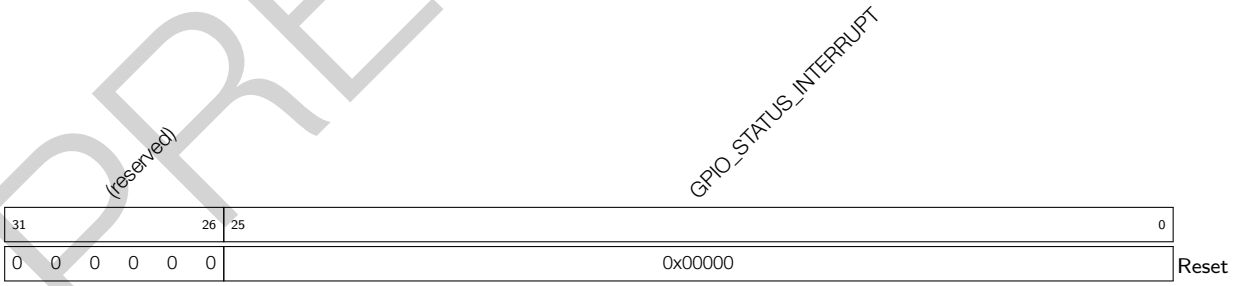
- bit 0: GPIO2
- bit 2: GPIO8
- bit 3: GPIO9

Register 4.9. GPIO_IN_REG (0x003C)



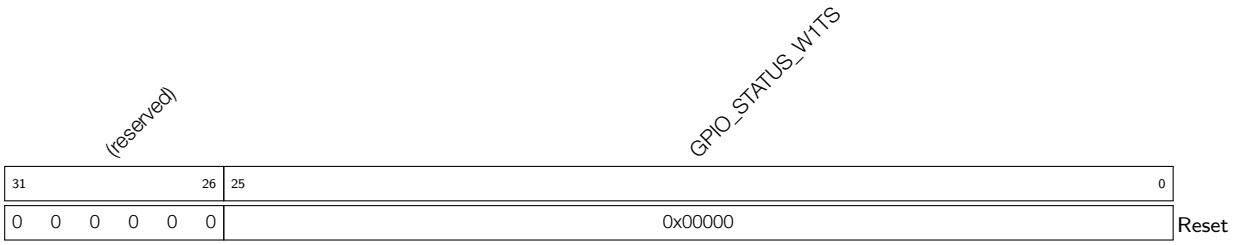
GPIO_IN_DATA_NEXT GPIO0 ~ 21 input value. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. Each bit represents a pin input value, 1 for high level and 0 for low level. (RO)

Register 4.10. GPIO_STATUS_REG (0x0044)



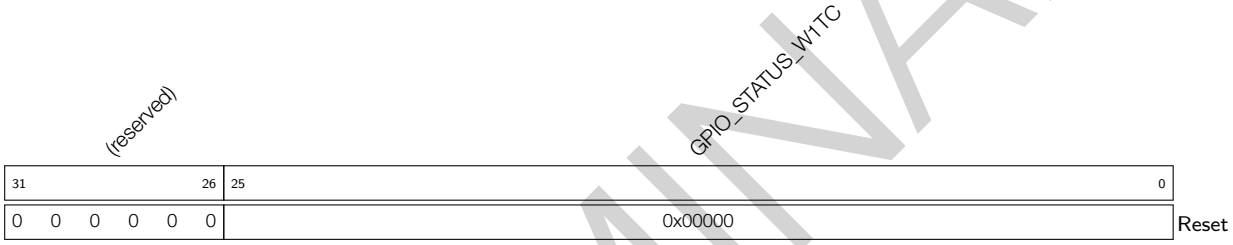
GPIO_STATUS_INTERRUPT GPIO0 ~ 21 interrupt status register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. (R/W/SS)

Register 4.11. GPIO_STATUS_W1TS_REG (0x0048)



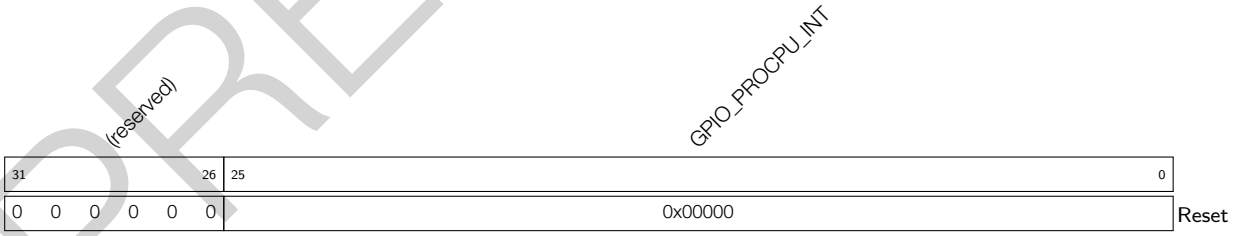
GPIO_STATUS_W1TS GPIO0 ~ 21 interrupt status set register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. If the value 1 is written to a bit here, the corresponding bit in GPIO_STATUS_INTERRUPT will be set to 1. Recommended operation: use this register to set GPIO_STATUS_INTERRUPT. (WT)

Register 4.12. GPIO_STATUS_W1TC_REG (0x004C)



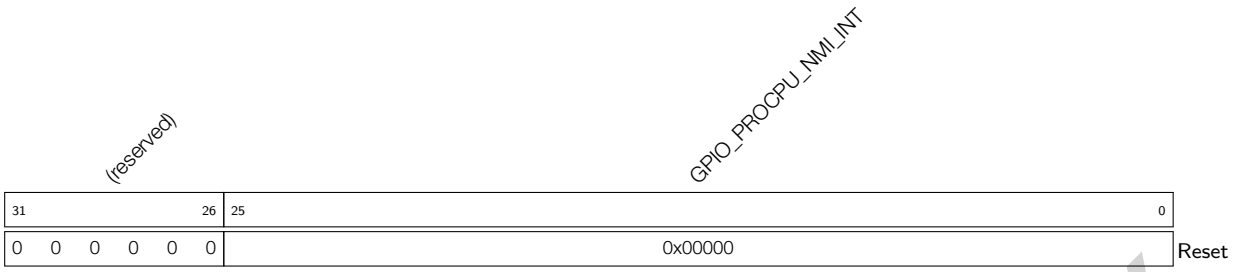
GPIO_STATUS_W1TC GPIO0 ~ 21 interrupt status clear register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. If the value 1 is written to a bit here, the corresponding bit in GPIO_STATUS_INTERRUPT will be cleared. Recommended operation: use this register to clear GPIO_STATUS_INTERRUPT. (WT)

Register 4.13. GPIO_PCPU_INT_REG (0x005C)



GPIO_PROCPU_INT GPIO0 ~ 21 PRO_CPU interrupt status. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. This interrupt status is corresponding to the bit in GPIO_STATUS_REG when assert (high) enable signal (bit13 of GPIO_PIN*n*_REG). (RO)

Register 4.14. GPIO_PCPU_NMI_INT_REG (0x0060)



GPIO_PROCPU_NMI_INT GPIO0 ~ 21 PRO_CPU non-maskable interrupt status. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. This interrupt status is corresponding to the bit in GPIO_STATUS_REG when assert (high) enable signal (bit 14 of GPIO_PIN*n*_REG). (RO)

Register 4.15. GPIO_PIN*n*_REG (*n*: 0-21) (0x0074+4n*)**

GPIO_PIN*n*_SYNC2_BYPASS For the second stage synchronization, GPIO input data can be synchronized on either edge of the APB clock. 0: no synchronization; 1: synchronized on falling edge; 2 and 3: synchronized on rising edge. (R/W)

GPIO_PIN*n*_PAD_DRIVER pin drive selection. 0: normal output; 1: open drain output. (R/W)

GPIO_PIN*n*_SYNC1_BYPASS For the first stage synchronization, GPIO input data can be synchronized on either edge of the APB clock. 0: no synchronization; 1: synchronized on falling edge; 2 and 3: synchronized on rising edge. (R/W)

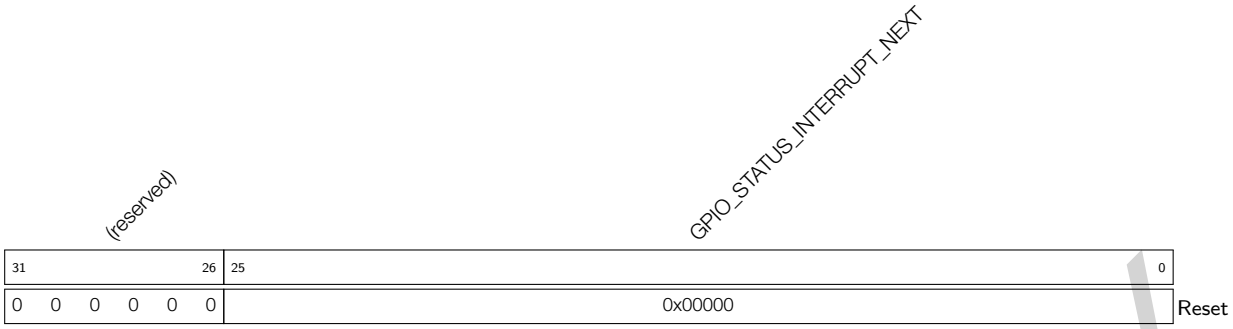
GPIO_PIN*n*_INT_TYPE Interrupt type selection. 0: GPIO interrupt disabled; 1: rising edge trigger; 2: falling edge trigger; 3: any edge trigger; 4: low level trigger; 5: high level trigger. (R/W)

GPIO_PIN*n*_WAKEUP_ENABLE GPIO wake-up enable bit, only wakes up the CPU from Light-sleep. (R/W)

GPIO_PIN*n*_CONFIG reserved (R/W)

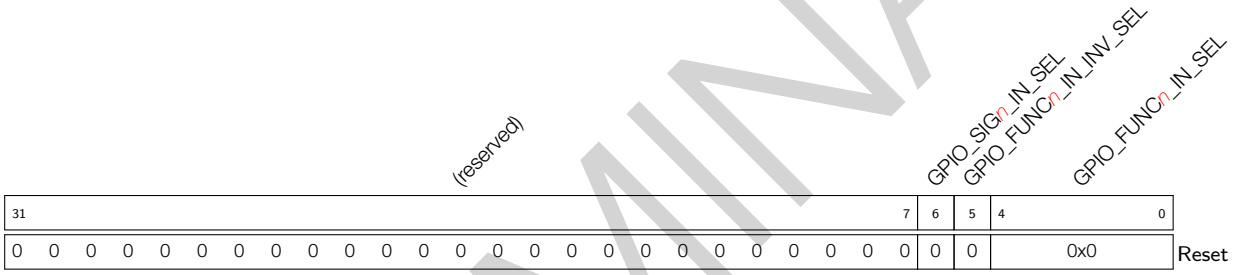
GPIO_PIN*n*_INT_ENA Interrupt enable bits. bit13: CPU interrupt enabled; bit14: CPU non-maskable interrupt enabled. (R/W)

Register 4.16. GPIO_STATUS_NEXT_REG (0x014C)



GPIO_STATUS_INTERRUPT_NEXT Interrupt source signal of GPIO0 ~ 21, could be rising edge interrupt, falling edge interrupt, level sensitive interrupt and any edge interrupt. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. (RO)

Register 4.17. GPIO_FUNC*n*_IN_SEL_CFG_REG (*n*: 0-127) (0x0154+4n*)**

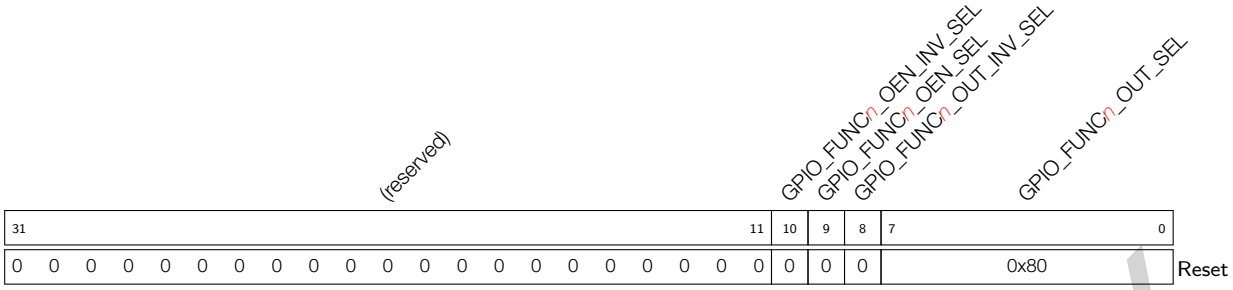


GPIO_FUNC*n*_IN_SEL Selection control for peripheral input signal *n*, selects a pin from the 22 GPIO matrix pins to connect this input signal. Or selects 0x1e for a constantly high input or 0x1f for a constantly low input. (R/W)

GPIO_FUNC*n*_IN_INV_SEL Invert the input value. 1: invert enabled; 0: invert disabled. (R/W)

GPIO_SIG*n*_IN_SEL Bypass GPIO matrix. 1: route signals via GPIO matrix, 0: connect signals directly to peripheral configured in IO MUX. (R/W)

Register 4.18. GPIO_FUNC*n*_OUT_SEL_CFG_REG (*n*: 0-21) (0x0554+4n*)**



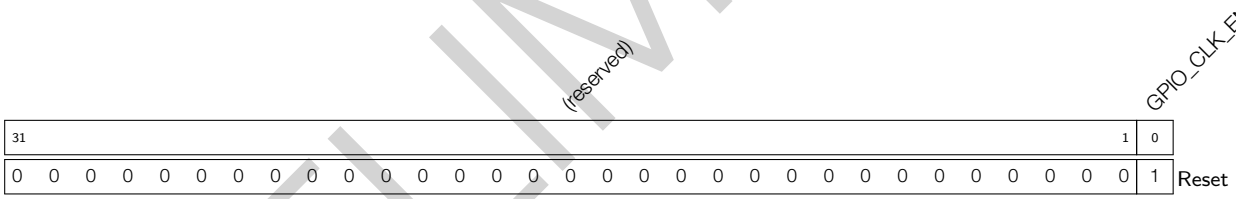
GPIO_FUNC*n*_OUT_SEL Selection control for GPIO output *n*. If a value *Y* (0<=Y<128) is written to this field, the peripheral output signal *Y* will be connected to GPIO output *X*. If a value 128 is written to this field, bit *n* of GPIO_OUT_REG and GPIO_ENABLE_REG will be selected as the output value and output enable. (R/W)

GPIO_FUNC*n*_OUT_INV_SEL 0: Do not invert the output value; 1: Invert the output value. (R/W)

GPIO_FUNC*n*_OEN_SEL 0: Use output enable signal from peripheral; 1: Force the output enable signal to be sourced from bit *n* of GPIO_ENABLE_REG. (R/W)

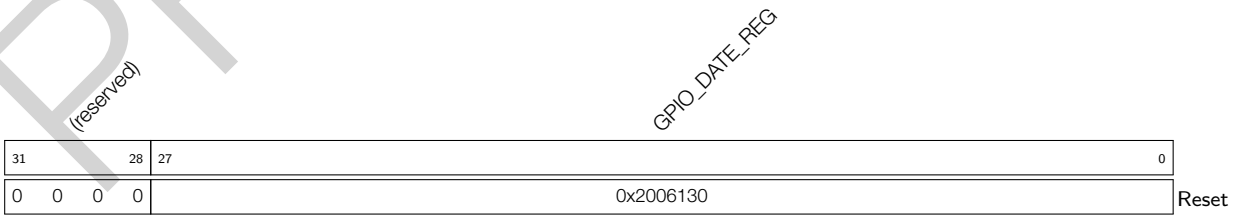
GPIO_FUNC*n*_OEN_INV_SEL 0: Do not invert the output enable signal; 1: Invert the output enable signal. (R/W)

Register 4.19. GPIO_CLOCK_GATE_REG (0x062C)



GPIO_CLK_EN Clock gating enable bit. If set to 1, the clock is free running. (R/W)

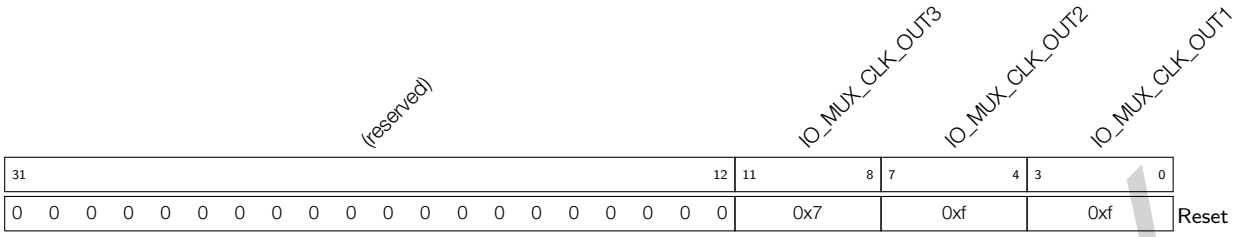
Register 4.20. GPIO_DATE_REG (0x06FC)



GPIO_DATE_REG Version control register (R/W)

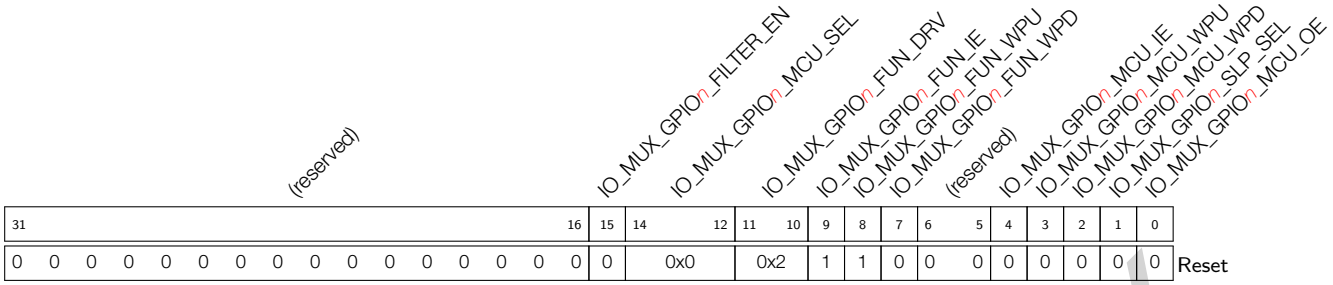
4.14.2 IO MUX Registers

Register 4.21. IO_MUX_PIN_CTRL_REG (0x0000)



IO_MUX_CLK_OUT*x* If you want to output clock for I2S to CLK_OUT_out*x*, set IO_MUX_CLK_OUT*x* to 0x0. CLK_OUT_out*x* can be found in Table 4-1. (R/W)

Register 4.22. IO_MUX_GPIO*n*_REG (*n*: 0-21) (0x0004+4n*)**



IO_MUX_GPIO*n*_MCU_OE Output enable of the pin in sleep mode. 1: output enabled; 0: output disabled. (R/W)

IO_MUX_GPIO*n*_SLP_SEL Sleep mode selection of this pin. Set to 1 to put the pin in sleep mode. (R/W)

IO_MUX_GPIO*n*_MCU_WPD Pull-down enable of the pin in sleep mode. 1: internal pull-down enabled; 0: internal pull-down disabled. (R/W)

IO_MUX_GPIO*n*_MCU_WPU Pull-up enable of the pin during sleep mode. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

IO_MUX_GPIO*n*_MCU_IE Input enable of the pin during sleep mode. 1: input enabled; 0: input disabled. (R/W)

IO_MUX_GPIO*n*_FUN_WPD Pull-down enable of the pin. 1: internal pull-down enabled; 0: internal pull-down disabled. (R/W)

IO_MUX_GPIO*n*_FUN_WPU Pull-up enable of the pin. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

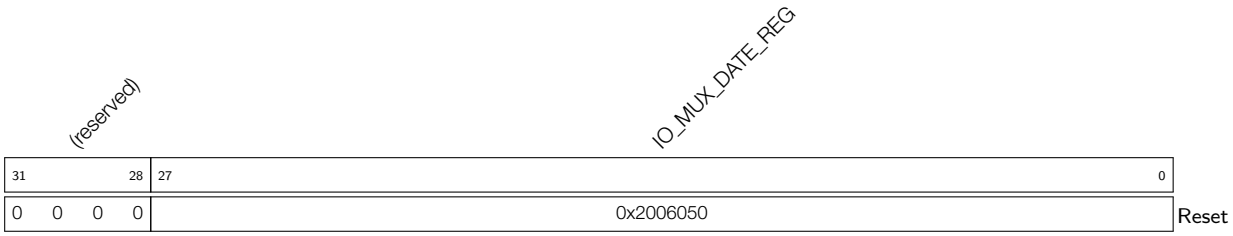
IO_MUX_GPIO*n*_FUN_IE Input enable of the pin. 1: input enabled; 0: input disabled. (R/W)

IO_MUX_GPIO*n*_FUN_DRV Select the drive strength of the pin. 0: ~5 mA; 1: ~ 10 mA; 2: ~ 20 mA; 3: ~40mA. (R/W)

IO_MUX_GPIO*n*_MCU_SEL Select IO MUX function for this signal. 0: Select Function 0; 1: Select Function 1; etc. (R/W)

IO_MUX_GPIO*n*_FILTER_EN Enable filter for pin input signals. 1: Filter enabled; 2: Filter disabled. (R/W)

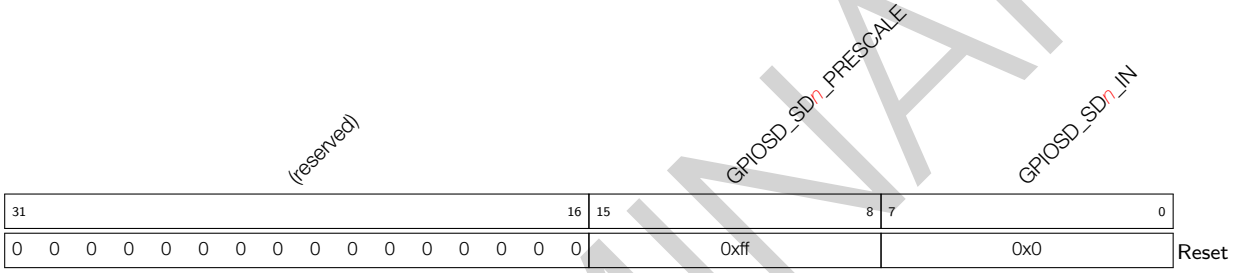
Register 4.23. IO_MUX_DATE_REG (0x00FC)



IO_MUX_DATE_REG Version control register (R/W)

4.14.3 SDM Output Registers

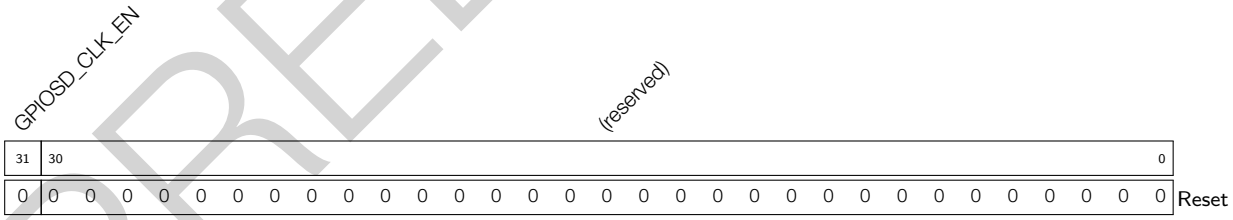
Register 4.24. GPIOSD_SIGMADELTA*n*_REG (*n*: 0-3) (0x0000+4n*)**



GPIOSD_SD*n*_IN This field is used to configure the duty cycle of sigma delta modulation output. (R/W)

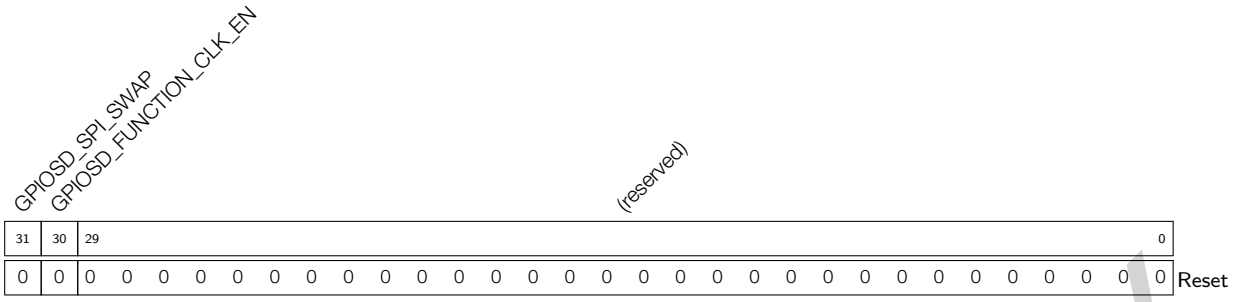
GPIOSD_SD*n*_PRESCALE This field is used to set a divider value to divide APB clock. (R/W)

Register 4.25. GPIOSD_SIGMADELTA_CG_REG (0x0020)



GPIOSD_CLK_EN Clock enable bit of configuration registers for sigma delta modulation. (R/W)

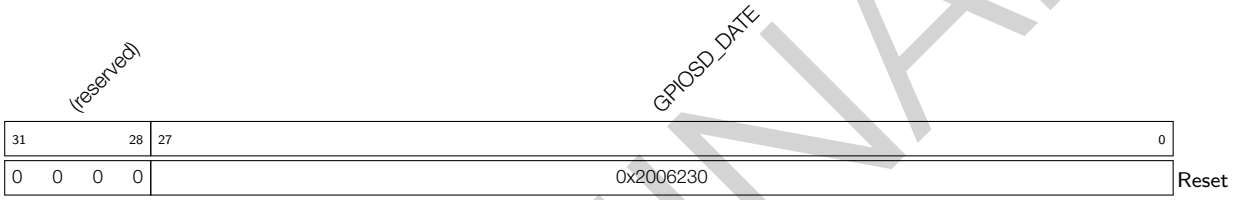
Register 4.26. GPIOSD_SIGMADELTA_MISC_REG (0x0024)



GPIOSD_FUNCTION_CLK_EN Clock enable bit of sigma delta modulation. (R/W)

GPIOSD_SPI_SWAP Reserved. (R/W)

Register 4.27. GPIOSD_SIGMADELTA_VERSION_REG (0x0028)



GPIOSD_DATE Version Control Register. (R/W)

5 SHA Accelerator

5.1 Introduction

ESP32-C3 integrates an SHA accelerator, which is a hardware device that speeds up SHA algorithm significantly, compared to SHA algorithm implemented solely in software. The SHA accelerator integrated in ESP32-C3 has two working modes, which are Typical SHA and DMA-SHA.

5.2 Features

The following functionality is supported:

- The following hash algorithms introduced in FIPS PUB 180-4 Spec.
  - SHA-1
  - SHA-224
  - SHA-256
- Two working modes
  - Typical SHA
  - DMA-SHA
- Interleaved function when working in Typical SHA working mode
- Interrupt function when working in DMA-SHA working mode

5.3 Working Modes

The SHA accelerator integrated in ESP32-C3 has two working modes.

- Typical SHA Working Mode: all the data is written and read via CPU directly.
- DMA-SHA Working Mode: all the data is read via DMA. That is, users can configure the DMA controller to read all the data needed for hash operation, thus releasing CPU for completing other tasks.

Users can start the SHA accelerator with different working modes by configuring registers SHA_START_REG and SHA_DMA_START_REG. For details, please see Table 5-1.

Table 5-1. SHA Accelerator Working Mode

Working Mode	Configuration Method
Typical SHA	Set SHA_START_REG to 1
DMA-SHA	Set SHA_DMA_START_REG to 1

Users can choose hash algorithms by configuring the SHA_MODE_REG register. For details, please see Table 5-2.

Table 5-2. SHA Hash Algorithm Selection

Hash Algorithm	SHA_MODE_REG Configuration
SHA-1	0
SHA-224	1
SHA-256	2

5.4 Function Description

SHA accelerator can generate the message digest via two steps: Preprocessing and Hash operation.

5.4.1 Preprocessing

Preprocessing consists of three steps: padding the message, parsing the message into message blocks and setting the initial hash value.

5.4.1.1 Padding the Message

The SHA accelerator can only process message blocks of 512 bits. Thus, all the messages should be padded to a multiple of 512 bits before the hash task.

Suppose that the length of the message M is m bits. Then M shall be padded as introduced below:

1. First, append the bit "1" to the end of the message;
2. Second, append k bits of zeros, where k is the smallest, non-negative solution to the equation $m + 1 + k \equiv 448 \ mod \ 512$;
3. Last, append the 64-bit block of value equal to the number m expressed using a binary representation.

For more details, please refer to Section "5.1 Padding the Message" in FIPS PUB 180-4 Spec.

5.4.1.2 Parsing the Message

The message and its padding must be parsed into N 512-bit blocks, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

During the task, all the message blocks are written into the SHA_M_*n*_REG: $M_0^{(i)}$ is stored in SHA_M_0_REG, $M_1^{(i)}$ stored in SHA_M_1_REG, ..., and $M_{15}^{(i)}$ stored in SHA_M_15_REG.

Note:

For more information about "message block", please refer to Section "2.1 Glossary of Terms and Acronyms" in FIPS PUB 180-4 Spec.

5.4.1.3 Initial Hash Value

Before hash task begins for any secure hash algorithms, the initial Hash value H(0) must be set based on different algorithms. However, the SHA accelerator uses the initial Hash values (constant C) stored in the hardware for hash tasks.

5.4.2 Hash Task Process

After the preprocessing, the ESP32-C3 SHA accelerator starts to hash a message M and generates message digest of different lengths, depending on different hash algorithms. As described above, the ESP32-C3 SHA accelerator supports two working modes, which are Typical SHA and DMA-SHA. The operation process for the SHA accelerator under two working modes is described in the following subsections.

5.4.2.1 Typical SHA Mode Process

Usually, the SHA accelerator will process all blocks of a message and produce a message digest before starting the computation of the next message digest.

However, ESP32-C3 SHA also supports optional "interleaved" message digest calculation. Users can insert new calculation (both Typical SHA and DMA-SHA) each time the SHA accelerator completes a sequence of operations.

- In Typical SHA mode, this can be done after each individual message block.
- In DMA-SHA mode, this can be done after a full sequence of DMA operations is complete.

Specifically, users can read out the message digest from registers SHA_H_*n*_REG after completing part of a message digest calculation, and use the SHA accelerator for a different calculation. After the different calculation completes, users can restore the previous message digest to registers SHA_H_*n*_REG, and resume the accelerator with the previously paused calculation.

Typical SHA Process

1. Select a hash algorithm.
   - Configure the SHA_MODE_REG register based on Table 5-2.
2. Process the current message block [1].
   - Write the message block in registers SHA_M_*n*_REG.
3. Start the SHA accelerator.
   - If this is the first time to execute this step, set the SHA_START_REG register to 1 to start the SHA accelerator. In this case, the accelerator uses the initial hash value stored in hardware for a given algorithm configured in Step 1 to start the calculation;
   - If this is not the first time to execute this step[2], set the SHA_CONTINUE_REG register to 1 to start the SHA accelerator. In this case, the accelerator uses the hash value stored in the SHA_H_*n*_REG register to start calculation.
4. Check the progress of the current message block.
   - Poll register SHA_BUSY_REG until the content of this register becomes 0, indicating the accelerator has completed the calculation for the current message block and now is in the "idle" status [3].

5. Decide if you have more message blocks to process:
    - If yes, please go back to Step 2.
    - Otherwise, please continue.
6. Obtain the message digest.
    - Read the message digest from registers SHA_H_*n*_REG.

Note:

1. In this step, the software can also write the next message block (to be processed) in registers SHA_M_*n*_REG, if any, while the hardware starts SHA calculation, to save time.
2. You are resuming the SHA accelerator with the previously paused calculation.
3. Here you can decide if you want to insert other calculations. If yes, please go to the process for interleaved calculations for details.

As mentioned above, ESP32-C3 SHA accelerator supports **"interleaving" calculation under the Typical SHA working mode**.

The process to implement interleaved calculation is described below.

1. Prepare to hand the SHA accelerator over for an interleaved calculation by storing the following data of the previous calculation.
    - The selected hash algorithm stored in the SHA_MODE_REG register.
    - The message digest stored in registers SHA_H_*n*_REG.
2. Perform the interleaved calculation. For the detailed process of the interleaved calculation, please refer to Typical SHA process or DMA-SHA process, depending on the working mode of your interleaved calculation.
3. Prepare to hand the SHA accelerator back to the previously paused calculation by restoring the following data of the previous calculation.
    - Write the previously stored hash algorithm back to register SHA_MODE_REG.
    - Write the previously stored message digest back to registers SHA_H_*n*_REG.
4. Write the next message block from the previous paused calculation in registers SHA_M_*n*_REG, and set the SHA_CONTINUE_REG register to 1 to restart the SHA accelerator with the previously paused calculation.

5.4.2.2 DMA-SHA Mode Process

ESP32-C3 SHA accelerator does not support "interleaving" message digest calculation at the level of individual message blocks when using DMA, which means you cannot insert new calculation before a complete DMA-SHA process (of one or more message blocks) completes. In this case, users who need interleaved operation are recommended to divide the message blocks and perform several DMA-SHA calculations, instead of trying to compute all the messages in one go.

Single DMA-SHA calculation supports up to 63 data blocks.

In contrast to the Typical SHA working mode, when the SHA accelerator is working under the DMA-SHA mode, all data read are completed via DMA. Therefore, users are required to configure the DMA controller.

DMA-SHA process

1. Select a hash algorithm.
   - Select a hash algorithm by configuring the SHA_MODE_REG register. For details, please refer to Table 5-2.
2. Configure the SHA_INT_ENA_REG register to enable or disable interrupt (Set 1 to enable).
3. Configure the number of message blocks.
   - Write the number of message blocks M to the SHA_DMA_BLOCK_NUM_REG register.
4. Start the DMA-SHA calculation.
   - If the current DMA-SHA calculation follows a previous calculation, firstly write the message digest from the previous calculation to registers SHA_H_*n*_REG, then write 1 to register SHA_DMA_CONTINUE_REG to start SHA accelerator;
   - Otherwise, write 1 to register SHA_DMA_START_REG to start the accelerator.
5. Wait till the completion of the DMA-SHA calculation, which happens when:
   - The content of SHA_BUSY_REG register becomes 0, or
   - An SHA interrupt occurs. In this case, please clear interrupt by writing 1 to the SHA_INT_CLEAR_REG register.
6. Obtain the message digest:
   - Read the message digest from registers SHA_H_*n*_REG.

5.4.3 Message Digest

After the hash task completes, the SHA accelerator writes the message digest from the task to registers SHA_H_*n*_REG(*n*: 0~7). The lengths of the generated message digest are different depending on different hash algorithms. For details, see Table 5-3 below:

Table 5-3. The Storage and Length of Message Digest from Different Algorithms

Hash Algorithm	Length of Message Digest (in bits)	Storage[1]
SHA-1	160	SHA_H_0_REG ~ SHA_H_4_REG
SHA-224	224	SHA_H_0_REG ~ SHA_H_6_REG
SHA-256	256	SHA_H_0_REG ~ SHA_H_7_REG

[1] The message digest is stored in registers from most significant bits to the least significant bits, with the first word stored in register SHA_H_0_REG and the second word stored in register SHA_H_1_REG... For details, please see subsection 5.4.1.2.

5.4.4 Interrupt

SHA accelerator supports interrupt on the completion of message digest calculation when working in the DMA-SHA mode. To enable this function, write 1 to register SHA_INT_ENA_REG. Note that the interrupt should be cleared by software after use via setting the SHA_INT_CLEAR_REG register to 1.

5.5 Register Summary

The addresses in this section are relative to the SHA accelerator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

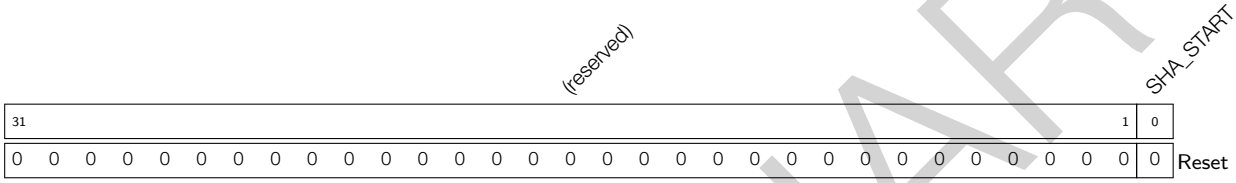
Name	Description	Address	Access
Control/Status registers			
SHA_CONTINUE_REG	Continues SHA operation (only effective in Typical SHA mode)	0x0014	WO
SHA_BUSY_REG	Indicates if SHA Accelerator is busy or not	0x0018	RO
SHA_DMA_START_REG	Starts the SHA accelerator for DMA-SHA operation	0x001C	WO
SHA_START_REG	Starts the SHA accelerator for Typical SHA operation	0x0010	WO
SHA_DMA_CONTINUE_REG	Continues SHA operation (only effective in DMA-SHA mode)	0x0020	WO
SHA_INT_CLEAR_REG	DMA-SHA interrupt clear register	0x0024	WO
SHA_INT_ENA_REG	DMA-SHA interrupt enable register	0x0028	R/W
Version Register			
SHA_DATE_REG	Version control register	0x002C	R/W
Configuration Registers			
SHA_MODE_REG	Defines the algorithm of SHA accelerator	0x0000	R/W
Data Registers			
SHA_DMA_BLOCK_NUM_REG	Block number register (only effective for DMA-SHA)	0x000C	R/W
SHA_H_0_REG	Hash value	0x0040	R/W
SHA_H_1_REG	Hash value	0x0044	R/W
SHA_H_2_REG	Hash value	0x0048	R/W
SHA_H_3_REG	Hash value	0x004C	R/W
SHA_H_4_REG	Hash value	0x0050	R/W
SHA_H_5_REG	Hash value	0x0054	R/W
SHA_H_6_REG	Hash value	0x0058	R/W
SHA_H_7_REG	Hash value	0x005C	R/W
SHA_M_0_REG	Message	0x0080	R/W
SHA_M_1_REG	Message	0x0084	R/W
SHA_M_2_REG	Message	0x0088	R/W
SHA_M_3_REG	Message	0x008C	R/W
SHA_M_4_REG	Message	0x0090	R/W
SHA_M_5_REG	Message	0x0094	R/W
SHA_M_6_REG	Message	0x0098	R/W
SHA_M_7_REG	Message	0x009C	R/W
SHA_M_8_REG	Message	0x00A0	R/W
SHA_M_9_REG	Message	0x00A4	R/W
SHA_M_10_REG	Message	0x00A8	R/W
SHA_M_11_REG	Message	0x00AC	R/W

Name	Description	Address	Access
SHA_M_12_REG	Message	0x00B0	R/W
SHA_M_13_REG	Message	0x00B4	R/W
SHA_M_14_REG	Message	0x00B8	R/W
SHA_M_15_REG	Message	0x00BC	R/W

5.6 Registers

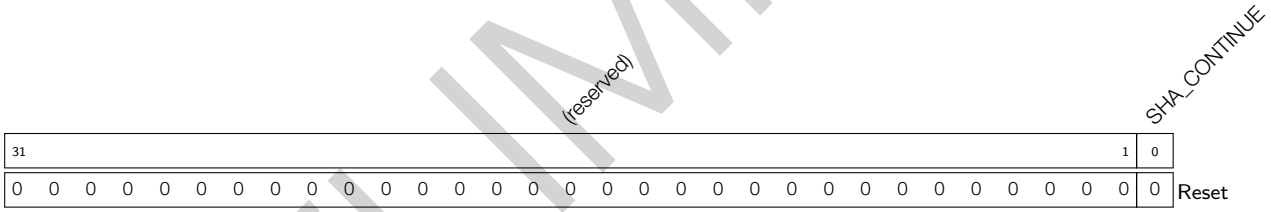
The addresses in this section are relative to the SHA accelerator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Register 5.1. SHA_START_REG (0x0010)



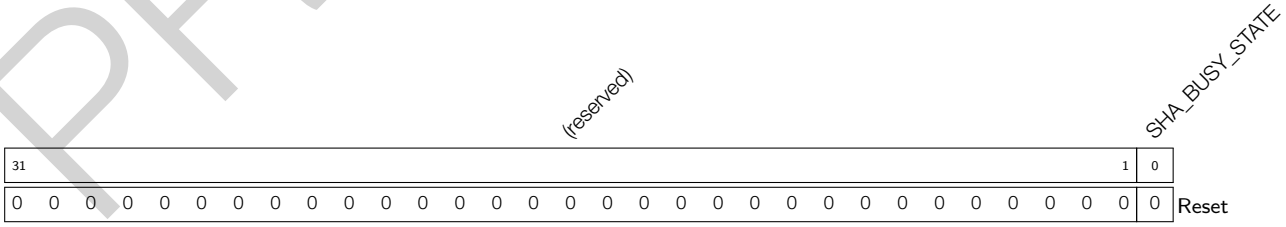
SHA_START Write 1 to start Typical SHA calculation. (WO)

Register 5.2. SHA_CONTINUE_REG (0x0014)



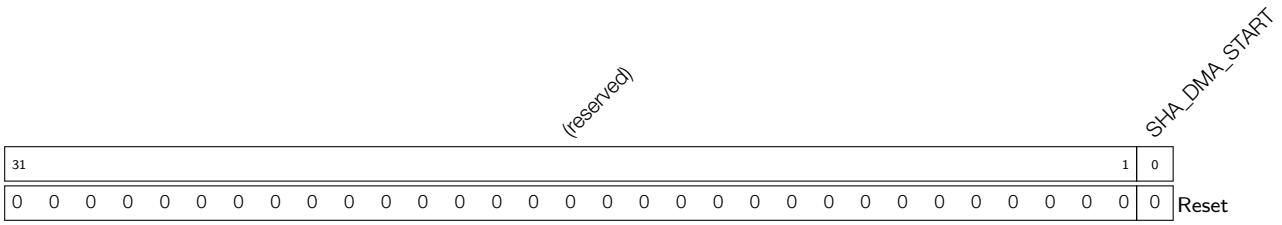
SHA_CONTINUE Write 1 to continue Typical SHA calculation. (WO)

Register 5.3. SHA_BUSY_REG (0x0018)



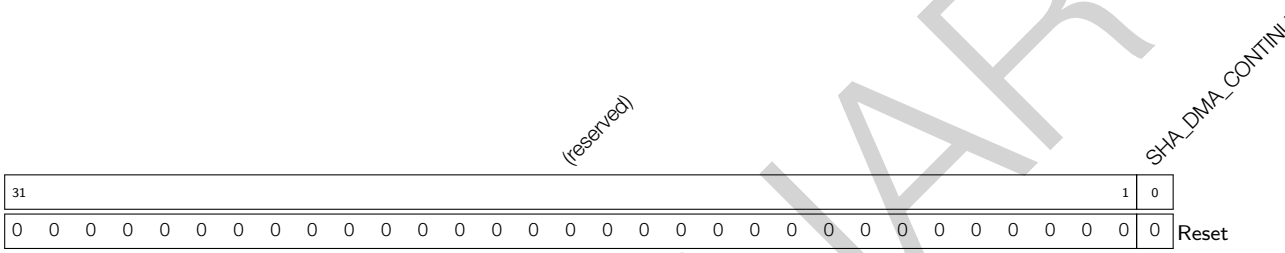
SHA_BUSY_STATE Indicates the states of SHA accelerator. (RO) 1'h0: idle 1'h1: busy

Register 5.4. SHA_DMA_START_REG (0x001C)



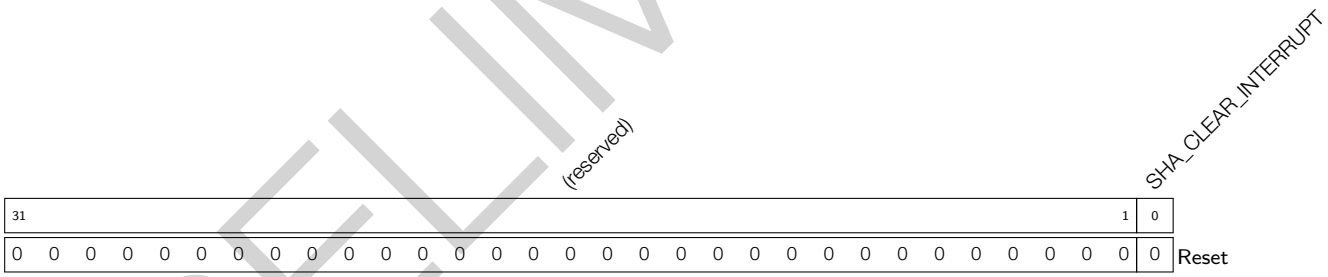
SHA_DMA_START Write 1 to start DMA-SHA calculation. (WO)

Register 5.5. SHA_DMA_CONTINUE_REG (0x0020)



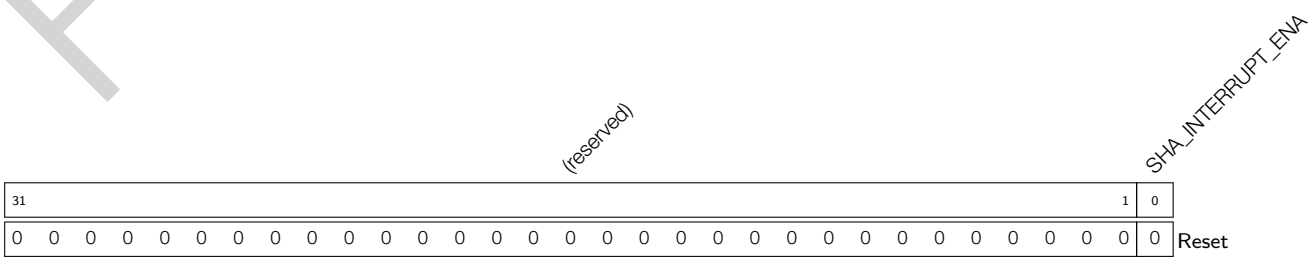
SHA_DMA_CONTINUE Write 1 to continue DMA-SHA calculation. (WO)

Register 5.6. SHA_INT_CLEAR_REG (0x0024)



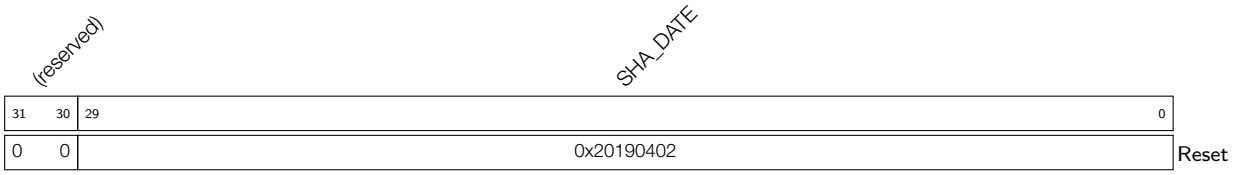
SHA_CLEAR_INTERRUPT Clears DMA-SHA interrupt. (WO)

Register 5.7. SHA_INT_ENA_REG (0x0028)



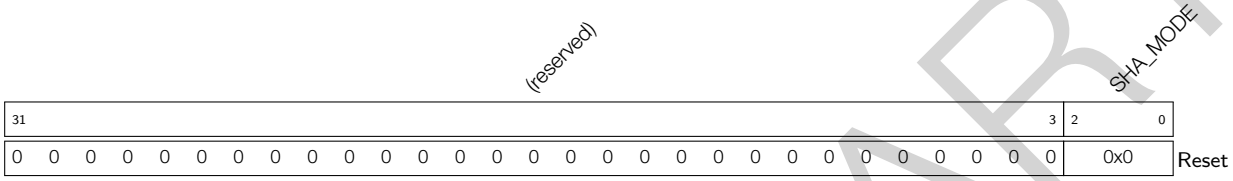
SHA_INTERRUPT_ENA Enables DMA-SHA interrupt. (R/W)

Register 5.8. SHA_DATE_REG (0x002C)



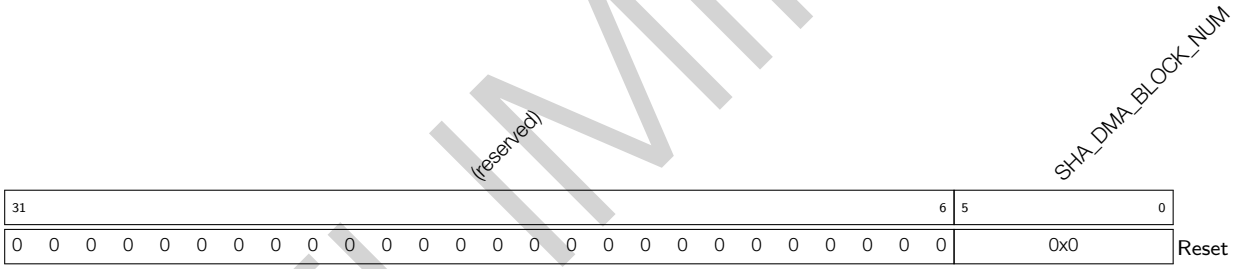
SHA_DATE Version control register. (R/W)

Register 5.9. SHA_MODE_REG (0x0000)



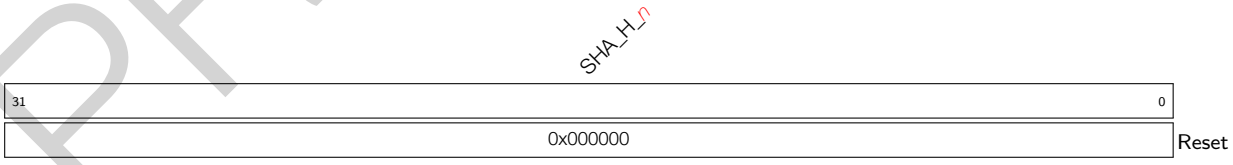
SHA_MODE Defines the SHA algorithm. For details, please see Table 5-2. (R/W)

Register 5.10. SHA_DMA_BLOCK_NUM_REG (0x000C)



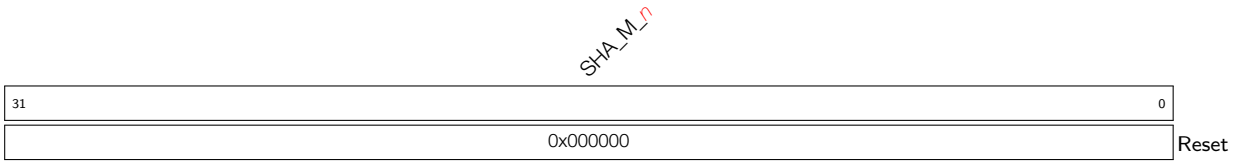
SHA_DMA_BLOCK_NUM Defines the DMA-SHA block number. (R/W)

Register 5.11. SHA_H_*n*_REG (*n*: 0-7) (0x0040+4n*)**



SHA_H_*n* Stores the *n*th 32-bit piece of the Hash value. (R/W)

Register 5.12. SHA_M_*n*_REG (*n*: 0-15) (0x0080+4n*)**



SHA_M_*n* Stores the *n*th 32-bit piece of the message. (R/W)

6 AES Accelerator

6.1 Introduction

ESP32-C3 integrates an Advanced Encryption Standard (AES) Accelerator, which is a hardware device that speeds up AES Algorithm significantly, compared to AES algorithms implemented solely in software. The AES Accelerator integrated in ESP32-C3 has two working modes, which are Typical AES and DMA-AES.

6.2 Features

The following functionality is supported:

- Typical AES working mode
  - AES-128/AES-256 encryption and decryption
- DMA-AES working mode
  - AES-128/AES-256 encryption and decryption
  - Block cipher mode
    - ECB (Electronic Codebook)
    - CBC (Cipher Block Chaining)
    - OFB (Output Feedback)
    - CTR (Counter)
    - CFB8 (8-bit Cipher Feedback)
    - CFB128 (128-bit Cipher Feedback)
  - Interrupt on completion of computation

6.3 AES Working Modes

The AES Accelerator integrated in ESP32-C3 has two working modes, which are Typical AES and DMA-AES.

- Typical AES Working Mode:
  - Supports encryption and decryption using cryptographic keys of 128 and 256 bits, specified in NIST FIPS 197.

  In this working mode, the plaintext and ciphertext is written and read via CPU directly.
- DMA-AES Working Mode:
  - Supports encryption and decryption using cryptographic keys of 128 and 256 bits, specified in NIST FIPS 197;
  - Supports block cipher modes ECB/CBC/OFB/CTR/CFB8/CFB128 under NIST SP 800-38A.

  In this working mode, the plaintext and ciphertext are written and read via DMA. An interrupt will be generated when operation completes.

Users can choose the working mode for AES accelerator by configuring the AES_DMA_ENABLE_REG register according to Table 6-1 below.

Table 6-1. AES Accelerator Working Mode

AES_DMA_ENABLE_REG	Working Mode
0	Typical AES
1	DMA-AES

Users can choose the length of cryptographic keys and encryption / decryption by configuring the AES_MODE_REG register according to Table 6-2 below.

Table 6-2. Key Length and Encryption/Decryption

AES_MODE_REG[2:0]	Key Length and Encryption / Decryption
0	AES-128 encryption
1	reserved
2	AES-256 encryption
3	reserved
4	AES-128 decryption
5	reserved
6	AES-256 decryption
7	reserved

For detailed introduction on these two working modes, please refer to Section 6.4 and Section 6.5 below.

6.4 Typical AES Working Mode

In the Typical AES working mode, users can check the working status of the AES accelerator by inquiring the AES_STATE_REG register and comparing the return value against the Table 6-3 below.

Table 6-3. Working Status under Typical AES Working Mode

AES_STATE_REG	Status	Description
0	IDLE	The AES accelerator is idle or completed operation.
1	WORK	The AES accelerator is in the middle of an operation.

6.4.1 Key, Plaintext, and Ciphertext

The encryption or decryption key is stored in AES_KEY_*n*_REG, which is a set of eight 32-bit registers.

- For AES-128 encryption/decryption, the 128-bit key is stored in AES_KEY_0_REG ~ AES_KEY_3_REG.
- For AES-256 encryption/decryption, the 256-bit key is stored in AES_KEY_0_REG ~ AES_KEY_7_REG.

The plaintext and ciphertext are stored in AES_TEXT_IN_*m*_REG and AES_TEXT_OUT_*m*_REG, which are two sets of four 32-bit registers.

- For AES-128/AES-256 encryption, the AES_TEXT_IN_*m*_REG registers are initialized with plaintext. Then, the AES Accelerator stores the ciphertext into AES_TEXT_OUT_*m*_REG after operation.
- For AES-128/AES-256 decryption, the AES_TEXT_IN_*m*_REG registers are initialized with ciphertext. Then, the AES Accelerator stores the plaintext into AES_TEXT_OUT_*m*_REG after operation.

6.4.2 Endianness

Text Endianness

In Typical AES working mode, the AES Accelerator uses cryptographic keys to encrypt and decrypt data in blocks of 128 bits. When filling data into AES_TEXT_IN_*m*_REG register or reading result from AES_TEXT_OUT_*m*_REG registers, users should follow the text endianness type specified in Table 6-4.

Table 6-4. Text Endianness Type for Typical AES

Plaintext/Ciphertext					
State[1]		c[2]			
		0	1	2	3
r	0	AES_TEXT_*x*_0_REG[7:0]	AES_TEXT_*x*_1_REG[7:0]	AES_TEXT_*x*_2_REG[7:0]	AES_TEXT_*x*_3_REG[7:0]
	1	AES_TEXT_*x*_0_REG[15:8]	AES_TEXT_*x*_1_REG[15:8]	AES_TEXT_*x*_2_REG[15:8]	AES_TEXT_*x*_3_REG[15:8]
	2	AES_TEXT_*x*_0_REG[23:16]	AES_TEXT_*x*_1_REG[23:16]	AES_TEXT_*x*_2_REG[23:16]	AES_TEXT_*x*_3_REG[23:16]
	3	AES_TEXT_*x*_0_REG[31:24]	AES_TEXT_*x*_1_REG[31:24]	AES_TEXT_*x*_2_REG[31:24]	AES_TEXT_*x*_3_REG[31:24]

[1] The definition of "State (including c and r)" is described in Section 3.4 The State in NIST FIPS 197.

[2] Where *x* = IN or OUT.

Key Endianness

In Typical AES working mode, when filling key into AES_KEY_*m*_REG registers, users should follow the key endianness type specified in Table 6-5 and Table 6-6.

Table 6-5. Key Endianness Type for AES-128 Encryption and Decryption

Bit[1]	w[0]	w[1]	w[2]	w[3][2]
[31:24]	AES_KEY_0_REG[7:0]	AES_KEY_1_REG[7:0]	AES_KEY_2_REG[7:0]	AES_KEY_3_REG[7:0]
[23:16]	AES_KEY_0_REG[15:8]	AES_KEY_1_REG[15:8]	AES_KEY_2_REG[15:8]	AES_KEY_3_REG[15:8]
[15:8]	AES_KEY_0_REG[23:16]	AES_KEY_1_REG[23:16]	AES_KEY_2_REG[23:16]	AES_KEY_3_REG[23:16]
[7:0]	AES_KEY_0_REG[31:24]	AES_KEY_1_REG[31:24]	AES_KEY_2_REG[31:24]	AES_KEY_3_REG[31:24]

[1] Column "Bit" specifies the bytes of each word stored in w[0] ~ w[3].

[2] w[0] ~ w[3] are "the first Nk words of the expanded key" as specified in Section 5.2 Key Expansion in NIST FIPS 197.

Table 6-6. Key Endianness Type for AES-256 Encryption and Decryption

Bit[1]	w[0]	w[1]	w[2]	w[3]	w[4]	w[5]	w[6]	w[7][2]
[31:24]	AES_KEY_0_REG[7:0]	AES_KEY_1_REG[7:0]	AES_KEY_2_REG[7:0]	AES_KEY_3_REG[7:0]	AES_KEY_4_REG[7:0]	AES_KEY_5_REG[7:0]	AES_KEY_6_REG[7:0]	AES_KEY_7_REG[7:0]
[23:16]	AES_KEY_0_REG[15:8]	AES_KEY_1_REG[15:8]	AES_KEY_2_REG[15:8]	AES_KEY_3_REG[15:8]	AES_KEY_4_REG[15:8]	AES_KEY_5_REG[15:8]	AES_KEY_6_REG[15:8]	AES_KEY_7_REG[15:8]
[15:8]	AES_KEY_0_REG[23:16]	AES_KEY_1_REG[23:16]	AES_KEY_2_REG[23:16]	AES_KEY_3_REG[23:16]	AES_KEY_4_REG[23:16]	AES_KEY_5_REG[23:16]	AES_KEY_6_REG[23:16]	AES_KEY_7_REG[23:16]
[7:0]	AES_KEY_0_REG[31:24]	AES_KEY_1_REG[31:24]	AES_KEY_2_REG[31:24]	AES_KEY_3_REG[31:24]	AES_KEY_4_REG[31:24]	AES_KEY_5_REG[31:24]	AES_KEY_6_REG[31:24]	AES_KEY_7_REG[31:24]

[1] Column "Bit" specifies the bytes of each word stored in w[0] ~ w[7].

[2] w[0] ~ w[7] are "the first Nk words of the expanded key" as specified in Chapter 5.2 Key Expansion in NIST FIPS 197.

6.4.3 Operation Process

Single Operation

1. Write 0 to the AES_DMA_ENABLE_REG register.
2. Initialize registers AES_MODE_REG, AES_KEY_*n*_REG, AES_TEXT_IN_*m*_REG.
3. Start operation by writing 1 to the AES_TRIGGER_REG register.
4. Wait till the content of the AES_STATE_REG register becomes 0, which indicates the operation is completed.
5. Read results from the AES_TEXT_OUT_*m*_REG register.

Consecutive Operations

In consecutive operations, primarily the input AES_TEXT_IN_*m*_REG and output AES_TEXT_OUT_*m*_REG registers are being written and read, while the content of AES_DMA_ENABLE_REG, AES_MODE_REG, AES_KEY_*n*_REG is kept unchanged. Therefore, the initialization can be simplified during the consecutive operation.

1. Write 0 to the AES_DMA_ENABLE_REG register before starting the first operation.
2. Initialize registers AES_MODE_REG and AES_KEY_*n*_REG before starting the first operation.
3. Update the content of AES_TEXT_IN_*m*_REG.
4. Start operation by writing 1 to the AES_TRIGGER_REG register.
5. Wait till the content of the AES_STATE_REG register becomes 0, which indicates the operation completes.
6. Read results from the AES_TEXT_OUT_*m*_REG register, and return to Step 3 to continue the next operation.

6.5 DMA-AES Working Mode

In the DMA-AES working mode, the AES accelerator supports six block cipher modes including ECB/CBC/OFB/CTR/CFB8/CFB128. Users can choose the block cipher mode by configuring the AES_BLOCK_MODE_REG register according to Table 6-7 below.

Table 6-7. Block Cipher Mode

AES_BLOCK_MODE_REG[2:0]	Block Cipher Mode
0	ECB (Electronic Codebook)
1	CBC (Cipher Block Chaining)
2	OFB (Output Feedback)
3	CTR (Counter)
4	CFB8 (8-bit Cipher Feedback)
5	CFB128 (128-bit Cipher Feedback)
6	reserved
7	reserved

Users can check the working status of the AES accelerator by inquiring the AES_STATE_REG register and comparing the return value against the Table 6-8 below.

Table 6-8. Working Status under DMA-AES Working mode

AES_STATE_REG[1:0]	Status	Description
0	IDLE	The AES accelerator is idle.
1	WORK	The AES accelerator is in the middle of an operation.
2	DONE	The AES accelerator completed operations.

When working in the DMA-AES working mode, the AES accelerator supports interrupt on the completion of computation. To enable this function, write 1 to the AES_INT_ENA_REG register. By default, the interrupt function is disabled. Also, note that the interrupt should be cleared by software after use.

6.5.1 Key, Plaintext, and Ciphertext

Block Operation

During the block operations, the AES Accelerator reads source data from DMA, and write result data to DMA after the computation.

- For encryption, DMA reads plaintext from memory, then passes it to AES as source data. After computation, AES passes ciphertext as result data back to DMA to write into memory.
- For decryption, DMA reads ciphertext from memory, then passes it to AES as source data. After computation, AES passes plaintext as result data back to DMA to write into memory.

During block operations, the lengths of the source data and result data are the same. The total computation time is reduced because the DMA data operation and AES computation can happen concurrently.

The length of source data for AES Accelerator under DMA-AES working mode must be 128 bits or the integral multiples of 128 bits. Otherwise, trailing zeros will be added to the original source data, so the length of source data equals to the nearest integral multiples of 128 bits. Please see details in Table 6-9 below.

Table 6-9. TEXT-PADDING

Function : TEXT-PADDING( )
Input : X, bit string. **Output** : Y = **TEXT-PADDING**(X), whose length is the nearest integral multiples of 128 bits.
Steps Let us assume that X is a data-stream that can be split into n parts as following: $X = X_1 \| \| X_2 \| \| \cdots \| \| X_{n-1} \| \| X_n$ Here, the lengths of $X_1, X_2, \cdots, X_{n-1}$ all equal to 128 bits, and the length of X_n is t (0<=t<=127). If $t = 0$, then **TEXT-PADDING**(X) = X; If $0 < t <= 127$, define a 128-bit block, X_n^*, and let $X_n^* = X_n \| \| 0^{128-t}$, then **TEXT-PADDING**($X$) = $X_1 \| \| X_2 \| \| \cdots \| \| X_{n-1} \| \| X_n^*$ = $X \| \| 0^{128-t}$

6.5.2 Endianness

Under the DMA-AES working mode, the transmission of source data and result data for AES Accelerator is solely controlled by DMA. Therefore, the AES Accelerator cannot control the Endianness of the source data and result

data, but does have requirement on how these data should be stored in memory and on the length of the data.

For example, let us assume DMA needs to write the following data into memory at address 0x0280.

- Data represented in hexadecimal:
  - 0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F20
- Data Length:
  - Equals to 2 blocks.

Then, this data will be stored in memory as shown in Table 6-10 below.

Table 6-10. Text Endianness for DMA-AES

Address	Byte	Address	Byte	Address	Byte	Address	Byte
0x0280	0x01	0x0281	0x02	0x0282	0x03	0x0283	0x04
0x0284	0x05	0x0285	0x06	0x0286	0x07	0x0287	0x08
0x0288	0x09	0x0289	0x0A	0x028A	0x0B	0x028B	0x0C
0x028C	0x0D	0x028D	0x0E	0x028E	0x0F	0x028F	0x10
0x0290	0x11	0x0291	0x12	0x0292	0x13	0x0293	0x14
0x0294	0x15	0x0295	0x16	0x0296	0x17	0x0297	0x18
0x0298	0x19	0x0299	0x1A	0x029A	0x1B	0x029B	0x1C
0x029C	0x1D	0x029D	0x1E	0x029E	0x1F	0x029F	0x20

6.5.3 Standard Incrementing Function

AES accelerator provides two Standard Incrementing Functions for the CTR block operation, which are INC_{32} and INC_{128} Standard Incrementing Functions. By setting the AES_INC_SEL_REG register to 0 or 1, users can choose the INC_{32} or INC_{128} functions respectively. For details on the Standard Incrementing Function, please see Chapter B.1 The Standard Incrementing Function in NIST SP 800-38A.

6.5.4 Block Number

Register AES_BLOCK_NUM_REG stores the Block Number of plaintext P or ciphertext C. The length of this register equals to length(**TEXT-PADDING**(P))/128 or length(**TEXT-PADDING**(C))/128. The AES Accelerator only uses this register when working in the DMA-AES mode.

6.5.5 Initialization Vector

AES_IV_MEM is a 16-byte memory, which is only available for AES Accelerator working in block operations. For CBC/OFB/CFB8/CFB128 operations, the AES_IV_MEM memory stores the Initialization Vector (IV). For the CTR operation, the AES_IV_MEM memory stores the Initial Counter Block (ICB).

Both IV and ICB are 128-bit strings, which can be divided into Byte0, Byte1, Byte2 ⋯ Byte15 (from left to right). AES_IV_MEM stores data following the Endianness pattern presented in Table 6-10, i.e. the most significant (i.e., left-most) byte Byte0 is stored at the lowest address while the least significant (i.e., right-most) byte Byte15 at the highest address.

For more details on IV and ICB, please refer to NIST SP 800-38A.

6.5.6 Block Operation Process

1. Select one of DMA channels to connect with AES, configure the DMA chained list, and then start DMA.
2. Initialize the AES accelerator-related registers:
   - Write 1 to the AES_DMA_ENABLE_REG register.
   - Configure the AES_INT_ENA_REG register to enable or disable the interrupt function.
   - Initialize registers AES_MODE_REG and AES_KEY_*n*_REG.
   - Select block cipher mode by configuring the AES_BLOCK_MODE_REG register. For details, see Table 6-7.
   - Initialize the AES_BLOCK_NUM_REG register. For details, see Section 6.5.4.
   - Initialize the AES_INC_SEL_REG register (only needed when AES Accelerator is working under CTR block operation).
   - Initialize the AES_IV_MEM memory (This is always needed except for ECB block operation).
3. Start operation by writing 1 to the AES_TRIGGER_REG register.
4. Wait for the completion of computation, which happens when the content of AES_STATE_REG becomes 2 or the AES interrupt occurs.
5. Check if DMA completes data transmission from AES to memory. At this time, DMA had already written the result data in memory, which can be accessed directly.
6. Clear interrupt by writing 1 to the AES_INT_CLR_REG register, if any AES interrupt occurred during the computation.
7. Release the AES Accelerator by writing 0 to the AES_DMA_EXIT_REG register. After this, the content of the AES_STATE_REG register becomes 0. Note that, you can release DMA earlier, but only after Step 4 is completed.

6.6 Memory Summary

The addresses in this section are relative to the AES accelerator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Name	Description	Size (byte)	Starting Address	Ending Address	Access
AES_IV_MEM	Memory IV	16 bytes	0x0050	0x005F	R/W

6.7 Register Summary

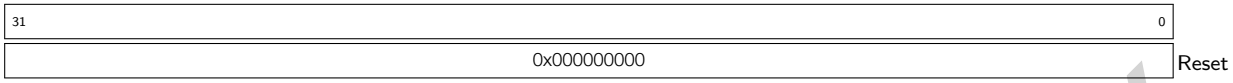
The addresses in this section are relative to the AES accelerator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Name	Description	Address	Access
Key Registers			
AES_KEY_0_REG	AES key data register 0	0x0000	R/W
AES_KEY_1_REG	AES key data register 1	0x0004	R/W
AES_KEY_2_REG	AES key data register 2	0x0008	R/W
AES_KEY_3_REG	AES key data register 3	0x000C	R/W
AES_KEY_4_REG	AES key data register 4	0x0010	R/W
AES_KEY_5_REG	AES key data register 5	0x0014	R/W
AES_KEY_6_REG	AES key data register 6	0x0018	R/W
AES_KEY_7_REG	AES key data register 7	0x001C	R/W
TEXT_IN Registers			
AES_TEXT_IN_0_REG	Source text data register 0	0x0020	R/W
AES_TEXT_IN_1_REG	Source text data register 1	0x0024	R/W
AES_TEXT_IN_2_REG	Source text data register 2	0x0028	R/W
AES_TEXT_IN_3_REG	Source text data register 3	0x002C	R/W
TEXT_OUT Registers			
AES_TEXT_OUT_0_REG	Result text data register 0	0x0030	RO
AES_TEXT_OUT_1_REG	Result text data register 1	0x0034	RO
AES_TEXT_OUT_2_REG	Result text data register 2	0x0038	RO
AES_TEXT_OUT_3_REG	Result text data register 3	0x003C	RO
Configuration Registers			
AES_MODE_REG	Defines key length and encryption / decryption	0x0040	R/W
AES_DMA_ENABLE_REG	Selects the working mode of the AES accelerator	0x0090	R/W
AES_BLOCK_MODE_REG	Defines the block cipher mode	0x0094	R/W
AES_BLOCK_NUM_REG	Block number configuration register	0x0098	R/W
AES_INC_SEL_REG	Standard incrementing function register	0x009C	R/W
Controlling / Status Registers			
AES_TRIGGER_REG	Operation start controlling register	0x0048	WO
AES_STATE_REG	Operation status register	0x004C	RO
AES_DMA_EXIT_REG	Operation exit controlling register	0x00B8	WO
Interruption Registers			
AES_INT_CLR_REG	DMA-AES interrupt clear register	0x00AC	WO
AES_INT_ENA_REG	DMA-AES interrupt enable register	0x00B0	R/W

6.8 Registers

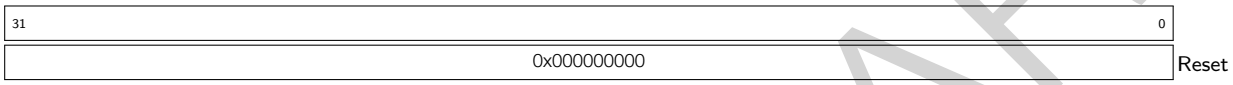
The addresses in this section are relative to the AES accelerator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Register 6.1. AES_KEY_*n*_REG (*n*: 0-7) (0x0000+4n*)**



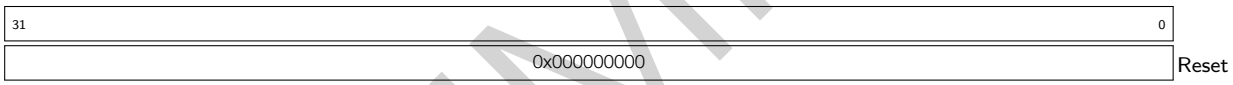
AES_KEY_*n*_REG (*n*: 0-7) Stores AES key data. (R/W)

Register 6.2. AES_TEXT_IN_*m*_REG (*m*: 0-3) (0x0020+4m*)**



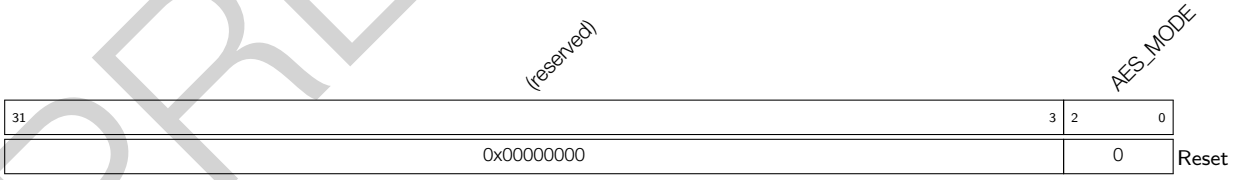
AES_TEXT_IN_*m*_REG (*m*: 0-3) Stores the source text data when the AES Accelerator operates in the Typical AES working mode. (R/W)

Register 6.3. AES_TEXT_OUT_*m*_REG (*m*: 0-3) (0x0030+4m*)**



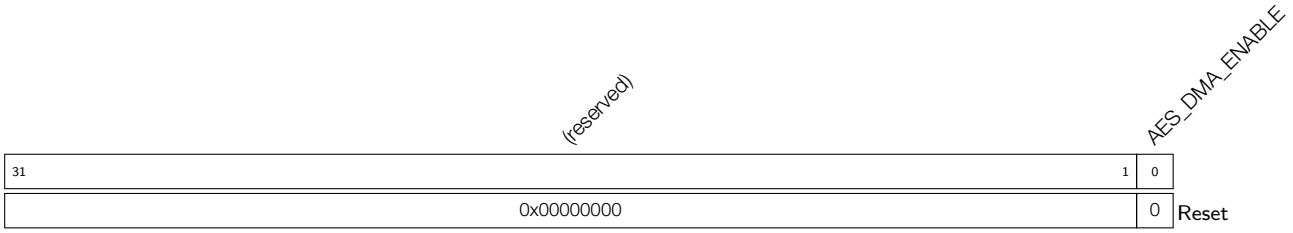
AES_TEXT_OUT_*m*_REG (*m*: 0-3) Stores the result text data when the AES Accelerator operates in the Typical AES working mode. (RO)

Register 6.4. AES_MODE_REG (0x0040)



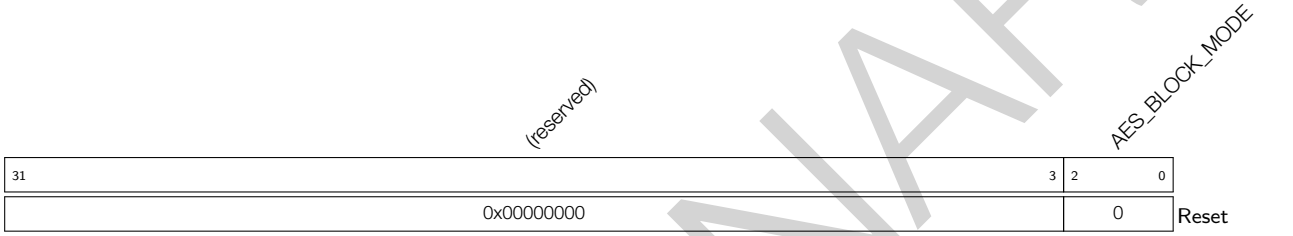
AES_MODE Defines the key length and encryption / decryption of the AES Accelerator. For details, see Table 6-2. (R/W)

Register 6.5. AES_DMA_ENABLE_REG (0x0090)



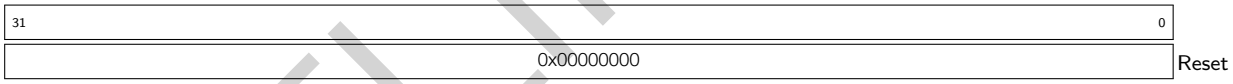
AES_DMA_ENABLE Defines the working mode of the AES Accelerator. 0: Typical AES, 1: DMA-AES. For details, see Table 6-1. (R/W)

Register 6.6. AES_BLOCK_MODE_REG (0x0094)



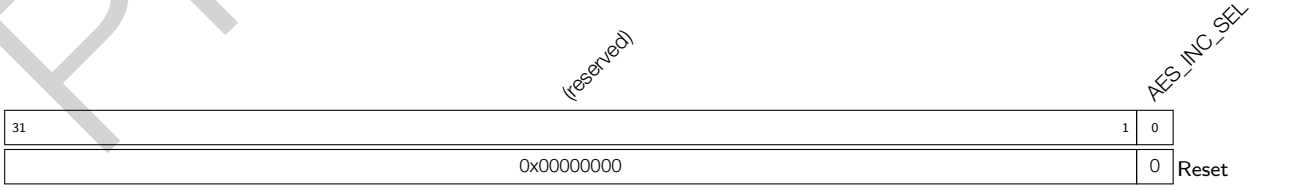
AES_BLOCK_MODE Defines the block cipher mode of the AES Accelerator operating under the DMA-AES working mode. For details, see Table 6-7. (R/W)

Register 6.7. AES_BLOCK_NUM_REG (0x0098)



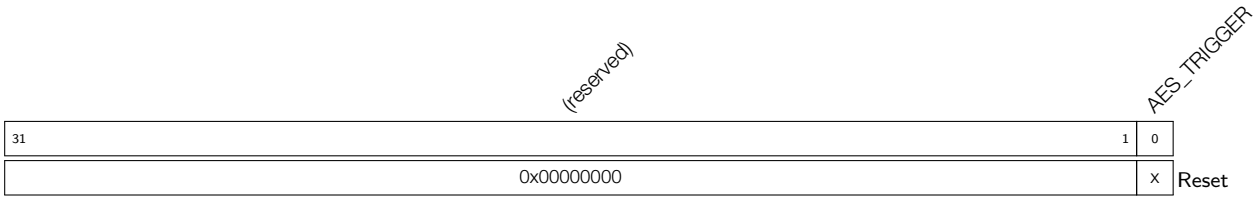
AES_BLOCK_NUM Stores the Block Number of plaintext or ciphertext when the AES Accelerator operates under the DMA-AES working mode. For details, see Section 6.5.4. (R/W)

Register 6.8. AES_INC_SEL_REG (0x009C)



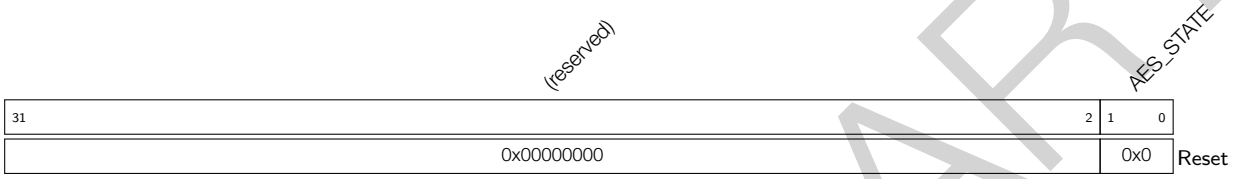
AES_INC_SEL Defines the Standard Incrementing Function for CTR block operation. Set this bit to 0 or 1 to choose INC_{32} or INC_{128}. (R/W)

Register 6.9. AES_TRIGGER_REG (0x0048)



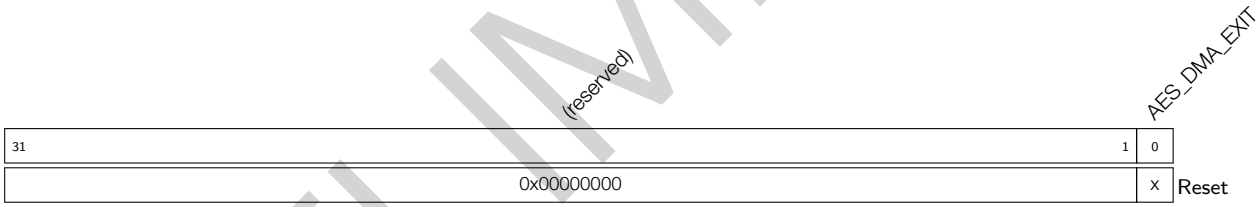
AES_TRIGGER Set this bit to 1 to start AES operation. (WO)

Register 6.10. AES_STATE_REG (0x004C)



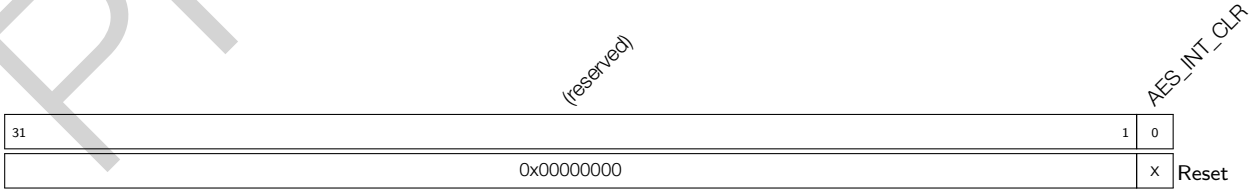
AES_STATE Stores the working status of the AES Accelerator. For details, see Table 6-3 for Typical AES working mode and Table 6-8 for DMA AES working mode. (RO)

Register 6.11. AES_DMA_EXIT_REG (0x00B8)



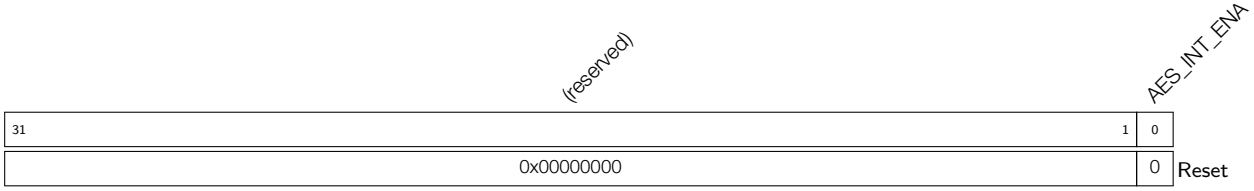
AES_DMA_EXIT Set this bit to 1 to exit AES operation. This register is only effective for DMA-AES operation. (WO)

Register 6.12. AES_INT_CLR_REG (0x00AC)



AES_INT_CLR Set this bit to 1 to clear AES interrupt. (WO)

Register 6.13. AES_INT_ENA_REG (0x00B0)



AES_INT_ENA Set this bit to 1 to enable AES interrupt and 0 to disable interrupt. (R/W)

7 RSA Accelerator

7.1 Introduction

The RSA Accelerator provides hardware support for high precision computation used in various RSA asymmetric cipher algorithms by significantly reducing their software complexity. Compared with RSA algorithms implemented solely in software, this hardware accelerator can speed up RSA algorithms significantly. Besides, the RSA Accelerator also supports operands of different lengths, which provides more flexibility during the computation.

7.2 Features

The following functionality is supported:

- Large-number modular exponentiation with two optional acceleration options
- Large-number modular multiplication
- Large-number multiplication
- Operands of different lengths
- Interrupt on completion of computation

7.3 Functional Description

The RSA Accelerator is activated by setting the SYSTEM_CRYPTO_RSA_CLK_EN bit in the SYSTEM_PERIP_CLK_EN1_REG register and clearing the SYSTEM_RSA_MEM_PD bit in the SYSTEM_RSA_PD_CTRL_REG register. This releases the RSA Accelerator from reset.

The RSA Accelerator is only available after the RSA-related memories are initialized. The content of the RSA_CLEAN_REG register is 0 during initialization and will become 1 after the initialization is done. Therefore, it is advised to wait until RSA_CLEAN_REG becomes 1 before using the RSA Accelerator.

The RSA_INTERRUPT_ENA_REG register is used to control the interrupt triggered on completion of computation. Write 1 or 0 to this register to enable or disable interrupt. By default, the interrupt function of the RSA Accelerator is enabled.

7.3.1 Large Number Modular Exponentiation

Large-number modular exponentiation performs $Z = X^Y \bmod M$. The computation is based on Montgomery multiplication. Therefore, aside from the X, Y, and M arguments, two additional ones are needed — $\overline{r}$ and M', which need to be calculated in advance by software.

RSA Accelerator supports operands of length $N = 32 \times x$, where $x \in \{1, 2, 3, \ldots, 96\}$. The bit lengths of arguments Z, X, Y, M, and $\overline{r}$ can be arbitrary N, but all numbers in a calculation must be of the same length. The bit length of M' must be 32.

To represent the numbers used as operands, let us define a base-b positional notation, as follows:

$$b = 2^{32}$$

Using this notation, each number is represented by a sequence of base-b digits:

$$n = \frac{N}{32}$$
$$Z = (Z_{n-1}Z_{n-2}\cdots Z_0)_b$$
$$X = (X_{n-1}X_{n-2}\cdots X_0)_b$$
$$Y = (Y_{n-1}Y_{n-2}\cdots Y_0)_b$$
$$M = (M_{n-1}M_{n-2}\cdots M_0)_b$$
$$\overline{r} = (\overline{r}_{n-1}\overline{r}_{n-2}\cdots \overline{r}_0)_b$$

Each of the n values in $Z_{n-1}\cdots Z_0$, $X_{n-1}\cdots X_0$, $Y_{n-1}\cdots Y_0$, $M_{n-1}\cdots M_0$, $\overline{r}_{n-1}\cdots \overline{r}_0$ represents one base-b digit (a 32-bit word).

Z_{n-1}, X_{n-1}, Y_{n-1}, M_{n-1} and $\overline{r}_{n-1}$ are the most significant bits of Z, X, Y, M, while Z_0, X_0, Y_0, M_0 and $\overline{r}_0$ are the least significant bits.

If we define $R = b^n$, the additional arguments can be calculated as $\overline{r} = R^2 \bmod M$.

The following equation in the form compatible with the extended binary GCD algorithm can be written as

$$M^{-1} \times M + 1 = R \times R^{-1}$$
$$M' = M^{-1} \bmod b$$

Large-number modular exponentiation can be implemented as follows:

1. Write 1 or 0 to the RSA_INTERRUPT_ENA_REG register to enable or disable the interrupt function.
2. Configure relevant registers:
   (a) Write $(\frac{N}{32} - 1)$ to the RSA_MODE_REG register.
   (b) Write M' to the RSA_M_PRIME_REG register.
   (c) Configure registers related to the acceleration options, which are described later in Section 7.3.4.
3. Write X_i, Y_i, M_i and $\overline{r}_i$ for $i \in \{0, 1, \ldots, n-1\}$ to memory blocks RSA_X_MEM, RSA_Y_MEM, RSA_M_MEM and RSA_Z_MEM. The capacity of each memory block is 96 words. Each word of each memory block can store one base-b digit. The memory blocks use the little endian format for storage, i.e. the least significant digit of each number is in the lowest address.

   Users need to write data to each memory block only according to the length of the number; data beyond this length are ignored.
4. Write 1 to the RSA_MODEXP_START_REG register to start computation.
5. Wait for the completion of computation, which happens when the content of RSA_IDLE_REG becomes 1 or the RSA interrupt occurs.

6. Read the result Z_i for $i \in \{0, 1, \ldots, n-1\}$ from RSA_Z_MEM.
7. Write 1 to RSA_CLEAR_INTERRUPT_REG to clear the interrupt, if you have enabled the interrupt function.

After the computation, the RSA_MODE_REG register, memory blocks RSA_Y_MEM and RSA_M_MEM, as well as the RSA_M_PRIME_REG remain unchanged. However, X_i in RSA_X_MEM and $\overline{r}_i$ in RSA_Z_MEM computation are overwritten, and only these overwritten memory blocks need to be re-initialized before starting another computation.

7.3.2 Large Number Modular Multiplication

Large-number modular multiplication performs $Z = X \times Y \bmod M$. This computation is based on Montgomery multiplication. Therefore, similar to the large number modular exponentiation, two additional arguments are needed – $\overline{r}$ and M', which need to be calculated in advance by software.

The RSA Accelerator supports large-number modular multiplication with operands of 96 different lengths.

The computation can be executed as follows:

1. Write 1 or 0 to the RSA_INTERRUPT_ENA_REG register to enable or disable the interrupt function.
2. Configure relevant registers:
   (a) Write $(\frac{N}{32} - 1)$ to the RSA_MODE_REG register.
   (b) Write M' to the RSA_M_PRIME_REG register.
3. Write X_i, Y_i, M_i, and $\overline{r}_i$ for $i \in \{0, 1, \ldots, n-1\}$ to memory blocks RSA_X_MEM, RSA_Y_MEM, RSA_M_MEM and RSA_Z_MEM. The capacity of each memory block is 96 words. Each word of each memory block can store one base-b digit. The memory blocks use the little endian format for storage, i.e. the least significant digit of each number is in the lowest address.

   Users need to write data to each memory block only according to the length of the number; data beyond this length are ignored.
4. Write 1 to the RSA_MODMULT_START_REG register.
5. Wait for the completion of computation, which happens when the content of RSA_IDLE_REG becomes 1 or the RSA interrupt occurs.
6. Read the result Z_i for $i \in \{0, 1, \ldots, n-1\}$ from RSA_Z_MEM.
7. Write 1 to RSA_CLEAR_INTERRUPT_REG to clear the interrupt, if you have enabled the interrupt function.

After the computation, the length of operands in RSA_MODE_REG, the X_i in memory RSA_X_MEM, the Y_i in memory RSA_Y_MEM, the M_i in memory RSA_M_MEM, and the M' in memory RSA_M_PRIME_REG remain unchanged. However, the $\overline{r}_i$ in memory RSA_Z_MEM has already been overwritten, and only this overwritten memory block needs to be re-initialized before starting another computation.

7.3.3 Large Number Multiplication

Large-number multiplication performs $Z = X \times Y$. The length of result Z is twice that of operand X and operand Y. Therefore, the RSA Accelerator only supports Large Number Multiplication with operand length $N = 32 \times x$, where $x \in \{1, 2, 3, \ldots, 48\}$. The length $\hat{N}$ of result Z is $2 \times N$.

The computation can be executed as follows:

1. Write 1 or 0 to the RSA_INTERRUPT_ENA_REG register to enable or disable the interrupt function.
2. Write $(\frac{\hat{N}}{32} - 1)$, i.e. $(\frac{N}{16} - 1)$ to the RSA_MODE_REG register.
3. Write X_i and Y_i for $\in \{0, 1, \ldots, n-1\}$ to memory blocks RSA_X_MEM and RSA_Z_MEM. Each word of each memory block can store one base-b digit. The memory blocks use the little endian format for storage, i.e. the least significant digit of each number is in the lowest address. n is $\frac{N}{32}$.

   Write X_i for $i \in \{0, 1, \ldots, n-1\}$ to the address of the i words of the RSA_X_MEM memory block. Note that Y_i for $i \in \{0, 1, \ldots, n-1\}$ will not be written to the address of the i words of the RSA_Z_MEM register, but the address of the $n+i$ words, i.e. the base address of the RSA_Z_MEM memory plus the address offset $4 \times (n+i)$.

   Users need to write data to each memory block only according to the length of the number; data beyond this length are ignored.
4. Write 1 to the RSA_MULT_START_REG register.
5. Wait for the completion of computation, which happens when the content of RSA_IDLE_REG becomes 1 or the RSA interrupt occurs.
6. Read the result Z_i for $i \in \{0, 1, \ldots, \hat{n}-1\}$ from the RSA_Z_MEM register. $\hat{n}$ is $2 \times n$.
7. Write 1 to RSA_CLEAR_INTERRUPT_REG to clear the interrupt, if you have enabled the interrupt function.

After the computation, the length of operands in RSA_MODE_REG and the X_i in memory RSA_X_MEM remain unchanged. However, the Y_i in memory RSA_Z_MEM has already been overwritten, and only this overwritten memory block needs to be re-initialized before starting another computation.

7.3.4 Options for Acceleration

The ESP32-C3 RSA accelerator also provides SEARCH and CONSTANT_TIME options that can be configured to accelerate the large-number modular exponentiation. By default, both options are configured for no acceleration. Users can choose to use one or two of these options to accelerate the computation.

To be more specific, when neither of these two options are configured for acceleration, the time required to calculate $Z = X^Y \bmod M$ is solely determined by the lengths of operands. When either or both of these two options are configured for acceleration, the time required is also correlated with the 0/1 distribution of Y.

To better illustrate how these two options work, first assume Y is represented in binaries as

$$Y = (\widetilde{Y}_{N-1}\widetilde{Y}_{N-2}\cdots\widetilde{Y}_{t+1}\widetilde{Y}_t\widetilde{Y}_{t-1}\cdots\widetilde{Y}_0)_2$$

where,

- N is the length of Y,
- $\widetilde{Y}_t$ is 1,
- $\widetilde{Y}_{N-1}$, $\widetilde{Y}_{N-2}$, ..., $\widetilde{Y}_{t+1}$ are all equal to 0,
- and $\widetilde{Y}_{t-1}$, $\widetilde{Y}_{t-2}$, ..., $\widetilde{Y}_0$ are either 0 or 1 but exactly m bits should be equal to 0 and t-m bits 1, i.e. the Hamming weight of $\widetilde{Y}_{t-1}\widetilde{Y}_{t-2}, \cdots, \widetilde{Y}_0$ is $t - m$.

When either of these two options is configured for acceleration:

- SEARCH Option (Configuring RSA_SEARCH_ENABLE to 1 for acceleration)
  - The accelerator ignores the bit positions of $\widetilde{Y}_i$, where $i > \alpha$. Search position α is set by configuring the RSA_SEARCH_POS_REG register. The maximum value of α is N-1, which leads to the same result when this option is not used for acceleration. The best acceleration performance can be achieved by setting α to t, in which case, all the $\widetilde{Y}_{N-1}$, $\widetilde{Y}_{N-2}$, ..., $\widetilde{Y}_{t+1}$ of 0s are ignored during the calculation. Note that if you set α to be less than t, then the result of the modular exponentiation $Z = X^Y \bmod M$ will be incorrect.
- CONSTANT_TIME Option (Configuring RSA_CONSTANT_TIME_REG to 0 for acceleration)
  - The accelerator speeds up the calculation by simplifying the calculation concerning the 0 bits of Y. Therefore, the higher the proportion of bits 0 against bits 1, the better the acceleration performance is.

We provide an example to demonstrate the performance of the RSA Accelerator under different combinations of SEARCH and CONSTANT_TIME configuration. Here we perform $Z = X^Y \bmod M$ with N = 3072 and Y = 65537. Table 7-1 below demonstrates the time costs under different combinations of SEARCH and CONSTANT_TIME configuration. Here, we should also mention that, α is set to 16 when the SEARCH option is enabled.

Table 7-1. Acceleration Performance

SEARCH Option	CONSTANT_TIME Option	Time Cost
No acceleration	No acceleration	376.405 ms
Accelerated	No acceleration	2.260 ms
No acceleration	Acceleration	1.203 ms
Acceleration	Acceleration	1.165 ms

It's obvious that:

- The time cost is the biggest when none of these two options is configured for acceleration.
- The time cost is the smallest when both of these two options are configured for acceleration.
- The time cost can be dramatically reduced when either or both option(s) are configured for acceleration.

7.4 Memory Summary

The addresses in this section are relative to the RSA accelerator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Table 7-2. RSA Accelerator Memory Blocks

Name	Description	Size (byte)	Starting Address	Ending Address	Access
RSA_M_MEM	Memory M	384	0x0000	0x017F	R/W
RSA_Z_MEM	Memory Z	384	0x0200	0x037F	R/W
RSA_Y_MEM	Memory Y	384	0x0400	0x057F	R/W
RSA_X_MEM	Memory X	384	0x0600	0x077F	R/W

7.5 Register Summary

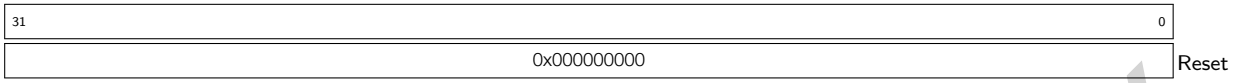
The addresses in this section are relative to the RSA accelerator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Name	Description	Address	Access
Configuration Registers			
RSA_M_PRIME_REG	Register to store M'	0x0800	R/W
RSA_MODE_REG	RSA length mode	0x0804	R/W
RSA_CONSTANT_TIME_REG	The constant_time option	0x0820	R/W
RSA_SEARCH_ENABLE_REG	The search option	0x0824	R/W
RSA_SEARCH_POS_REG	The search position	0x0828	R/W
Status/Control Registers			
RSA_CLEAN_REG	RSA clean register	0x0808	RO
RSA_MODEXP_START_REG	Modular exponentiation starting bit	0x080C	WO
RSA_MODMULT_START_REG	Modular multiplication starting bit	0x0810	WO
RSA_MULT_START_REG	Normal multiplication starting bit	0x0814	WO
RSA_IDLE_REG	RSA idle register	0x0818	RO
Interrupt Registers			
RSA_CLEAR_INTERRUPT_REG	RSA clear interrupt register	0x081C	WO
RSA_INTERRUPT_ENA_REG	RSA interrupt enable register	0x082C	R/W
Version Register			
RSA_DATE_REG	Version control register	0x0830	R/W

7.6 Registers

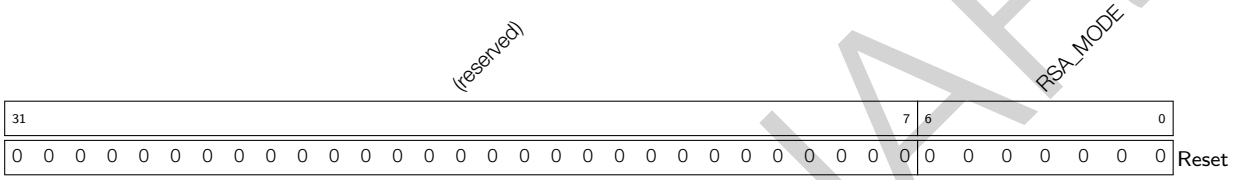
The addresses in this section are relative to the RSA accelerator base address provided in Table 3-4 in Chapter 3 *System and Memory*.

Register 7.1. RSA_M_PRIME_REG (0x0800)



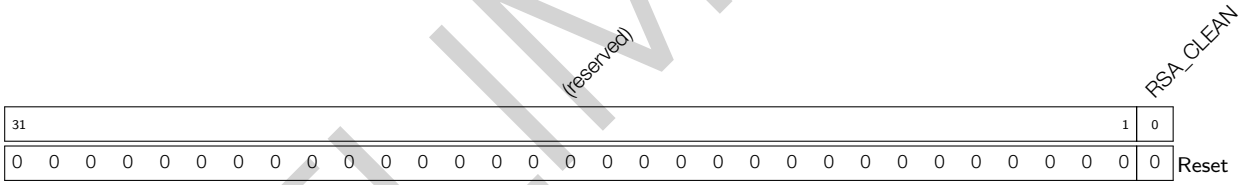
RSA_M_PRIME_REG Stores M'.(R/W)

Register 7.2. RSA_MODE_REG (0x0804)



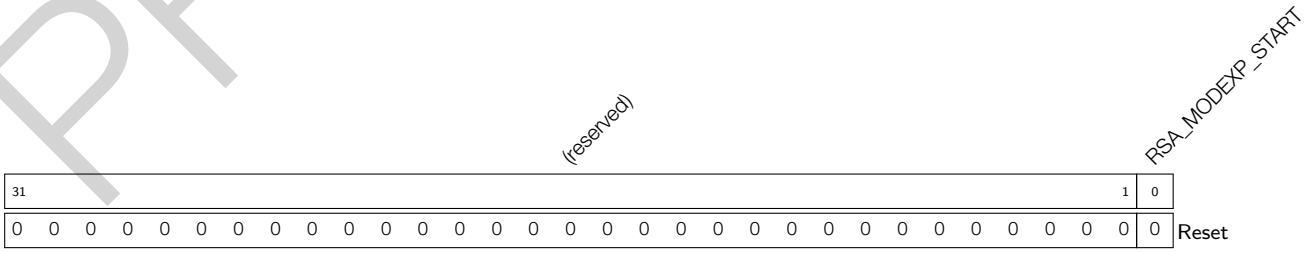
RSA_MODE Stores the mode of modular exponentiation. (R/W)

Register 7.3. RSA_CLEAN_REG (0x0808)



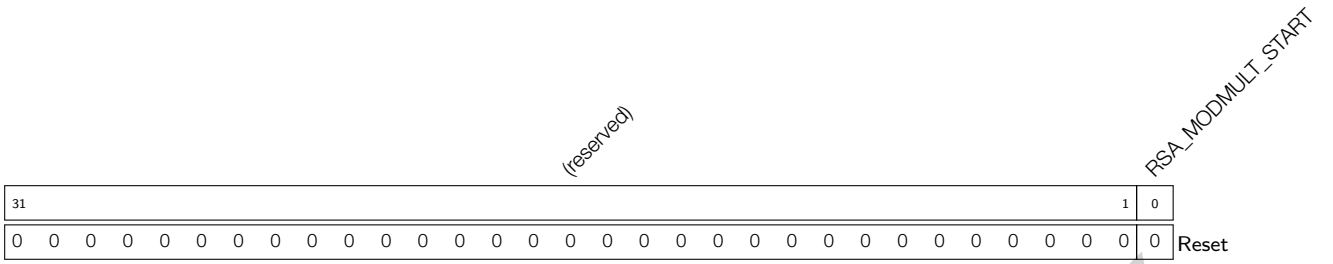
RSA_CLEAN The content of this bit is 1 when memories complete initialization. (RO)

Register 7.4. RSA_MODEXP_START_REG (0x080C)



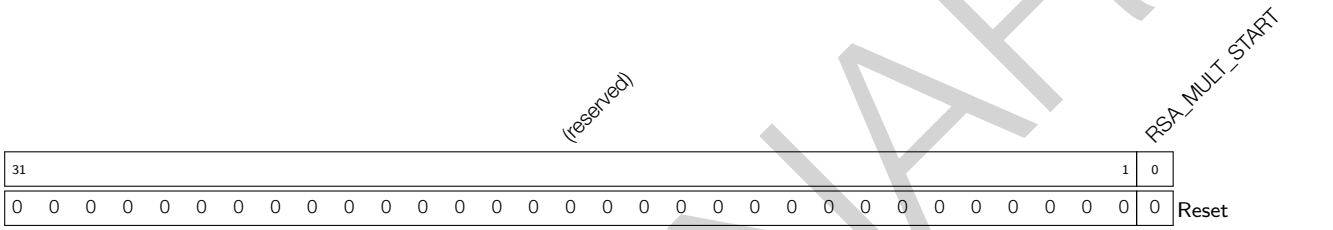
RSA_MODEXP_START Set this bit to 1 to start the modular exponentiation. (WO)

Register 7.5. RSA_MODMULT_START_REG (0x0810)



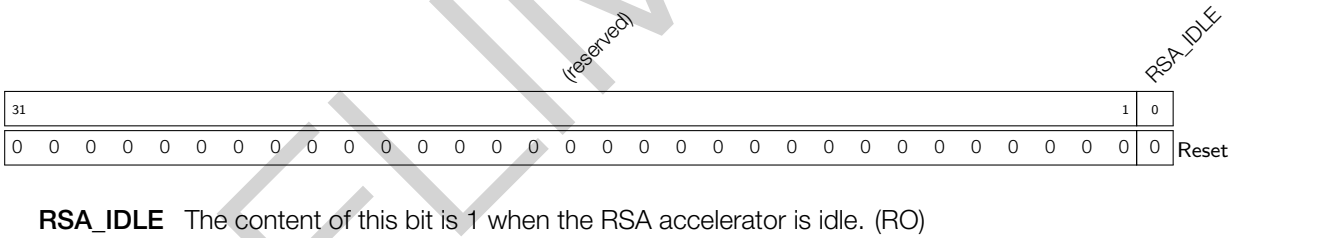
RSA_MODMULT_START Set this bit to 1 to start the modular multiplication. (WO)

Register 7.6. RSA_MULT_START_REG (0x0814)



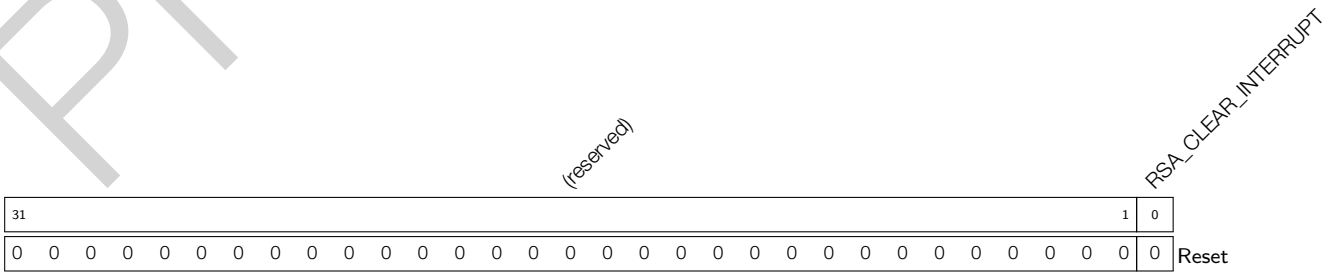
RSA_MULT_START Set this bit to 1 to start the multiplication. (WO)

Register 7.7. RSA_IDLE_REG (0x0818)



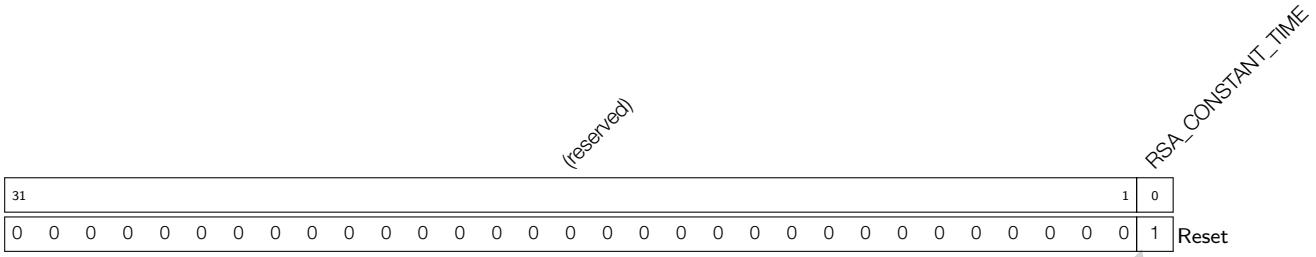
RSA_IDLE The content of this bit is 1 when the RSA accelerator is idle. (RO)

Register 7.8. RSA_CLEAR_INTERRUPT_REG (0x081C)



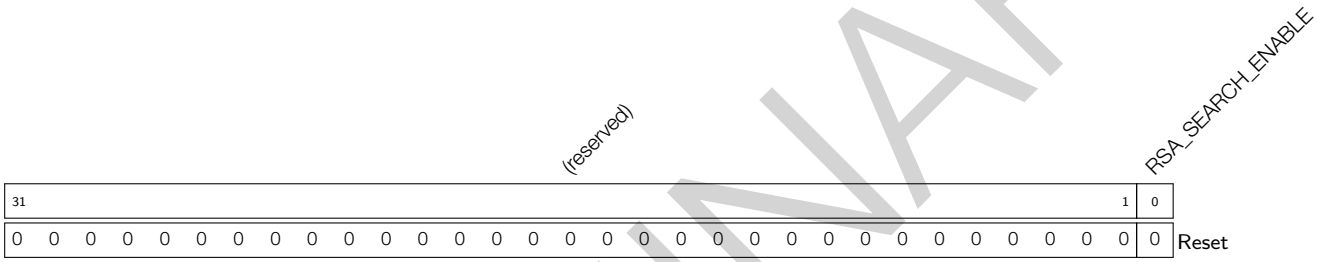
RSA_CLEAR_INTERRUPT Set this bit to 1 to clear the RSA interrupts. (WO)

Register 7.9. RSA_CONSTANT_TIME_REG (0x0820)



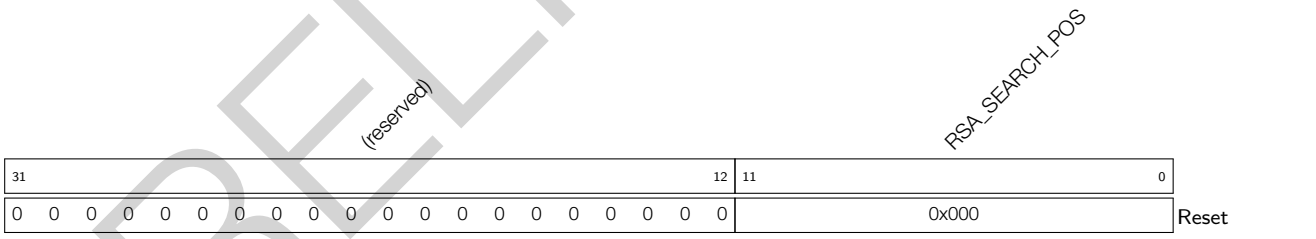
RSA_CONSTANT_TIME_REG Controls the constant_time option. 0: acceleration. 1: no acceleration (by default). (R/W)

Register 7.10. RSA_SEARCH_ENABLE_REG (0x0824)



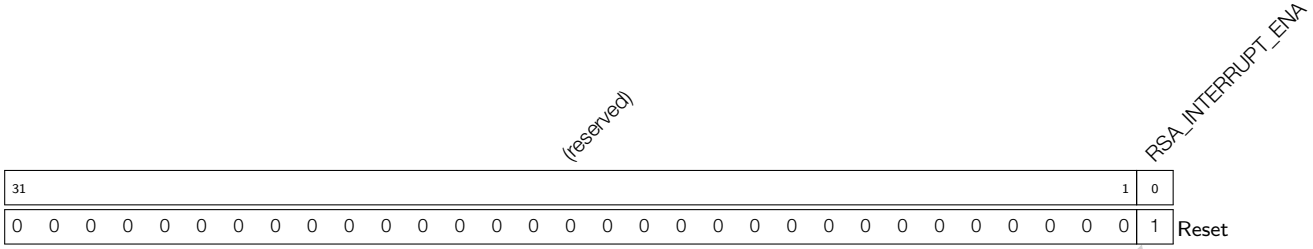
RSA_SEARCH_ENABLE Controls the search option. 0: no acceleration (by default). 1: acceleration. (R/W)

Register 7.11. RSA_SEARCH_POS_REG (0x0828)



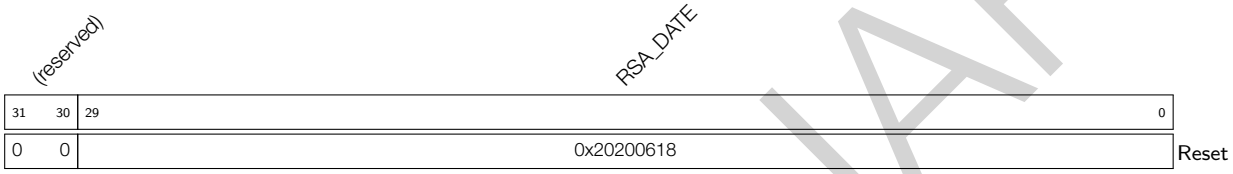
RSA_SEARCH_POS Is used to configure the starting address when the acceleration option of search is used. (R/W)

Register 7.12. RSA_INTERRUPT_ENA_REG (0x082C)



RSA_INTERRUPT_ENA Set this bit to 1 to enable the RSA interrupt. This option is enabled by default. (R/W)

Register 7.13. RSA_DATE_REG (0x0830)



RSA_DATE Version control register. (R/W)

8 Chip Boot Control

8.1 Overview

ESP32-C3 has four strapping pins:

- GPIO2
- GPIO8
- GPIO9
- GPIO10

These strapping pins are used to control the following functions during chip power-on or hardware reset:

- control chip boot mode
- enable or disable ROM code printing to UART
- control the source of JTAG signals

During system reset triggered by power-on, brown-out or by analog super watchdog (see Chapter 1 *Reset and Clock*), hardware samples and stores the voltage level of strapping pins as strapping bit of "0" or "1" in latches, and holds these bits until the chip is powered down or shut down. Software can read the latch status (strapping value) of GPIO2, GPIO8, and GPIO9 from the register GPIO_STRAPPING.

By default, GPIO9 is connected to the chip's internal pull-up resistor. If GPIO9 is not connected or connected to an external high-impedance circuit, the internal weak pull-up determines the default input level of this strapping pin (see Table 8-1).

Table 8-1. Default Configuration of Strapping Pins

Strapping Pin	Defualt Configuration
GPIO2	N/A
GPIO8	N/A
GPIO9	Pull-up
GPIO10	N/A

To change the strapping bit values, users can apply external pull-down/pull-up resistors, or use host MCU GPIOs to control the voltage level of these pins when powering on ESP32-C3. After the reset is released, the strapping pins work as normal-function pins.

8.2 Boot Mode Control

GPIO2, GPIO8 and GPIO9 control the boot mode after the reset is released.

Table 8-2. Boot Mode

Pin	SPI Boot	Download Boot
GPIO2	1	1
GPIO8	x	1
GPIO9	1	0

Table 8-2 shows the strapping pin values of GPIO2, GPIO8 and GPIO9, and the associated boot modes. "x" means that this value is ignored.

In SPI Boot mode, the CPU boots the system by reading the program stored in SPI flash. SPI Boot mode can be further classified as follows:

- Normal Flash Boot: supports Security Boot and programs run in RAM.
- Direct Boot: does not support Security Boot and programs run directly in flash. To enable this mode, make sure that the first two words of the bin file downloading to flash (address: 0x42000000) are 0xaebd041d.

In Download Boot mode, users can download code to SRAM or flash using UART0 or USB interface. It is also possible to load a program into SRAM and execute it in this mode.

The following eFuses control boot mode behaviors:

- EFUSE_DIS_FORCE_DOWNLOAD

  If this eFuse is 0 (default), software can force switch the chip from SPI Boot mode to Download Boot mode by setting register RTC_CNTL_FORCE_DOWNLOAD_BOOT and triggering a CPU reset. If this eFuse is 1, RTC_CNTL_FORCE_DOWNLOAD_BOOT is disabled.

- EFUSE_DIS_DOWNLOAD_MODE

  If this eFuse is 1, Download Boot mode is disabled.

- EFUSE_ENABLE_SECURITY_DOWNLOAD

  If this eFuse is 1, Download Boot mode only allows reading, writing, and erasing plaintext flash and does not support any SRAM or register operations. Ignore this eFuse if Download Boot mode is disabled.

8.3 ROM Code Printing Control

GPIO8 controls ROM code printing of information during the early boot process. This GPIO is used together with EFUSE_UART_PRINT_CONTROL.

Table 8-3. ROM Code Printing Control

eFuse[1]	GPIO8	ROM Code Printing
0	-	ROM code is always printed to UART during boot. GPIO8 is not used.
1	0	Print is enabled during boot
	1	Print is disabled during boot
2	0	Print is disabled during boot
	1	Print is enabled during boot
3	-	Print is always disabled during boot. GPIO8 is not used.

[1] eFuse: EFUSE_UART_PRINT_CONTROL

ROM code will print to pin U0TXD (default) or to USB Serial/JTAG Controller during power-on, depending on the eFuse bit EFUSE_USB_PRINT_CHANNEL (0: USB; 1: UART). Note that if this eFuse bit is set to 0, i.e., USB is selected, but USB Serial/JTAG Controller is disabled, then ROM code will not be printed.

8.4 JTAG Signals Source Control

GPIO10 controls the source of JTAG signals during the early boot process. This GPIO is used together with EFUSE_DIS_PAD_JTAG, EFUSE_DIS_USB_JTAG, and EFUSE_JTAG_SEL_ENABLE, see Table 8-4.

Table 8-4. JTAG Signals Source Control

eFuse 1[a]	eFuse 2[b]	eFuse 3[c]	GPIO 10	Signals Source
0	0	0	-	JTAG signals come from USB Serial/JTAG Controller. GPIO10 is not used.
		1	0	JTAG signals come from corresponding pins.
			1	JTAG signals come from USB Serial/JTAG Controller.
0	1	-	-	JTAG signals come from corresponding pins. EFUSE_JTAG_SEL_ENABLE and GPIO10 are not used.
1	0	-	-	JTAG signals come from USB Serial/JTAG Controller. EFUSE_JTAG_SEL_ENABLE and GPIO10 are not used.
1	1	-	-	JTAG is disabled. EFUSE_JTAG_SEL_ENABLE and GPIO10 are not used.

[a] eFuse 1: EFUSE_DIS_PAD_JTAG
[b] eFuse 2: EFUSE_DIS_USB_JTAG
[c] eFuse 3: EFUSE_JTAG_SEL_ENABLE

8.5 USB Serial/JTAG Controller

USB Serial/JTAG Controller can force the chip into Download Boot mode from SPI Boot mode, as well as force the chip into SPI Boot mode from Download Boot mode.

9 ESP-RISC-V CPU

9.1 Overview

ESP-RISC-V CPU is a 32-bit core based upon RISC-V ISA comprising base integer (I), multiplication/division (M) and compressed (C) standard extensions. The core has 4-stage, in-order, scalar pipeline optimized for area, power and performance. CPU core complex has an interrupt-controller (INTC), debug module (DM) and system bus (SYS BUS) interfaces for memory and peripheral access.

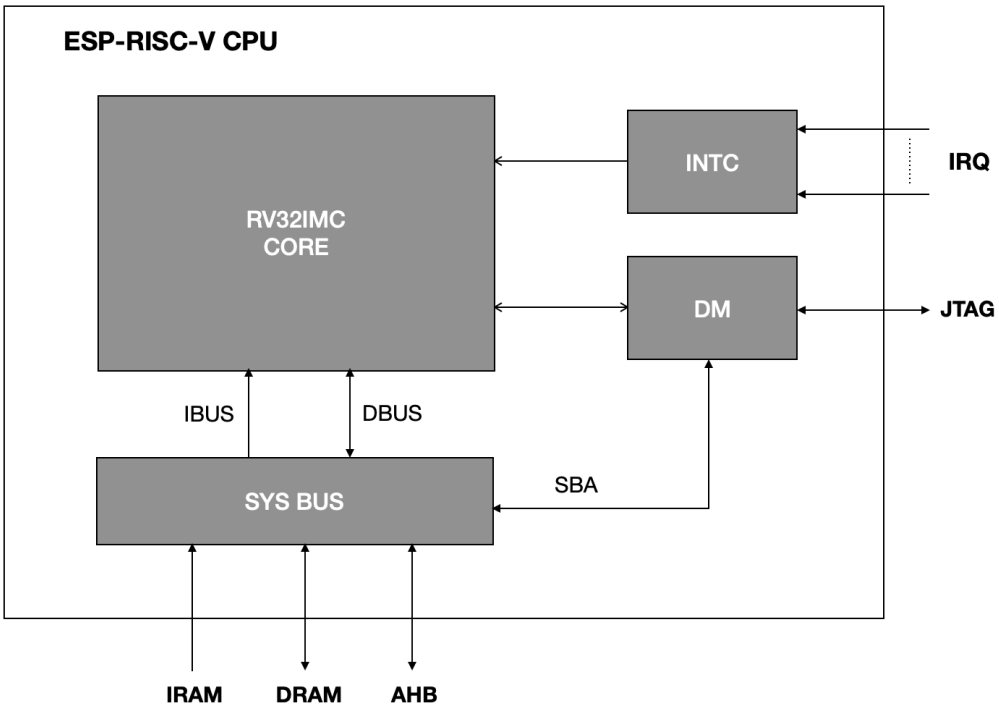


Figure 9-1. CPU Block Diagram

9.2 Features

- Operating clock frequency up to 160 MHz
- Zero wait cycle access to on-chip SRAM and Cache for program and data access over IRAM/DRAM interface
- Interrupt controller (INTC) with up to 31 vectored interrupts with programmable priority and threshold levels
- Debug module (DM) compliant with RISC-V debug specification v0.13 with external debugger support over an industry-standard JTAG/USB port
- Debugger direct system bus access (SBA) to memory and peripherals
- Hardware trigger compliant to RISC-V debug specification v0.13 with up to 8 breakpoints/watchpoints
- Physical memory protection (PMP) for up to 16 configurable regions
- 32-bit AHB system bus for peripheral access
- Configurable events for core performance metrics

9.3 Address Map

Below table shows address map of various regions accessible by CPU for instruction, data, system bus peripheral and debug.

Table 9-1. CPU Address Map

Name	Description	Starting Address	Ending Address	Access
IRAM	Instruction Address Map	0x4000_0000	0x47FF_FFFF	R/W
DRAM	Data Address Map	0x3800_0000	0x3FFF_FFFF	R/W
DM	Debug Address Map	0x2000_0000	0x27FF_FFFF	R/W
AHB	AHB Address Map	*default	*default	R/W

*default : Address not matching any of the specified ranges (IRAM, DRAM, DM) are accessed using AHB bus.

9.4 Configuration and Status Registers (CSRs)

9.4.1 Register Summary

Below is a list of CSRs available to the CPU. Except for the custom performance counter CSRs, all the implemented CSRs follow the standard mapping of bit fields as described in the RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10. It must be noted that even among the standard CSRs, not all bit fields have been implemented, limited by the subset of features implemented in the CPU. Refer to the next section for detailed description of the subset of fields implemented under each of these CSRs.

Name	Description	Address	Access
Machine Information CSRs			
mvendorid	Machine Vendor ID	0xF11	RO
marchid	Machine Architecture ID	0xF12	RO
mimpid	Machine Implementation ID	0xF13	RO
mhartid	Machine Hart ID	0xF14	RO
Machine Trap Setup CSRs			
mstatus	Machine Mode Status	0x300	R/W
misa [1]	Machine ISA	0x301	R/W
mtvec [2]	Machine Trap Vector	0x305	R/W
Machine Trap Handling CSRs			
mscratch	Machine Scratch	0x340	R/W
mepc	Machine Trap Program Counter	0x341	R/W
mcause [3]	Machine Trap Cause	0x342	R/W
mtval	Machine Trap Value	0x343	R/W
Physical Memory Protection (PMP) CSRs			
pmpcfg0	Physical memory protection configuration	0x3A0	R/W

[1] Although misa is specified as having both read and write access (R/W), its fields are hardwired and thus write has no effect. This is what would be termed WARL (Write Any Read Legal) in RISC-V terminology

[2] mtvec only provides configuration for trap handling in vectored mode with the base address aligned to 256 bytes

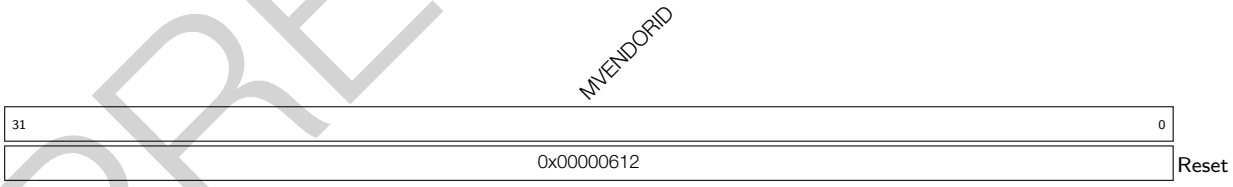
[3] External interrupt IDs reflected in mcause include even those IDs which have been reserved by RISC-V standard for core internal sources.

Name	Description	Address	Access
pmpcfg1	Physical memory protection configuration	0x3A1	R/W
pmpcfg2	Physical memory protection configuration	0x3A2	R/W
pmpcfg3	Physical memory protection configuration	0x3A3	R/W
pmpaddr0	Physical memory protection address register	0x3B0	R/W
pmpaddr1	Physical memory protection address register	0x3B1	R/W
....			
pmpaddr15	Physical memory protection address register	0x3BF	R/W
Trigger Module CSRs (shared with Debug Mode)			
tselect	Trigger Select Register	0x7A0	R/W
tdata1	Trigger Abstract Data 1	0x7A1	R/W
tdata2	Trigger Abstract Data 2	0x7A2	R/W
tcontrol	Global Trigger Control	0x7A5	R/W
Debug Mode CSRs			
dcsr	Debug Control and Status	0x7B0	R/W
dpc	Debug PC	0x7B1	R/W
dscratch0	Debug Scratch Register 0	0x7B2	R/W
dscratch1	Debug Scratch Register 1	0x7B3	R/W
Performance Counter CSRs (Custom) [4]			
mpcer	Machine Performance Counter Event	0x7E0	R/W
mpcmr	Machine Performance Counter Mode	0x7E1	R/W
mpccr	Machine Performance Counter Count	0x7E2	R/W

Note that if write/set/clear operation is attempted on any of the CSRs which are read-only (RO), as indicated in the above table, the CPU will generate illegal instruction exception.

9.4.2 Register Description

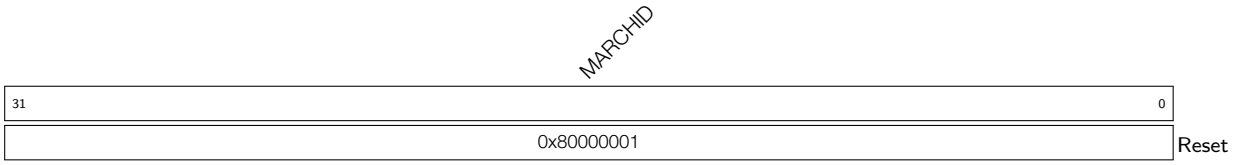
Register 9.1. mvendorid (0xF11)



MVENDORID Vendor ID. (RO)

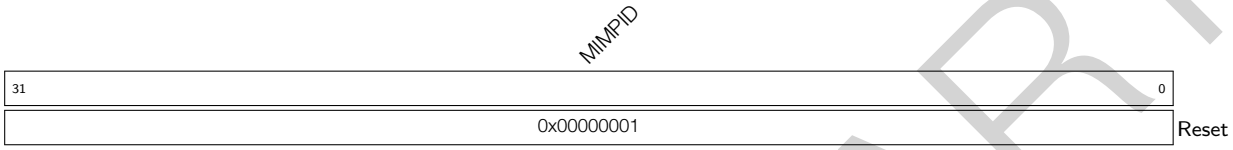
[4] These custom machine-mode CSRs have been implemented in the address space reserved by RISC-V standard for custom use

Register 9.2. marchid (0xF12)



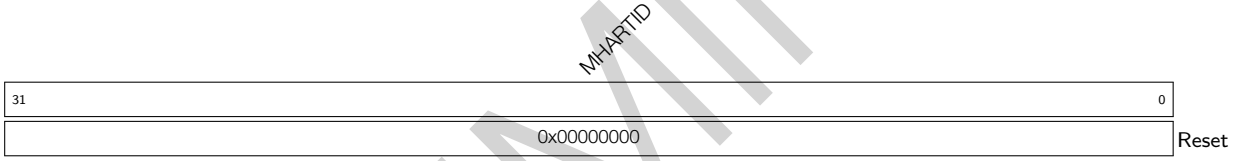
MARCHID Architecture ID. (RO)

Register 9.3. mimpid (0xF13)



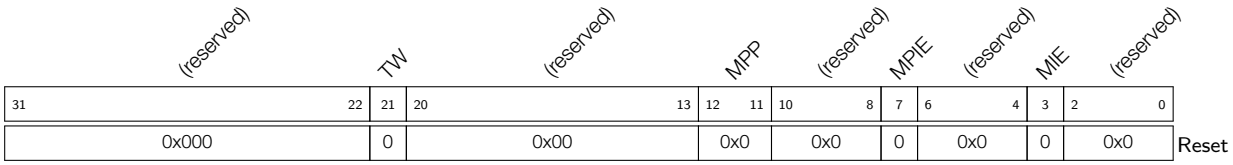
MIMPID Implementation ID. (RO)

Register 9.4. mhartid (0xF14)



MHARTID Hart ID. (RO)

Register 9.5. mstatus (0x300)



MIE Global machine mode interrupt enable. (R/W)

MPIE Previous MIE. (R/W)

MPP Machine previous privilege mode. (R/W)

Possible values:

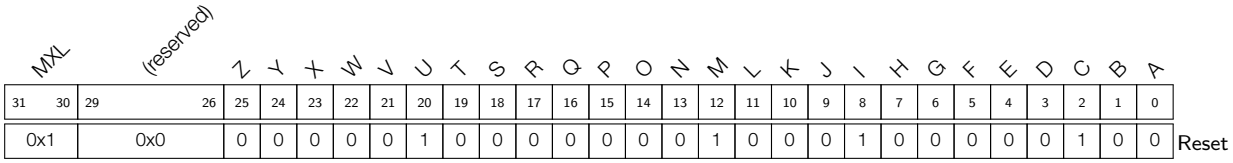
- 0x0: User mode
- 0x3: Machine mode

Note : Only lower bit is writable. Write to the higher bit is ignored as it is directly tied to the lower bit.

TW Timeout wait. (R/W)

If this bit is set, executing WFI (Wait-for-Interrupt) instruction in User mode will cause illegal instruction exception.

Register 9.6. misa (0x301)



MXL Machine XLEN = 1 (32-bit). (RO)

Z Reserved = 0. (RO)

Y Reserved = 0. (RO)

X Non-standard extensions present = 0. (RO)

W Reserved = 0. (RO)

V Reserved = 0. (RO)

U User mode implemented = 1. (RO)

T Reserved = 0. (RO)

S Supervisor mode implemented = 0. (RO)

R Reserved = 0. (RO)

Q Quad-precision floating-point extension = 0. (RO)

P Reserved = 0. (RO)

O Reserved = 0. (RO)

N User-level interrupts supported = 0. (RO)

M Integer Multiply/Divide extension = 1. (RO)

L Reserved = 0. (RO)

K Reserved = 0. (RO)

J Reserved = 0. (RO)

I RV32I base ISA = 1. (RO)

H Hypervisor extension = 0. (RO)

G Additional standard extensions present = 0. (RO)

F Single-precision floating-point extension = 0. (RO)

E RV32E base ISA = 0. (RO)

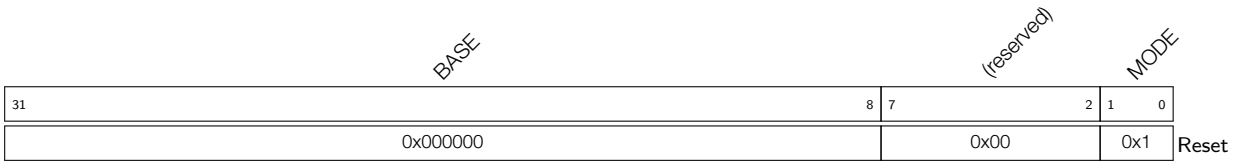
D Double-precision floating-point extension = 0. (RO)

C Compressed Extension = 1. (RO)

B Reserved = 0. (RO)

A Atomic Extension = 0. (RO)

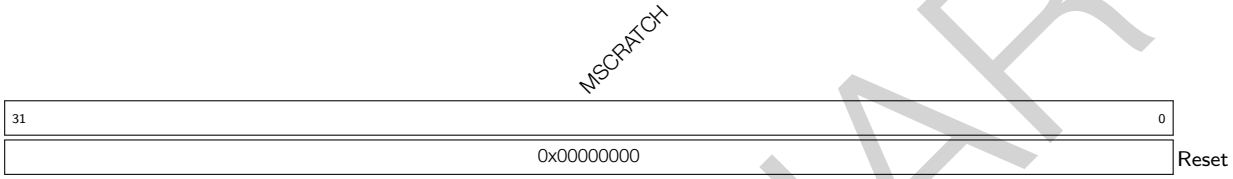
Register 9.7. mtvec (0x305)



MODE Only vectored mode **0x1** is available. (RO)

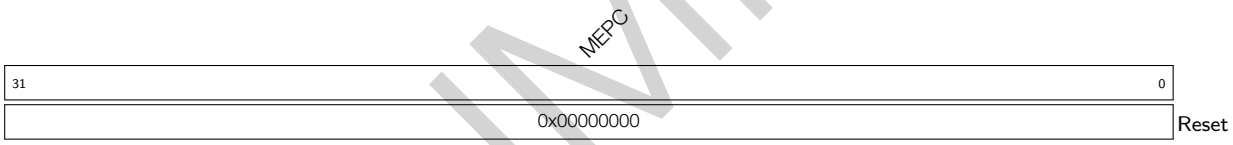
BASE Higher 24 bits of trap vector base address aligned to 256 bytes. (R/W)

Register 9.8. mscratch (0x340)



MSCRATCH Machine scratch register for custom use. (R/W)

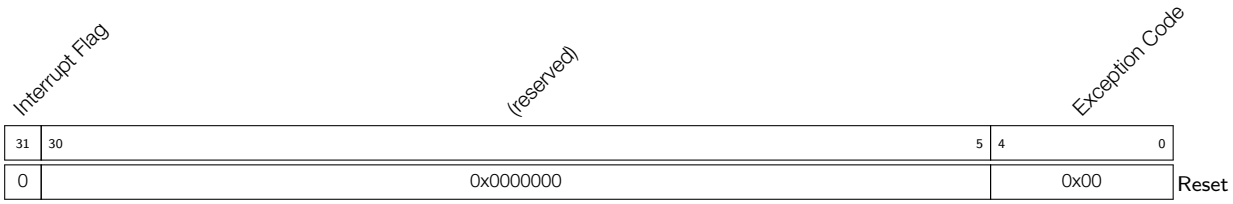
Register 9.9. mepc (0x341)



MEPC Machine trap/exception program counter. (R/W)

This is automatically updated with address of the instruction which was about to be executed while CPU encountered the most recent trap.

Register 9.10. mcause (0x342)



Exception Code This field is automatically updated with unique ID of the most recent exception or interrupt due to which CPU entered trap. (R/W)

Possible exception IDs are:

- 0x1: PMP Instruction access fault
- 0x2: Illegal Instruction
- 0x3: Hardware Breakpoint/Watchpoint or EBREAK
- 0x5: PMP Load access fault
- 0x7: PMP Store access fault
- 0x8: ECALL from U mode
- 0xb: ECALL from M mode

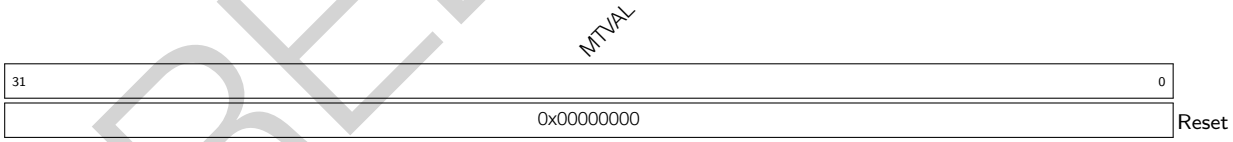
Note : Exception ID 0x0 (instruction access misaligned) is not present because CPU always masks the lowest bit of the address during instruction fetch.

Interrupt Flag This flag is automatically updated when CPU enters trap. (R/W)

If this is found to be set, indicates that the latest trap occurred due to interrupt. For exceptions it remains unset.

Note : The interrupt controller is using up IDs in range 1-31 for all external interrupt sources. This is different from the RISC-V standard which has reserved IDs in range 0-15 for core internal interrupt sources.

Register 9.11. mtval (0x343)



MTVAL Machine trap value. (R/W)

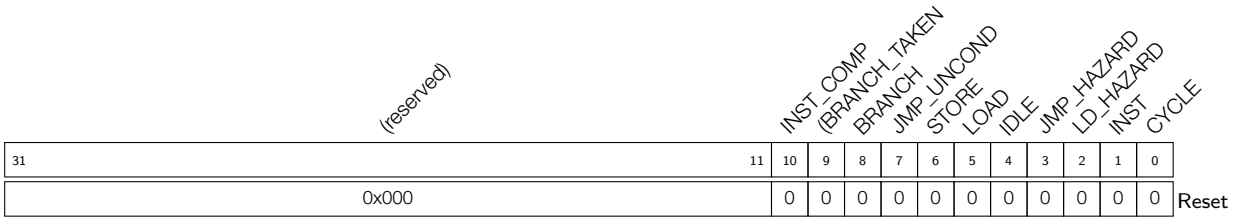
This is automatically updated with an exception dependent data which may be useful for handling that exception.

Data is to be interpreted depending upon exception IDs:

- 0x1: Faulting virtual address of instruction
- 0x2: Faulting instruction opcode
- 0x5: Faulting data address of load operation
- 0x7: Faulting data address of store operation

Note : The value of this register is not valid for other exception IDs and interrupts.

Register 9.12. mpcer (0x7E0)



INST_COMP Count Compressed Instructions. (R/W)

BRANCH_TAKEN Count Branches Taken. (R/W)

BRANCH Count Branches. (R/W)

JMP_UNCOND Count Unconditional Jumps. (R/W)

STORE Count Stores. (R/W)

LOAD Count Loads. (R/W)

IDLE Count IDLE Cycles. (R/W)

JMP_HAZARD Count Jump Hazards. (R/W)

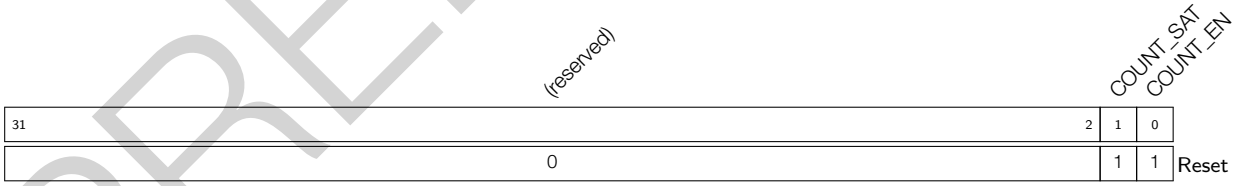
LD_HAZARD Count Load Hazards. (R/W)

INST Count Instructions. (R/W)

CYCLE Count Clock Cycles. (R/W)

Note: Each bit selects a specific event for counter to increment. If more than one event is selected and occurs simultaneously, then counter increments by one only.

Register 9.13. mpcmr (0x7E1)



COUNT_SAT Counter Saturation Control. (R/W)

Possible values:

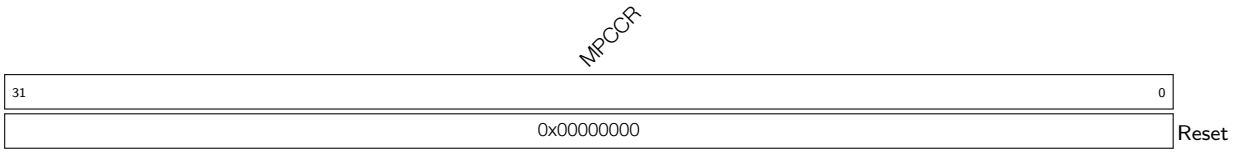
- 0: Overflow on maximum value
- 1: Halt on maximum value

COUNT_EN Counter Enable Control. (R/W)

Possible values:

- 0: Disabled
- 1: Enabled

Register 9.14. mpccr (0x7E2)



MPCCR Machine Performance Counter Value. (R/W)

9.5 Interrupt Controller

9.5.1 Features

The interrupt controller allows capturing, masking and dynamic prioritization of interrupt sources routed from peripherals to the RISC-V CPU. It supports:

- Up to 31 asynchronous interrupts with unique IDs (1-31)
- Configurable via read/write to memory mapped registers
- 15 levels of priority, programmable for each interrupt
- Support for both level and edge type interrupt sources
- Programmable global threshold for masking interrupts with lower priority
- Interrupts IDs mapped to trap-vector address offsets

9.5.2 Functional Description

Each interrupt ID has 5 properties associated with it:

1. Enable State (0-1):
   - Determines if an interrupt is enabled to be captured and serviced by the CPU.
   - Programmed by writing the corresponding bit in INT_ENABLE_REG.
2. Type (0-1):
   - Enables latching the state of an interrupt signal on its rising edge.
   - Programmed by writing the corresponding bit in INT_TYPE_REG.
   - An interrupt for which type is kept 0 is referred as a 'level' type interrupt.
   - An interrupt for which type is set to 1 is referred as an 'edge' type interrupt.
3. Priority (1-15):
   - Determines which interrupt, among multiple pending interrupts, the CPU will service first.
   - Programmed by writing to the INT_PRIORITY_*n*_REG for a particular ID *n* in range (1-31).
   - Enabled interrupts with priorities zero or less than the threshold value in INT_THRESH_REG are masked.
   - Priority levels increase from 1 (lowest) to 15 (highest).
   - Interrupts with same priority are statically prioritized by their IDs, lowest ID having highest priority.
4. Pending State (0-1):
   - Reflects the captured state of an enabled and unmasked interrupt signal.
   - For each interrupt ID, the corresponding bit in read-only INT_EIP_REG gives its pending state.
   - A pending interrupt will cause CPU to enter trap if no other pending interrupt has higher priority.
   - A pending interrupt is said to be 'claimed' if it preempts the CPU and causes it to jump to the corresponding trap vector address.

- All pending interrupts which are yet to be serviced are termed as 'unclaimed'.

5. Clear State (0-1):
   - Toggling this will clear the pending state of claimed edge-type interrupts only.
   - Toggled by first setting and then clearing the corresponding bit in INT_CLEAR_REG.
   - Pending state of a level type interrupt is unaffected by this and must be cleared from source.
   - Pending state of an unclaimed edge type interrupt can be flushed, if required, by first clearing the corresponding bit in INT_ENABLE_REG and then toggling same bit in INT_CLEAR_REG.

When CPU services a pending interrupt, it:

- saves the address of the current un-executed instruction in mepc for resuming execution later.
- updates the value of mcause with the ID of the interrupt being serviced.
- copies the state of MIE into MPIE, and subsequently clears MIE, thereby disabling interrupts globally.
- enters trap by jumping to a word-aligned offset of the address stored in mtvec.

Table 9-3 shows the mapping of each interrupt ID with the corresponding trap-vector address. In short, the word aligned trap address for an interrupt with a certain $ID = i$ can be calculated as $(mtvec + 4i)$.

Note : $ID = 0$ is unavailable and therefore cannot be used for capturing interrupts. This is because the corresponding trap vector address (mtvec + 0x00) is reserved for exceptions.

Table 9-3. ID wise map of Interrupt Trap-Vector Addresses

ID	Address	ID	Address	ID	Address	ID	Address
0	NA	8	mtvec + 0x20	16	mtvec + 0x40	24	mtvec + 0x60
1	mtvec + 0x04	9	mtvec + 0x24	17	mtvec + 0x44	25	mtvec + 0x64
2	mtvec + 0x08	10	mtvec + 0x28	18	mtvec + 0x48	26	mtvec + 0x68
3	mtvec + 0x0c	11	mtvec + 0x2c	19	mtvec + 0x4c	27	mtvec + 0x6c
4	mtvec + 0x10	12	mtvec + 0x30	20	mtvec + 0x50	28	mtvec + 0x70
5	mtvec + 0x14	13	mtvec + 0x34	21	mtvec + 0x54	29	mtvec + 0x74
6	mtvec + 0x18	14	mtvec + 0x38	22	mtvec + 0x58	30	mtvec + 0x78
7	mtvec + 0x1c	15	mtvec + 0x3c	23	mtvec + 0x5c	31	mtvec + 0x7c

After jumping to the trap-vector, the execution flow is dependent on software implementation, although it can be presumed that the interrupt will get handled (and cleared) in some interrupt service routine (ISR) and later the normal execution will resume once the CPU encounters MRET instruction.

Upon execution of MRET instruction, the CPU:

- copies the state of MPIE back into MIE, and subsequently clears MPIE. This means that if previously MPIE was set, then, after MRET, MIE will be set, thereby enabling interrupts globally.
- jumps to the address stored in mepc and resumes execution.

It is possible to perform software assisted nesting of interrupts inside an ISR as explained in 9.5.3.

The below listed points outline the functional behavior of the controller:

- Only if an interrupt has non-zero priority, higher or equal to the value in the threshold register, will it be

reflected in INT_EIP_REG.

- If an interrupt is visible in INT_EIP_REG and has yet to be serviced, then it's possible to mask it (and thereby prevent the CPU from servicing it) by either lowering the value of its priority or increasing the global threshold.
- If an interrupt, visible in INT_EIP_REG, is to be flushed (and prevented from being serviced at all), then it must be disabled (and cleared if it is of edge type).

9.5.3 Suggested Operation

9.5.3.1 Latency Aspects

There is latency involved while configuring the Interrupt Controller.

In steady state operation, the Interrupt Controller has a fixed latency of 4 cycles. Steady state means that no changes have been made to the Interrupt Controller registers recently. This implies that any interrupt that is asserted to the controller will take exactly 4 cycles before the CPU starts processing the interrupt. This further implies that CPU may execute up to 5 instructions before the preemption happens.

Whenever any of its registers are modified, the Interrupt Controller enters into transient state, which may take up to 4 cycles for it to settle down into steady state again. During this transient state, the ordering of interrupts may not be predictable, and therefore, a few safety measures need to be taken in software to avoid any synchronization issues.

Also, it must be noted that the Interrupt Controller configuration registers lie in the APB address range, hence any R/W access to these registers may take multiple cycles to complete.

In consideration of above mentioned characteristics, users are advised to follow the sequence described below, whenever modifying any of the Interrupt Controller registers:

1. save the state of MIE and clear MIE to 0
2. read-modify-write one or more Interrupt Controller registers
3. execute FENCE instruction to wait for any pending write operations to complete
4. finally, restore the state of MIE

Due to its critical nature, it is recommended to disable interrupts globally (MIE=0) beforehand, whenever configuring interrupt controller registers, and then restore MIE right after, as shown in the sequence above.

After execution of the sequence above, the Interrupt Controller will resume operation in steady state.

9.5.3.2 Configuration Procedure

By default, interrupts are disabled globally, since the reset value of MIE bit in mstatus is 0. Software must set MIE=1 after initialization of the interrupt stack (including setting mtvec to the interrupt vector address) is done.

During normal execution, if an interrupt *n* is to be enabled, the below sequence may be followed:

1. save the state of MIE and clear MIE to 0
2. depending upon the type of the interrupt (edge/level), set/unset the *n*th bit of INT_TYPE_REG
3. set the priority by writing a value to INT_PRIORITY_*n*_REG in range 1(lowest) to 15 (highest)

4. set the *n*th bit of INT_ENABLE_REG
5. execute FENCE instruction
6. restore the state of MIE

When one or more interrupts become pending, the CPU acknowledges (claims) the interrupt with the highest priority and jumps to the trap vector address corresponding to the interrupt's ID. Software implementation may read mcause to infer the type of trap (mcause(31) is 1 for interrupts and 0 for exceptions) and then the ID of the interrupt (mcause(4-0) gives ID of interrupt or exception). This inference may not be necessary if each entry in the trap vector are jump instructions to different trap handlers. Ultimately, the trap handler(s) will redirect execution to the appropriate ISR for this interrupt.

Upon entering into an ISR, software must toggle the *n*th bit of INT_CLEAR_REG if the interrupt is of edge type, or clear the source of the interrupt if it is of level type.

Software may also update the value of INT_THRESH_REG and program MIE=1 for allowing higher priority interrupts to preempt the current ISR (nesting), however, before doing so, all the state CSRs must be saved (mepc, mstatus, mcause, etc.) since they will get overwritten due to occurrence of such an interrupt. Later, when exiting the ISR, the values of these CSRs must be restored.

Finally, after the execution returns from the ISR back to the trap handler, MRET instruction is used to resume normal execution.

Later, if the *n* interrupt is no longer needed and needs to be disabled, the following sequence may be followed:

1. save the state of MIE and clear MIE to 0
2. check if the interrupt is pending in INT_EIP_REG
3. set/unset the *n*th bit of INT_ENABLE_REG
4. if the interrupt is of edge type and was found to be pending in step 2 above, *n*th bit of INT_CLEAR_REG must be toggled, so that its pending status gets flushed
5. execute FENCE instruction
6. restore the state of MIE

Above is only a suggested scheme of operation. Actual software implementation may vary.

9.5.4 Register Summary

The addresses in this section are relative to Interrupt Controller base address provided in Table 3-4 in Chapter 3 *System and Memory*.

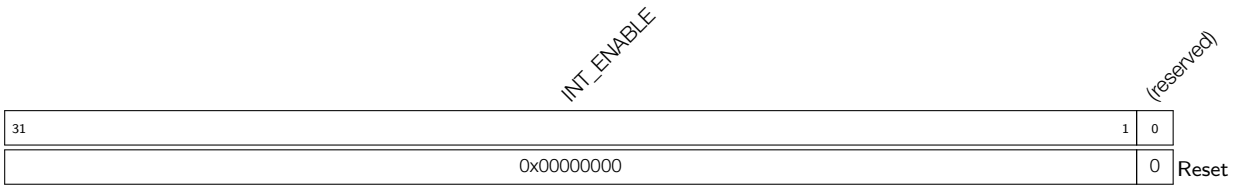
Name	Description	Address	Access
INT_ENABLE_REG	Enables assertion of interrupt to the CPU	0x0104	R/W
INT_TYPE_REG	Specify interrupt type as level/edge	0x0108	R/W
INT_CLEAR_REG	Write to clear "pulse" type interrupts	0x010C	R/W
INT_EIP_REG	External/peripheral interrupt pending status to CPU	0x0110	RO
INT_PRIORITY_1_REG	Priority setting for interrupt ID=1	0x0118	R/W
INT_PRIORITY_2_REG	Priority setting for interrupt ID=2	0x011C	R/W
INT_PRIORITY_3_REG	Priority setting for interrupt ID=3	0x0120	R/W

Name	Description	Address	Access
INT_PRIORITY_4_REG	Priority setting for interrupt ID=4	0x0124	R/W
INT_PRIORITY_5_REG	Priority setting for interrupt ID=5	0x0128	R/W
INT_PRIORITY_6_REG	Priority setting for interrupt ID=6	0x012C	R/W
INT_PRIORITY_7_REG	Priority setting for interrupt ID=7	0x0130	R/W
INT_PRIORITY_8_REG	Priority setting for interrupt ID=8	0x0134	R/W
INT_PRIORITY_9_REG	Priority setting for interrupt ID=9	0x0138	R/W
INT_PRIORITY_10_REG	Priority setting for interrupt ID=10	0x013C	R/W
INT_PRIORITY_11_REG	Priority setting for interrupt ID=11	0x0140	R/W
INT_PRIORITY_12_REG	Priority setting for interrupt ID=12	0x0144	R/W
INT_PRIORITY_13_REG	Priority setting for interrupt ID=13	0x0148	R/W
INT_PRIORITY_14_REG	Priority setting for interrupt ID=14	0x014C	R/W
INT_PRIORITY_15_REG	Priority setting for interrupt ID=15	0x0150	R/W
INT_PRIORITY_16_REG	Priority setting for interrupt ID=16	0x0154	R/W
INT_PRIORITY_17_REG	Priority setting for interrupt ID=17	0x0158	R/W
INT_PRIORITY_18_REG	Priority setting for interrupt ID=18	0x015C	R/W
INT_PRIORITY_19_REG	Priority setting for interrupt ID=19	0x0160	R/W
INT_PRIORITY_20_REG	Priority setting for interrupt ID=20	0x0164	R/W
INT_PRIORITY_21_REG	Priority setting for interrupt ID=21	0x0168	R/W
INT_PRIORITY_22_REG	Priority setting for interrupt ID=22	0x016C	R/W
INT_PRIORITY_23_REG	Priority setting for interrupt ID=23	0x0170	R/W
INT_PRIORITY_24_REG	Priority setting for interrupt ID=24	0x0174	R/W
INT_PRIORITY_25_REG	Priority setting for interrupt ID=25	0x0178	R/W
INT_PRIORITY_26_REG	Priority setting for interrupt ID=26	0x017C	R/W
INT_PRIORITY_27_REG	Priority setting for interrupt ID=27	0x0180	R/W
INT_PRIORITY_28_REG	Priority setting for interrupt ID=28	0x0184	R/W
INT_PRIORITY_29_REG	Priority setting for interrupt ID=29	0x0188	R/W
INT_PRIORITY_30_REG	Priority setting for interrupt ID=30	0x018C	R/W
INT_PRIORITY_31_REG	Priority setting for interrupt ID=31	0x0190	R/W
INT_THRESH_REG	Priority threshold setting for interrupt assertion to CPU	0x0194	R/W

9.5.5 Register Description

The addresses in this section are relative to Interrupt Controller base address provided in Table 3-4 in Chapter 3 *System and Memory*.

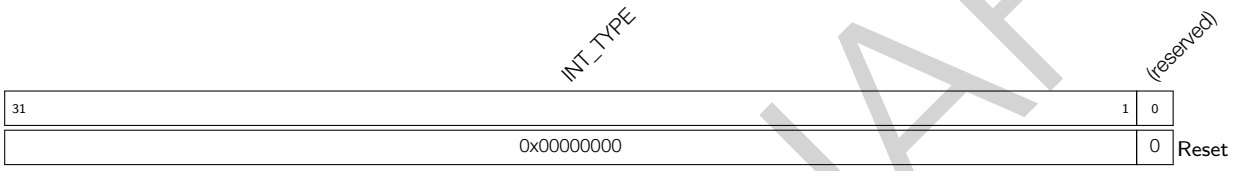
Register 9.15. INT_ENABLE_REG (0x0104)



INT_ENABLE[*n*] Setting *n*th bit enables assertion of *n*th interrupt to the CPU. (R/W)

- 0: Disabled;
- 1: Enabled;

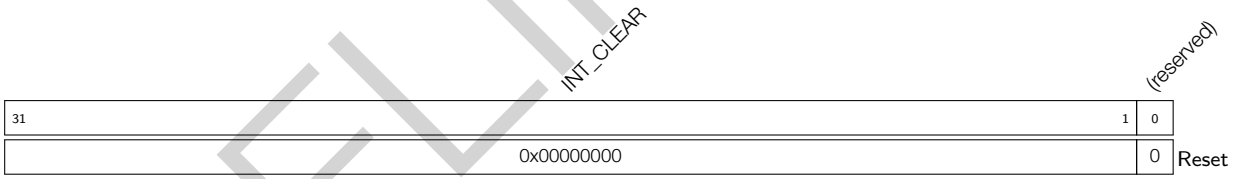
Register 9.16. INT_TYPE_REG (0x0108)



INT_TYPE[*n*] Setting *n*th bit enables capturing the rising edge of *n*th interrupt. (R/W)

- 0: Level type (signal level detection);
- 1: Pulse type (rising edge detection);

Register 9.17. INT_CLEAR_REG (0x010C)

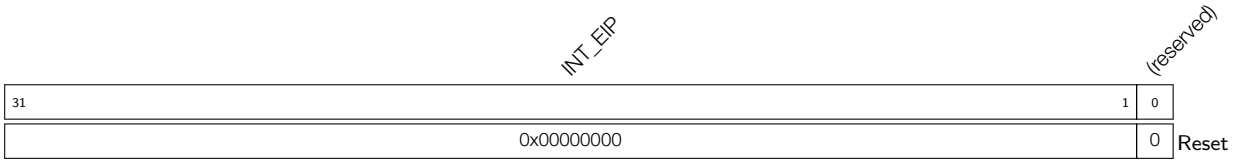


INT_CLEAR[*n*] Set *n*th bit to clear pending status of the *n*th interrupt. (R/W)

This is only useful for "pulse" type interrupts, since "level" type interrupts must be cleared at source. Note that the set bit must be manually toggled back to 0 afterwards.

- 0: Don't care;
- 1: Clear pending status;

Register 9.18. INT_EIP_REG (0x0110)

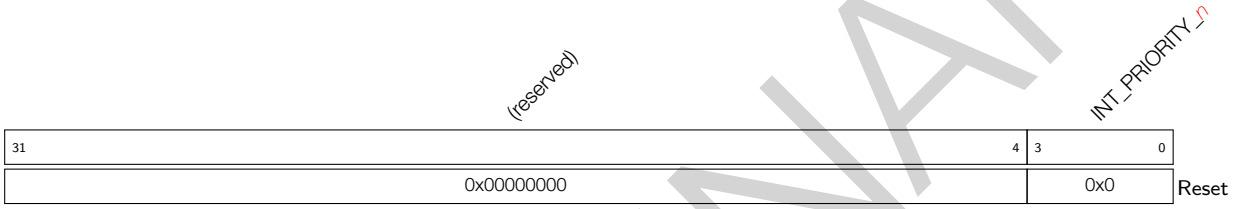


INT_EIP[*n*] Read *n*th bit to get the pending status of *n*th interrupt to CPU. (RO)

Only enabled and above threshold interrupts are reflected here.

- 0: Not pending
- 1: Pending

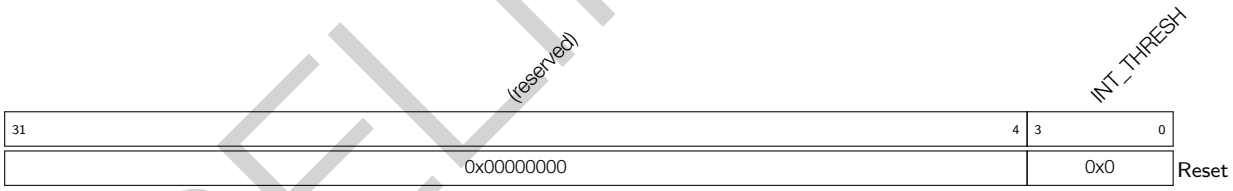
Register 9.19. INT_PRIORITY_*n*_REG (*n*: 1-31) (0x0114+4n*)**



INT_PRIORITY_*n* Writing a 4-bit value to *n*th register configures priority of *n*th interrupt. (R/W)

Note : Interrupts with 0 priority are masked regardless of threshold value.

Register 9.20. INT_THRESH_REG (0x0194)



INT_THRESH Writing a 4-bit value configures the global priority threshold for all interrupts. (R/W)

All interrupts with priority lower than the threshold are masked.

Note : Interrupts with 0 priority are masked regardless of threshold value.

9.6 Debug

9.6.1 Overview

This section describes how to debug and test software running on CPU core. Debug support is provided through standard JTAG pins and complies to RISC-V External Debug Support Specification version 0.13.

Figure 9-2 below shows the main components of External Debug Support.

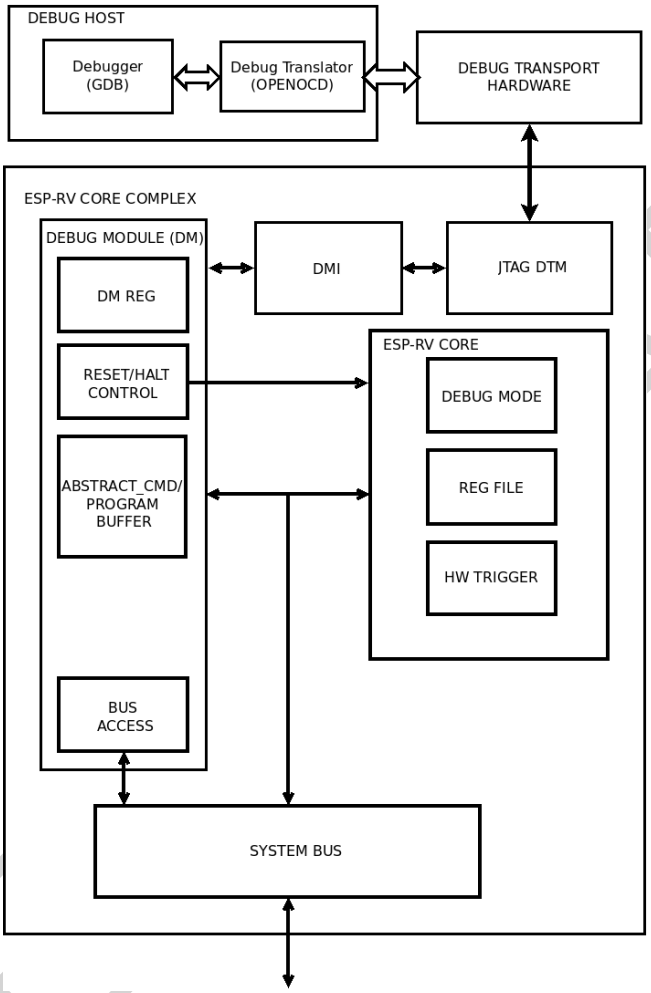


Figure 9-2. Debug System Overview

The user interacts with the Debug Host (eg. laptop), which is running a debugger (eg. gdb). The debugger communicates with a Debug Translator (eg. OpenOCD, which may include a hardware driver) to communicate with Debug Transport Hardware (eg. Olimex USB-JTAG adapter). The Debug Transport Hardware connects the Debug Host to the ESP-RV Core's Debug Transport Module (DTM) through standard JTAG interface. The DTM provides access to the Debug Module (DM) using the Debug Module Interface (DMI).

The DM allows the debugger to halt the core. Abstract commands provide access to its GPRs (general purpose registers). The Program Buffer allows the debugger to execute arbitrary code on the core, which allows access to additional CPU core state. Alternatively, additional abstract commands can provide access to additional CPU core state. ESP-RV core contains Trigger Module supporting 8 triggers. When trigger conditions are met, cores will halt spontaneously and inform the debug module that they have halted.

System bus access block allows memory and peripheral register access without using RISC-V core.

9.6.2 Features

Basic debug functionality supports below features.

- Provides necessary information about the implementation to the debugger.
- Allows the CPU core to be halted and resumed.
- CPU core registers (including CSR's) can be read/written by debugger.
- CPU can be debugged from the first instruction executed after reset.
- CPU core can be reset through debugger.
- CPU can be halted on software breakpoint (planted breakpoint instruction).
- Hardware single-stepping.
- Execute arbitrary instructions in the halted CPU by means of the program buffer. 16-word program buffer is supported.
- System bus access is supported through word aligned address access.
- Supports eight Hardware Triggers (can be used as breakpoints/watchpoints) as described in Section 9.7.

9.6.3 Functional Description

As mentioned earlier, Debug Scheme conforms to RISC-V External Debug Support Specification version 0.13. Please refer the specs for functional operation details.

9.6.4 Register Summary

Below is the list of Debug CSR's supported by ESP-RV core.

Name	Description	Address	Access
dcsr	Debug Control and Status	0x7B0	R/W
dpc	Debug PC	0x7B1	R/W
dscratch0	Debug Scratch Register 0	0x7B2	R/W
dscratch1	Debug Scratch Register 1	0x7B3	R/W

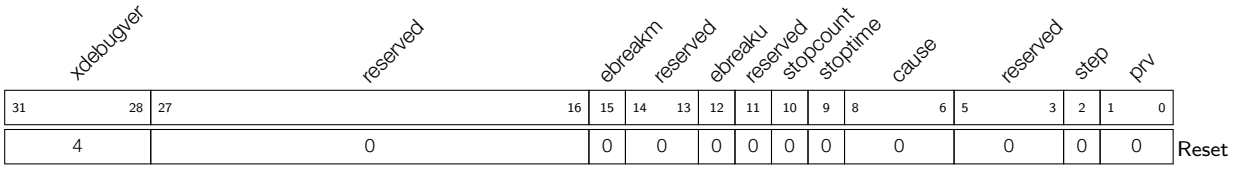
.

All the debug module registers are implemented in conformance to RISC-V External Debug Support Specification version 0.13. Please refer it for more details.

9.6.5 Register Description

Below are the details of Debug CSR's supported by ESP-RV core

Register 9.21. dcsr (0x7B0)



xdebugver Debug version. (RO)

- 4: External debug support exists

ebreakm When 1, ebreak instructions in Machine Mode enter Debug Mode. (R/W)

ebreaku When 1, ebreak instructions in User/Application Mode enter Debug Mode. (R/W)

stopcount This bit is not implemented. Debugger will always read this bit as 0. (RO)

stoptime This feature is not implemented. Debugger will always read this bit as 0. (RO)

cause Explains why Debug Mode was entered. When there are multiple reasons to enter Debug Mode in a single cycle, the cause with the highest priority number is the one written.

1. An ebreak instruction was executed. (priority 3)
2. The Trigger Module caused a halt. (priority 4)
3. haltreq was set. (priority 2)
4. The CPU core single stepped because step was set. (priority 1)

Other values are reserved for future use. (RO)

step When set and not in Debug Mode, the core will only execute a single instruction and then enter Debug Mode. Interrupts are **enabled*** when this bit is set. If the instruction does not complete due to an exception, the core will immediately enter Debug Mode before executing the trap handler, with appropriate exception registers set. (R/W)

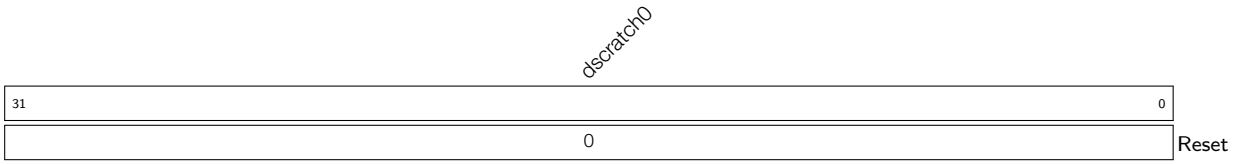
prv Contains the privilege level the core was operating in when Debug Mode was entered. A debugger can change this value to change the core's privilege level when exiting Debug Mode. Only **0x3** (machine mode) and **0x0**(user mode) are supported.

*__Note__: Different from RISC-V Debug specification 0.13

Register 9.22. dpc (0x7B1)

dpc Upon entry to debug mode, dpc is written with the virtual address of the instruction that encountered the exception. When resuming, the CPU core's PC is updated to the virtual address stored in dpc. A debugger may write dpc to change where the CPU resumes. (R/W)

Register 9.23. dscratch0 (0x7B2)



dscratch0 Used by Debug Module internally. (R/W)

Register 9.24. dscratch1 (0x7B3)

dscratch1 Used by Debug Module internally. (R/W)

9.7 Hardware Trigger

9.7.1 Features

Hardware Trigger module provides breakpoint and watchpoint capability for debugging. It includes the following features:

- 8 independent trigger units
- each unit can be configured for matching the address of program counter or load-store accesses
- can preempt execution by causing breakpoint exception
- can halt execution and transfer control to debugger
- support NAPOT (naturally aligned power of two) address encoding

9.7.2 Functional Description

The Hardware Trigger module provides four CSRs, which are listed under register summary section. Among these, tdata1 and tdata2 are abstract CSRs, which means they are shadow registers for accessing internal registers for each of the eight trigger units, one at a time.

To choose a particular trigger unit write the index (0-7) of that unit into tselect CSR. When tselect is written with a valid index, the abstract CSRs tdata1 and tdata2 are automatically mapped to reflect internal registers of that trigger unit. Each trigger unit has two internal registers, namely mcontrol and maddress, which are mapped to tdata1 and tdata2, respectively.

Writing larger than allowed indexes to tselect will clip the written value to the largest valid index, which can be read back. This property may be used for enumerating the number of available triggers during initialization or when using a debugger.

Since software or debugger may need to know the type of the selected trigger to correctly interpret tdata1 and tdata2, the 4 bits (31-28) of tdata1 encodes the type of the selected trigger. This type field is read-only and always provides a value of 0x2 for every trigger, which stands for match type trigger, hence, it is inferred that tdata1 and tdata2 are to be interpreted as mcontrol and maddress. The information regarding other possible values can be found in the RISC-V Debug Specification v0.13, but this trigger module only supports type 0x2.

Once a trigger unit has been chosen by writing its index to tselect, it will become possible to configure it by setting the appropriate bits in mcontrol CSR (tdata1) and writing the target address to maddress CSR (tdata2).

Each trigger unit can be configured to either cause breakpoint exception or enter debug mode, by writing to the action bit of mcontrol. This bit can only be written from debugger, thus by default a trigger, if enabled, will cause breakpoint exception.

mcontrol for each trigger unit has a hit bit which may be read, after CPU halts or enters exception, to find out if this was the trigger unit that fired. This bit is set as soon as the corresponding trigger fires, but it has to be manually cleared before resuming operation. Although, failing to clear it doesn't affect normal execution in any way.

Each trigger unit only supports match on address, although this address could either be that of a load/store access or the virtual address of an instruction. The address and size of a region are specified by writing to maddress (tdata2) CSR for the selected trigger unit. Larger than 1 byte region sizes are specified through NAPOT (naturally aligned power of two) encoding (see Table 9-6) and enabled by setting match bit in mcontrol. Note that

for NAPOT encoded addresses, by definition, the start address is constrained to be aligned to (i.e. an integer multiple of) the region size.

Table 9-6. NAPOT encoding for maddress

maddress(31-0)	Start Address	Size (bytes)
aaa...aaaaaaaaa0	aaa...aaaaaaaaa0	2
aaa...aaaaaaaa01	aaa...aaaaaaaa00	4
aaa...aaaaaaa011	aaa...aaaaaaa000	8
aaa...aaaaaa0111	aaa...aaaaaa0000	16
	....	
a01...1111111111	a00...0000000000	2^{31}

tcontrol CSR is common to all trigger units. It is used for preventing triggers from causing repeated exceptions in machine-mode while execution is happening inside a trap handler. This also disables breakpoint exceptions inside ISRs by default, although, it is possible to manually enable this right before entering an ISR, for debugging purposes. This CSR is not relevant if a trigger is configured to enter debug mode.

9.7.3 Trigger Execution Flow

When hart is halted and enters debug mode due to the firing of a trigger (action = 1):

- dpc is set to current PC (in decode stage)
- cause field in dcsr is set to 2, which means halt due to trigger
- hit bit is set to 1, corresponding to the trigger(s) which fired

When hart goes into trap due to the firing of a trigger (action = 0) :

- mepc is set to current PC (in decode stage)
- mcause is set to 3, which means breakpoint exception
- mpte is set to the value in mte right before trap
- mte is set to 0
- hit bit is set to 1, corresponding to the trigger(s) which fired

Note : If two different triggers fire at the same time, one with action = 0 and another with action = 1, then hart is halted and enters debug mode.

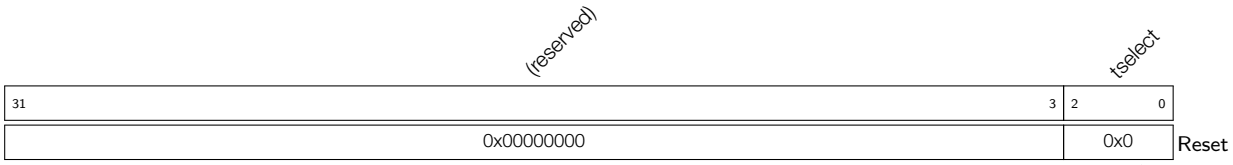
9.7.4 Register Summary

Below is a list of Trigger Module CSRs supported by the CPU. These are only accessible from machine-mode.

Name	Description	Address	Access
tselect	Trigger Select Register	0x7A0	R/W
tdata1	Trigger Abstract Data 1	0x7A1	R/W
tdata2	Trigger Abstract Data 2	0x7A2	R/W
tcontrol	Global Trigger Control	0x7A5	R/W

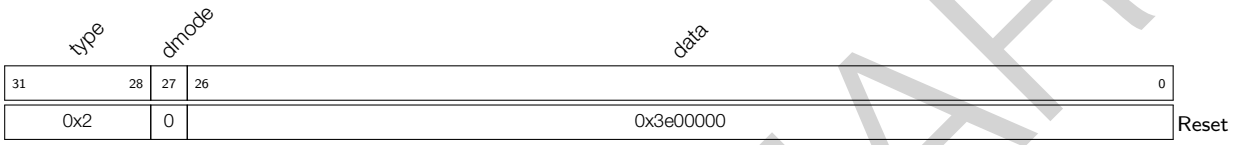
9.7.5 Register Description

Register 9.25. tselect (0x7A0)



tselect Index (0-7) of the selected trigger unit. (R/W)

Register 9.26. tdata1 (0x7A1)



type Type of trigger. (RO)

This field is reserved since only match type (0x2) triggers are supported.

dmode This is set to 1 if a trigger is being used by the debugger. (R/W *)

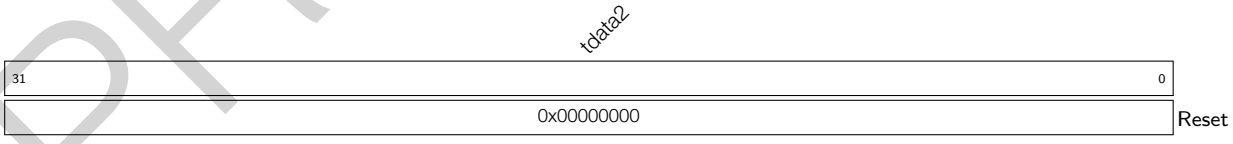
- 0: Both Debug and M-mode can write the tdata1 and tdata2 registers at the selected tselect.
- 1: Only Debug Mode can write the tdata1 and tdata2 registers at the selected tselect. Writes from other modes are ignored.

** Note : Only writable from debug mode.*

data Abstract tdata1 content. (R/W)

This will always be interpreted as fields of mcontrol since only match type (0x2) triggers are supported.

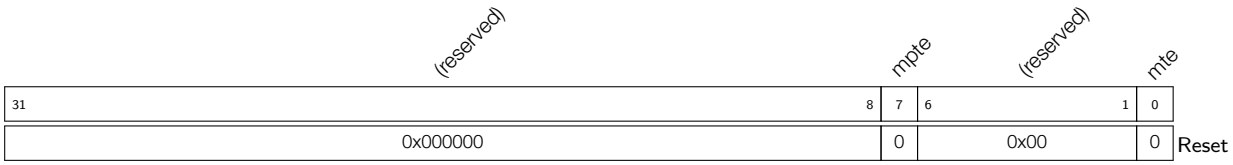
Register 9.27. tdata2 (0x7A2)



tdata2 Abstract tdata2 content. (R/W)

This will always be interpreted as maddress since only match type (0x2) triggers are supported.

Register 9.28. tcontrol (0x7A5)



mpte Machine mode previous trigger enable bit. (R/W)

- When CPU is taking a machine mode trap, the value of mte is automatically pushed into this.
- When CPU is executing MRET, its value is popped back into mte, so this becomes 0.

mte Machine mode trigger enable bit. (R/W)

- When CPU is taking a machine mode trap, its value is automatically pushed into mpte, so this becomes 0 and triggers with action=0 are disabled globally.
- When CPU is executing MRET, the value of mpte is automatically popped back into this.

Register 9.29. mcontrol (0x7A1)

(reserved)	dmode	(reserved)	hit	(reserved)	action	(reserved)	match	m	(reserved)	u	execute	store	load	
31 28	27	26 21	20	19 16	15 12	11	10 7	6	5 4	3	2	1	0	
0x2	0	0x1f	0	0	0	0	0	0	0	0	0	0	0	Reset

dmode Same as dmode in tdata1.

hit This is found to be 1 if the selected trigger had fired previously. (R/W)
This bit is to be cleared manually.

action Write this for configuring the selected trigger to perform one of the available actions when firing. (R/W)
Valid options are:

- 0x0: cause breakpoint exception.
- 0x1: enter debug mode (only valid when dmode = 1)

Note : Writing an invalid value will set this to the default value 0x0.

match Write this for configuring the selected trigger to perform one of the available matching operations on a data/instruction address. (R/W) Valid options are:

- 0x0: exact byte match, i.e. address corresponding to one of the bytes in an access must match the value of maddress exactly.
- 0x1: NAPOT match, i.e. at least one of the bytes of an access must lie in the NAPOT region specified in maddress.

Note : Writing a larger value will clip it to the largest possible value 0x1.

m Set this for enabling selected trigger to operate in machine mode. (R/W)

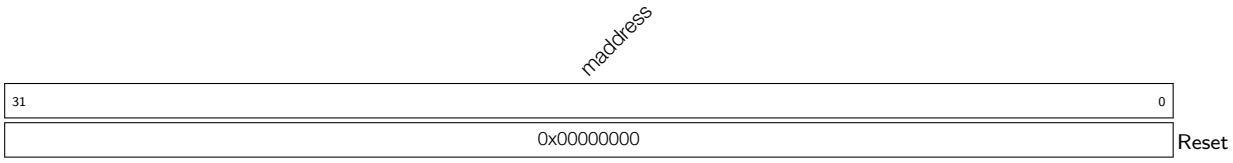
u Set this for enabling selected trigger to operate in user mode. (R/W)

execute Set this for configuring the selected trigger to fire right before an instruction with matching virtual address is executed by the CPU. (R/W)

store Set this for configuring the selected trigger to fire right before a store operation with matching data address is executed by the CPU. (R/W)

load Set this for configuring the selected trigger to fire right before a load operation with matching data address is executed by the CPU. (R/W)

Register 9.30. maddress (0x7A2)



maddress Address used by the selected trigger when performing match operation. (R/W)

This is decoded as NAPOT when match=1 in mcontrol.

9.8 Memory Protection

9.8.1 Overview

The CPU core includes a physical memory protection unit, which can be used by software to set memory access privileges (read, write and execute permissions) for required memory regions. However it is not fully compliant to the Physical Memory Protection (PMP) description specified in **RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10**. Details of existing non-conformance are provided in next section.

For detailed understanding of the RISC-V PMP concept, please refer to RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10.

9.8.2 Features

The PMP unit can be used to restrict access to physical memory. It supports 16 regions and a minimum granularity of 4 bytes. Below are the current non-conformance with PMP description from RISC-V Privilege specifications:

- Static priority i.e. overlapping regions are not supported
- Maximum supported NAPOT range is 1 GB

As per RISC-V Privilege specifications, PMP entries should be statically prioritized and the lowest-numbered PMP entry that matches any address byte of an access will determine whether that access succeeds or fails. This means, when any address matches more than one PMP entry i.e. overlapping regions among different PMP entries, lowest number PMP entry will decide whether such address access will succeed or fail.

However, RISC-V CPU PMP unit in ESP32-C3 does not implement static priority. So, software should make sure that all enabled PMP entries are programmed with unique regions i.e. without any region overlap among them. If software still tries to program multiple PMP entries with overlapping region having contradicting permissions, then access will succeed if it matches at least one of enabled PMP entries. An exception will be generated, if access matches none of the enabled PMP entries.

9.8.3 Functional Description

Software can program the PMP unit's configuration and address registers in order to contain faults and support secure execution. PMP CSR's can only be programmed in machine-mode. Once enabled, write, read and execute permission checks are applied to all the accesses in user-mode as per programmed values of enabled 16 pmpcfgX and pmpaddrX registers (refer Register Summary).

By default, PMP grants permission to all accesses in machine-mode and revokes permission of all access in user-mode. This implies that it is mandatory to program address range and valid permissions in pmpcfg and pmpaddr registers (refer Register Summary) for any valid access to pass through in user-mode. However, it is not required for machine-mode as PMP permits all accesses to go through by deafult. In cases where PMP checks are also required in machine-mode, software can set the lock bit of required PMP entry to enable permission checks on it. Once lock bit is set, it can only be cleared through CPU reset.

When any instruction is being fetched from memory region without execute permissions, exception is generated at processor level and exception cause is set as instruction access fault in mcause CSR. Similarly, any load/store access without valid read/write permissions, will result in exception generation with mcause updated as load access and store access fault respectively. In case of load/store access faults, violating address is captured in mtval CSR.

9.8.4 Register Summary

Below is a list of PMP CSRs supported by the CPU. These are only accessible from machine-mode.

Name	Description	Address	Access
pmpcfg0	Physical memory protection configuration.	0x3A0	R/W
pmpcfg1	Physical memory protection configuration.	0x3A1	R/W
pmpcfg2	Physical memory protection configuration.	0x3A2	R/W
pmpcfg3	Physical memory protection configuration.	0x3A3	R/W
pmpaddr0	Physical memory protection address register.	0x3B0	R/W
pmpaddr1	Physical memory protection address register.	0x3B1	R/W
pmpaddr2	Physical memory protection address register.	0x3B2	R/W
pmpaddr3	Physical memory protection address register.	0x3B3	R/W
pmpaddr4	Physical memory protection address register.	0x3B4	R/W
pmpaddr5	Physical memory protection address register.	0x3B5	R/W
pmpaddr6	Physical memory protection address register.	0x3B6	R/W
pmpaddr7	Physical memory protection address register.	0x3B7	R/W
pmpaddr8	Physical memory protection address register.	0x3B8	R/W
pmpaddr9	Physical memory protection address register.	0x3B9	R/W
pmpaddr10	Physical memory protection address register.	0x3BA	R/W
pmpaddr11	Physical memory protection address register.	0x3BB	R/W
pmpaddr12	Physical memory protection address register.	0x3BC	R/W
pmpaddr13	Physical memory protection address register.	0x3BD	R/W
pmpaddr14	Physical memory protection address register.	0x3BE	R/W
pmpaddr15	Physical memory protection address register.	0x3BF	R/W

9.8.5 Register Description

PMP unit implements all pmpcfg0-3 and pmpaddr0-15 CSRs as defined in **RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 1.10**.

Glossary

Abbreviations for Peripherals

AES	AES (Advanced Encryption Standard) Accelerator
BOOTCTRL	Chip Boot Control
DS	Digital Signature
DMA	DMA (Direct Memory Access) Controller
eFuse	eFuse Controller
HMAC	HMAC (Hash-based Message Authentication Code) Accelerator
I2C	I2C (Inter-Integrated Circuit) Controller
I2S	I2S (Inter-IC Sound) Controller
LEDC	LED Control PWM (Pulse Width Modulation)
MCPWM	Motor Control PWM (Pulse Width Modulation)
PCNT	Pulse Count Controller
RMT	Remote Control Peripheral
RNG	Random Number Generator
RSA	RSA (Rivest Shamir Adleman) Accelerator
SDHOST	SD/MMC Host Controller
SHA	SHA (Secure Hash Algorithm) Accelerator
SPI	SPI (Serial Peripheral Interface) Controller
SYSTIMER	System Timer
TIMG	Timer Group
TWAI	Two-wire Automotive Interface
UART	UART (Universal Asynchronous Receiver-Transmitter) Controller
ULP Coprocessor	Ultra-low-power Coprocessor
USB OTG	USB On-The-Go
WDT	Watchdog Timers

Abbreviations for Registers

ISO	Isolation. When a module is power down, its output pins will be stuck in unknown state (some middle voltage). "ISO" registers will control to isolate its output pins to be a determined value, so it will not affect the status of other working modules which are not power down.
NMI	Non-maskable interrupt.
REG	Register.
R/W	Read/write. Software can read and write to these bits.
RO	Read-only. Software can only read these bits.
SYSREG	System Registers
WO	Write-only. Software can only write to these bits.

Revision History

Date	Version	Release notes
2021-04-08	V0.1	Preliminary release

www.espressif.com

Disclaimer and Copyright Notice