ESP32-S2

Technical Reference Manual

Preliminary V0.1

Espressif Systems

About This Manual

The **ESP32-S2 Technical Reference Manual** is addressed to application developers. The manual provides detailed and complete information on how to use the ESP32-S2 memory and peripherals.

For pin definition, electrical characteristics and package information, please see *ESP32-S2 Datasheet*.

Document Updates

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

Revision History

For any changes to this document over time, 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.

Disclaimer and Copyright Notice

Contents

1 System and Memory 13
1.1 Overview 13
1.2 Features 13
1.3 Functional Description 14
1.3.1 Address Mapping 14
1.3.2 Internal Memory 15
1.3.2.1 Internal ROM 0 16
1.3.2.2 Internal ROM 1 16
1.3.2.3 Internal SRAM 0 16
1.3.2.4 Internal SRAM 1 17
1.3.2.5 RTC FAST Memory 17
1.3.2.6 RTC SLOW Memory 17
1.3.3 External Memory 17
1.3.3.1 External Memory Address Mapping 17
1.3.3.2 Cache 18
1.3.3.3 Cache Operations 18
1.3.4 DMA 19
1.3.5 Modules / Peripherals 20
1.3.5.1 Naming Conventions for Peripheral Buses 20
1.3.5.2 Differences Between PeriBus1 and PeriBus2 20
1.3.5.3 Module / Peripheral Address Mapping 20
1.3.5.4 Addresses with Restricted Access from PeriBus1 22
2 Reset and Clock 23
2.1 Reset 23
2.1.1 Introduction 23
2.1.2 Reset Source 23
2.2 Clock 24
2.2.1 Introduction 24
2.2.2 Clock Source 25
2.2.3 CPU Clock 25
2.2.4 Peripheral Clock 27
2.2.4.1 APB_CLK Source 27
2.2.4.2 REF_TICK Source 27
2.2.4.3 LEDC_PWM_CLK Source 28
2.2.4.4 APLL_SCLK Source 28
2.2.4.5 PLL_160M_CLK Source 28
2.2.4.6 Clock Source Considerations 28
2.2.5 Wi-Fi Clock 29
2.2.6 RTC Clock 29
2.2.7 Audio PLL Clock 29
3 Chip Boot Control 30

3.1 Overview 30
3.2 Boot Mode 30
3.3 ROM Code Printing to UART 31
3.4 VDD_SPI Voltage 31

4 Interrupt Matrix 32
4.1 Overview 32
4.2 Features 32
4.3 Functional Description 32
4.3.1 Peripheral Interrupt Sources 32
4.3.2 CPU Interrupts 36
4.3.3 Allocate Peripheral Interrupt Source to CPU Interrupt 37
4.3.3.1 Allocate one peripheral interrupt source Source_X to CPU 37
4.3.3.2 Allocate multiple peripheral interrupt sources Source_X*n* to CPU 37
4.3.3.3 Disable CPU peripheral interrupt source Source_X 38
4.3.4 Disable CPU NMI Interrupt Sources 38
4.3.5 Query Current Interrupt Status of Peripheral Interrupt Source 38
4.4 Base Address 38
4.5 Register Summary 38
4.6 Registers 43

5 System Registers 80
5.1 Overview 80
5.2 Features 80
5.3 Function Description 80
5.3.1 System and Memory Registers 80
5.3.2 Reset and Clock Registers 82
5.3.3 Interrupt Matrix Registers 82
5.3.4 JTAG Software Enable Registers 82
5.3.5 Low-power Management Registers 83
5.3.6 Peripheral Clock Gating and Reset Registers 83
5.4 Base Address 85
5.5 Register Summary 85
5.6 Registers 86

6 LED PWM Controller 101
6.1 Overview 101
6.2 Features 101
6.3 Functional Description 101
6.3.1 Architecture 101
6.3.2 Timers 102
6.3.3 PWM Generators 103
6.3.4 Duty Cycle Fading 103
6.3.5 Interrupts 104
6.4 Base Address 104
6.5 Register Summary 105

6.6 Registers 107

7 Remote Control Peripheral 114

7.1 Introduction 114
7.2 Functional Description 114
7.2.1 RMT Architecture 114
7.2.2 RMT RAM 115
7.2.3 Clock 115
7.2.4 Transmitter 115
7.2.5 Receiver 116
7.2.6 Interrupts 117
7.3 Base Address 117
7.4 Register Summary 117
7.5 Registers 119

8 Pulse Count Controller 128

8.1 Features 128
8.2 Functional Description 129
8.3 Applications 131
8.3.1 Channel 0 Incrementing Independently 131
8.3.2 Channel 0 Decrementing Independently 132
8.3.3 Channel 0 and Channel 1 Incrementing Together 132
8.4 Base Address 133
8.5 Register Summary 133
8.6 Registers 135

9 64-bit Timers 141

9.1 Overview 141
9.2 Functional Description 142
9.2.1 16-bit Prescaler and Clock Selection 142
9.2.2 64-bit Time-based Counter 142
9.2.3 Alarm Generation 142
9.2.4 Timer Reload 142
9.2.5 Interrupts 143
9.3 Configuration and Usage 144
9.3.1 Timer as a Simple Clock 144
9.3.2 Timer as One-shot Alarm 144
9.3.3 Timer as Periodic Alarm 145
9.4 Base Address 145
9.5 Register Summary 145
9.6 Registers 147

10 Watchdog Timers 156

10.1 Overview 156
10.2 Features 156
10.3 Functional Description 156

10.3.1 Clock Source and 32-Bit Counter 156
10.3.2 Stages and Timeout Actions 157
10.3.3 Write Protection 157
10.3.4 Flash Boot Protection 158
10.4 Registers 158

11 eFuse Controller 159
11.1 Overview 159
11.2 Features 159
11.3 Functional Description 159
11.3.1 Structure 159
11.3.1.1 EFUSE_WR_DIS 163
11.3.1.2 EFUSE_RD_DIS 164
11.3.1.3 Data Storage 164
11.3.2 Software Programming of Parameters 164
11.3.3 Software Reading of Parameters 166
11.3.4 Timing 167
11.3.4.1 eFuse-Programming Timing 167
11.3.4.2 eFuse VDDQ Timing Setting 168
11.3.4.3 eFuse-Read Timing 168
11.3.5 The Use of Parameters by Hardware Modules 169
11.3.6 Interrupts 169
11.4 Base Address 170
11.5 Register Summary 170
11.6 Registers 173

12 I²C Controller 196
12.1 Overview 196
12.2 Features 196
12.3 I²C Functional Description 196
12.3.1 I²C Introduction 196
12.3.2 I²C Architecture 197
12.3.2.1 TX/RX RAM 198
12.3.2.2 CMD_Controller 198
12.3.2.3 SCL_FSM 200
12.3.2.4 SCL_MAIN_FSM 200
12.3.2.5 DATA_Shifter 200
12.3.2.6 SCL_Filter and SDA_Filter 200
12.3.3 I²C Bus Timing 200
12.4 Typical Applications 202
12.4.1 An I²C Master Writes to an I²C Slave with a 7-bit Address in One Command Sequence 202
12.4.2 An I²C Master Writes to an I²C Slave with a 10-bit Address in One Command Sequence 204
12.4.3 An I²C Master Writes to an I²C Slave with Two 7-bit Addresses in One Command Sequence 204
12.4.4 An I²C Master Writes to an I²C Slave with a 7-bit Address in Multiple Command Sequences 205
12.4.5 An I²C Master Reads an I²C Slave with a 7-bit Address in One Command Sequence 206
12.4.6 An I²C Master Reads an I²C Slave with a 10-bit Address in One Command Sequence 207

12.4.7 An I²C Master Reads an I²C Slave with Two 7-bit Addresses in One Command Sequence 207
12.4.8 An I²C Master Reads an I²C Slave with a 7-bit Address in Multiple Command Sequences 208
12.5 Clock Stretching 209
12.6 Interrupts 209
12.7 Base Address 210
12.8 Register Summary 210
12.9 Registers 212

13 AES Accelerator 234
13.1 Introduction 234
13.2 Features 234
13.3 Working Modes 234
13.4 Typical AES Working Mode 235
13.4.1 Key, Plaintext, and Ciphertext 236
13.4.2 Endianness 236
13.4.3 Operation Process 240
13.5 DMA-AES Working Mode 241
13.5.1 Key, Plaintext, and Cipertext 241
13.5.2 Endianness 242
13.5.3 Standard Incrementing Function 243
13.5.4 Block Number 243
13.5.5 Initialization Vector 243
13.5.6 Block Operation Process 243
13.5.7 GCM Operation Process 244
13.6 GCM Algorithm 246
13.6.1 Hash Subkey 247
13.6.2 J_0 247
13.6.3 Authenticated Tag 247
13.6.4 AAD Block Number 247
13.6.5 Remainder Bit Number 248
13.7 Base Address 248
13.8 Memory Summary 248
13.9 Register Summary 248
13.10 Registers 250

14 SHA Accelerator 255
14.1 Introduction 255
14.2 Features 255
14.3 Working Modes 255
14.4 Function Description 256
14.4.1 Preprocessing 256
14.4.1.1 Padding the Message 256
14.4.1.2 Parsing the Message 257
14.4.1.3 Initial Hash Value 258
14.4.2 Hash Computation Process 259
14.4.2.1 Typical SHA Process 259

14.4.2.2 DMA-SHA Process 261
14.4.3 Message Digest 263
14.4.4 Interrupt 263
14.5 Base Address 264
14.6 Register Summary 264
14.7 Registers 266

15 RSA Accelerator 270
15.1 Introduction 270
15.2 Features 270
15.3 Functional Description 270
15.3.1 Large Number Modular Exponentiation 271
15.3.2 Large Number Modular Multiplication 272
15.3.3 Large Number Multiplication 273
15.3.4 Acceleration Options 273
15.4 Base Address 275
15.5 Memory Summary 275
15.6 Register Summary 275
15.7 Registers 276

16 Random Number Generator 280
16.1 Introduction 280
16.2 Features 280
16.3 Functional Description 280
16.4 Base Address 281
16.5 Register Summary 281
16.6 Register 281

17 External Memory Encryption and Decryption 282
17.1 Overview 282
17.2 Features 282
17.3 Functional Description 282
17.3.1 XTS Algorithm 283
17.3.2 Key 283
17.3.3 Target Memory Space 284
17.3.4 Data Padding 284
17.3.5 Manual Encryption Block 285
17.3.6 Auto Encryption Block 286
17.3.7 Auto Decryption Block 287
17.4 Base Address 287
17.5 Register Summary 287
17.6 Registers 288

18 Digital Signature 292
18.1 Overview 292
18.2 Features 292

18.3 Functional Description 292
18.3.1 Overview 292
18.3.2 Private Key Operands 292
18.3.3 Conventions 293
18.3.4 Software Storage of Private Key Data 293
18.3.5 DS Operation at the Hardware Level 294
18.3.6 DS Operation at the Software Level 295
18.4 Base Address 296
18.5 Memory Blocks 296
18.6 Register Summary 296
18.7 Registers 297

Revision History 300

List of Tables

1-1 Address Mapping 15
1-2 Internal Memory Address Mapping 15
1-3 External Memory Address Mapping 18
1-4 Peripherals with DMA Support 19
1-5 Module / Peripheral Address Mapping 20
1-6 Addresses with Restricted Access 22
2-1 Reset Source 24
2-2 CPU_CLK Source 26
2-3 CPU_CLK Selection 26
2-4 Peripheral Clock Usage 27
2-5 APB_CLK Source 27
2-6 REF_TICK Source 28
2-7 LEDC_PWM_CLK Source 28
3-1 Default configuration of strapping pins 30
3-2 Boot Mode 30
3-3 ROM Code Printing Control 31
4-1 CPU peripheral interrupt configuration/status registers and peripheral interrupt sources 33
4-2 CPU Interrupts 36
4-3 Interrupt Matrix Base Address 38
5-1 ROM Controlling Bit 81
5-2 SRAM Controlling Bit 81
5-3 Peripheral Clock Gating and Reset Bits 83
5-4 System Register Base Address 85
6-1 LED_PWM Base Address 105
7-1 RMT Base Address 117
8-1 Counter Mode. Positive Edge of Input Pulse Signal. Control Signal in Low State 130
8-2 Counter Mode. Positive Edge of Input Pulse Signal. Control Signal in High State 130
8-3 Counter Mode. Negative Edge of Input Pulse Signal. Control Signal in Low State 130
8-4 Counter Mode. Negative Edge of Input Pulse Signal. Control Signal in High State 130
8-5 PCNT Base Address 133
9-1 64-bit Timers Base Address 145
11-1 Parameters in BLOCK0 159
11-2 Key Purpose Values 162
11-3 Parameters in BLOCK1-10 162
11-5 Configuration of eFuse-Programming Timing Parameters 167
11-6 Configuration of VDDQ Timing Parameters 168
11-7 Configuration of eFuse-Reading Parameters 168
11-8 eFuse Controller Base Address 170
12-1 I²C Controller Base Address 210
13-1 AES Accelerator Working Mode 235
13-2 Operation Type under Typical AES Working Mode 235
13-3 Working Status under Typical AES Working Mode 235
13-4 Text Endianness Types for Typical AES 236
13-5 Key Endianness Types for AES-128 Encryption and Decryption 238

13-6 Key Endianness Types for AES-192 Encryption and Decryption 238
13-7 Key Endianness Types for AES-256 Encryption and Decryption 239
13-8 Operation Type under DMA-AES Working Mode 241
13-9 Working Status under DMA-AES Working mode 241
13-10TEXT-PADDING 242
13-11Text Endianness for DMA-AES 242
13-12AES Accelerator Base Address 248
14-1 SHA Accelerator Working Mode 256
14-2 SHA Hash Algorithm 256
14-6 The Storage and Length of Message Digests from Different Algorithms 263
14-7 Base Address 264
15-1 Acceleration Performace 274
15-2 RSA Base Address 275
16-1 Random Number Generator Base Address 281
17-1 Key 284
17-2 Mapping Between Offsets and Registers 285
17-3 Manual Encryption Block Base Address 287
18-1 Base Address 296

List of Figures

1-1 System Structure and Address Mapping 14
1-2 Cache Structure 18
2-1 System Reset 23
2-2 System Clock 25
4-1 Interrupt Matrix Structure 32
6-1 LED_PWM Architecture 101
6-2 LED_PWM generator Diagram 102
6-3 LED_PWM Divider 102
6-4 LED_PWM Output Signal Diagram 103
6-5 Output Signal Diagram of Fading Duty Cycle 104
7-1 RMT Architecture 114
7-2 Format of Pulse Code in RAM 115
8-1 PCNT Block Diagram 128
8-2 PCNT Unit Architecture 129
8-3 Channel 0 Up Counting Diagram 131
8-4 Channel 0 Down Counting Diagram 132
8-5 Two Channels Up Counting Diagram 132
9-1 Timer Units within Groups 141
11-1 Shift Register Circuit 164
11-2 eFuse-Programming Timing Diagram 169
11-3 Timing Diagram for Reading eFuse 169
12-1 I²C Master Architecture 197
12-2 I²C Slave Architecture 197
12-3 Structure of I²C Command Register 198
12-4 I²C Timing Diagram 201
12-5 An I²C Master Writing to an I²C Slave with a 7-bit Address 202
12-6 A Master Writing to a Slave with a 10-bit Address 204
12-7 An I²C Master Writing Address M in the RAM to an I²C Slave with a 7-bit Address 204
12-8 An I²C Master Writing to an I²C Slave with a 7-bit Address in Multiple Sequences 205
12-9 An I²C Master Reading an I²C Slave with a 7-bit Address 206
12-10 An I²C Master Reading an I²C Slave with a 10-bit Address 207
12-11 An I²C Master Reading N Bytes of Data from addrM of an I²C Slave with a 7-bit Address 207
12-12 An I²C Master Reading an I²C Slave with a 7-bit Address in Segments 208
13-1 GCM Encryption Process 246
16-1 Noise Source 280
17-1 Architecture of the External Memory Encryption and Decryption Module 282
18-1 Preparations and DS Operation 293

1. System and Memory

1.1 Overview

The ESP32-S2 is a single-core system with one Harvard Architecture Xtensa® LX7 CPU. All internal memory, external memory, and peripherals are located on the CPU buses.

1.2 Features

- **Address Space**
  - 4 GB (32 bits wide) address space in total accessed from the data bus and instruction bus
  - 464 KB internal memory address space accessed from the instruction bus
  - 400 KB internal memory address space accessed from the data bus
  - 1.77 MB peripheral address space
  - 7.5 MB external memory virtual address space accessed from the instruction bus
  - 14.5 MB external memory virtual address space accessed from the data bus
  - 320 KB internal DMA address space
  - 10.5 MB external DMA address space
- **Internal Memory**
  - 128 KB Internal ROM
  - 320 KB Internal SRAM
  - 8 KB RTC FAST Memory
  - 8 KB RTC SLOW Memory
- **External Memory**
  - Supports up to 1 GB external SPI flash
  - Supports up to 1 GB external SPI RAM
- **DMA**
  - 9 DMA-supported modules / peripherals

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

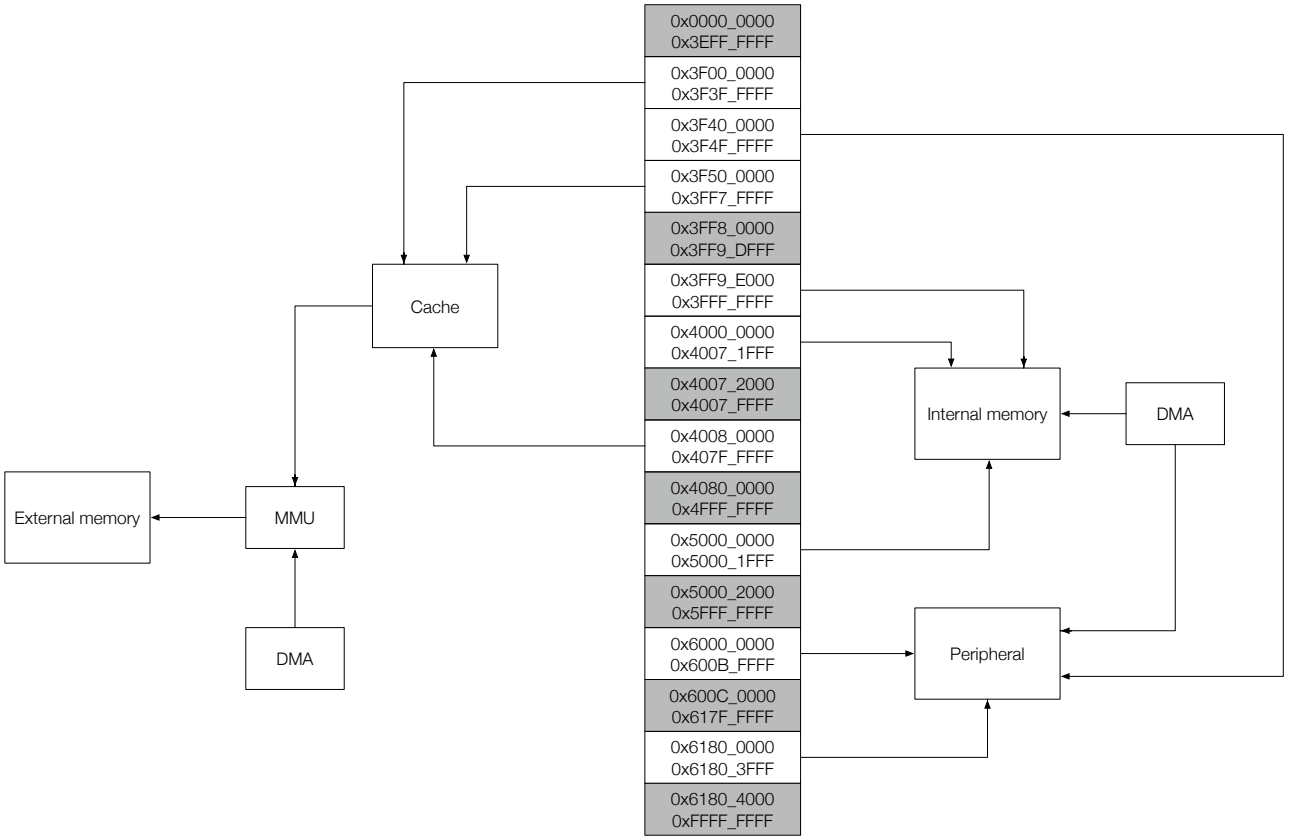


Figure 1-1. System Structure and Address Mapping

Note:

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

1.3 Functional Description

1.3.1 Address Mapping

The Harvard Architecture Xtensa® LX7 CPU can address 4 GB (32 bits wide) memory space.

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

Both data bus and instruction bus are little-endian. The CPU can access data via the data bus in a byte-, half-word-, or word-aligned matter. The CPU can also access data via the instruction bus, but only in a word-aligned manner; non-word-aligned access will cause a CPU exception.

The CPU can:

- directly access the internal memory via both data bus and instruction bus;

- access the external memory which is mapped into the address space via cache;
- access modules / peripherals via data bus.

Table 1-1 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 same memory is available to the CPU at two address ranges.

Table 1-1. Address Mapping

Bus Type	Boundary Address		Size	Target
	Low Address	High Address		
	0x0000_0000	0x3EFF_FFFF		Reserved
Data bus	0x3F00_0000	0x3F3F_FFFF	4 MB	External memory
	0x3F40_0000	0x3F4F_FFFF	1 MB	Peripherals
	0x3F50_0000	0x3FF7_FFFF	10.5 MB	External memory
	0x3FF8_0000	0x3FF9_DFFF		Reserved
Data bus	0x3FF9_E000	0x3FFF_FFFF	392 KB	Internal memory
Instruction bus	0x4000_0000	0x4007_1FFF	456 KB	Internal memory
	0x4007_2000	0x4007_FFFF		Reserved
Instruction bus	0x4008_0000	0x407F_FFFF	7.5 MB	External memory
	0x4080_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	0x600B_FFFF	768 KB	Peripherals
	0x600C_0000	0x617F_FFFF		Reserved
Data / Instruction bus	0x6180_0000	0x6180_3FFF	16 KB	Peripherals
	0x6180_4000	0xFFFF_FFFF		Reserved

1.3.2 Internal Memory

The internal memory consists of four segments: Internal ROM (128 KB), Internal SRAM (320 KB), RTC FAST Memory (8 KB), and RTC SLOW Memory (8 KB).

The Internal ROM is broken down into two parts: Internal ROM 0 (64 KB) and Internal ROM 1 (64 KB).

The Internal SRAM is broken down into two parts: Internal SRAM 0 (32 KB) and Internal SRAM 1 (288 KB).

RTC FAST Memory and RTC SLOW Memory are both implemented as SRAM.

Table 1-2 lists all types of internal memory and their address ranges on the data bus and instruction bus.

Table 1-2. Internal Memory Address Mapping

Bus Type	Boundary Address		Size	Target	Permission Control
	Low Address	High Address			
Data bus	0x3FF9_E000	0x3FF9_FFFF	8 KB	RTC FAST Memory	YES
	0x3FFA_0000	0x3FFA_FFFF	64 KB	Internal ROM 1	NO
	0x3FFB_0000	0x3FFB_7FFF	32 KB	Internal SRAM 0	YES

	0x3FFB_8000	0x3FFF_FFFF	288 KB	Internal SRAM 1	YES
Bus Type	Boundary Address		Size	Target	Permission Control
	Low Address	High Address			
Instruction bus	0x4000_0000	0x4000_FFFF	64 KB	Internal ROM 0	NO
	0x4001_0000	0x4001_FFFF	64 KB	Internal ROM 1	NO
	0x4002_0000	0x4002_7FFF	32 KB	Internal SRAM 0	YES
	0x4002_8000	0x4006_FFFF	288 KB	Internal SRAM 1	YES
	0x4007_0000	0x4007_1FFF	8 KB	RTC FAST Memory	YES
Bus Type	Boundary Address		Size	Target	Permission Control
	Low Address	High Address			
Data / Instruction bus	0x5000_0000	0x5000_1FFF	8 KB	RTC SLOW Memory	YES

Note:

"YES" in the "Permission Control" column indicates that a permission is required for memory access. Permission Control registers can be used to limit Instruction or Data bus access to individual regions of these memory types.

1.3.2.1 Internal ROM 0

Internal ROM 0 is a 64-KB, read-only memory space, addressed by the CPU on the instruction bus via range(s) described in Table 1-2.

1.3.2.2 Internal ROM 1

Internal ROM 1 is a 64-KB, read-only memory space, addressed by the CPU on the data or instruction bus via range(s) described in Table 1-2.

The two address ranges access Internal ROM 1 in the same order, so, for example, addresses 0x3FFA_0000 and 0x4001_0000 access the same word, 0x3FFA_0004 and 0x4001_0004 access the same word, 0x3FFA_0008 and 0x4001_0008 access the same word, etc.

1.3.2.3 Internal SRAM 0

Internal SRAM 0 is a 32-KB, read-and-write memory space, addressed by the CPU on the data or instruction bus, in the same order, via range(s) described in Table 1-2.

Hardware can be configured to use 8 KB, 16 KB, 24 KB, or the entire 32 KB space in this memory to cache external memory. The space used as cache cannot be accessed by the CPU, while the remaining space can still be accessed by the CPU.

1.3.2.4 Internal SRAM 1

Internal SRAM 1 is a 288-KB, read-and-write memory space, addressed by the CPU on the data or instruction bus, in the same order, via range(s) described in Table 1-2.

Internal SRAM 1 comprises eighteen 16-KB (sub)memory blocks. One block can be used as Trace Memory, in which case this block's address range cannot be accessed by the CPU.

1.3.2.5 RTC FAST Memory

RTC FAST Memory is an 8-KB, read-and-write SRAM, addressed by the CPU on the data or instruction bus, in the same order, via range(s) described in Table 1-2.

1.3.2.6 RTC SLOW Memory

RTC SLOW Memory is an 8-KB, read-and-write SRAM, addressed by the CPU via range(s) shared by the data bus and the instruction bus, as described in Table 1-2.

RTC SLOW Memory can also be used as a peripheral addressable to the CPU via either 0x3F42_1000 ~ 0x3F42_2FFF or 0x6002_1000 ~ 0x6002_2FFF on the data bus.

1.3.3 External Memory

ESP32-S2 supports multiple QSPI/OSPI flash and RAM chips. It also supports hardware encryption/decryption based on XTS-AES to protect user programs and data in the flash and external RAM.

1.3.3.1 External Memory Address Mapping

The CPU accesses the external flash and RAM via the cache. According to the MMU settings, the cache maps the CPU's address to the external physical memory address. Due to this address mapping, the ESP32-S2 can address up to 1 GB external flash and 1 GB external RAM.

Using the cache, ESP32-S2 can support the following address space mappings at the same time.

- Up to 7.5 MB instruction bus address space can be mapped into the external flash or RAM as individual 64 KB blocks, via the instruction cache (ICache). Byte (8-bit), half-word (16-bit) and word (32-bit) reads are supported.
- Up to 4 MB read-only data bus address space can be mapped into the external flash or RAM as individual 64 KB blocks, via ICache. Byte (8-bit), half-word (16-bit) and word (32-bit) reads are supported.
- Up to 10.5 MB data bus address space can be mapped into the external RAM as individual 64 KB blocks, via DCache. Byte (8-bit), half-word (16-bit) or word (32-bit) reads and writes are supported. Blocks from this 10.5 MB space can also be mapped into the external flash or RAM, for read operations only.

Table 1-3 lists the mapping between the cache and the corresponding address ranges on the data bus and instruction bus.

Table 1-3. External Memory Address Mapping

Bus Type	Boundary Address		Size	Target	Permission Control
	Low Address	High Address			
Data bus	0x3F00_0000	0x3F3F_FFFF	4 MB	ICache	YES
Data bus	0x3F50_0000	0x3FF7_FFFF	10.5 MB	DCache	YES
Instruction bus	0x4008_0000	0x407F_FFFF	7.5 MB	ICache	YES

Note:

"YES" in the "Permission Control" column indicates that a permission is required for memory access. Permission Control registers can be used to limit Instruction or Data bus access to individual regions of these memory types.

1.3.3.2 Cache

As shown in Figure 1-2, the caches on ESP32-S2 are separated and allow prompt response upon simultaneous requests from the data bus and instruction bus. Some internal memory space can be used as cache (see Section 1.3.2.3). When a cache miss occurs, the cache controller will initiate a request to the external memory. When ICache and DCache simultaneously initiate a request, the arbiter determines which gets the access to the external memory first. The cache size of ICache and DCache can be configured to 8 KB and 16 KB, respectively, while their block size can be configured to 16 bytes and 32 bytes, respectively.

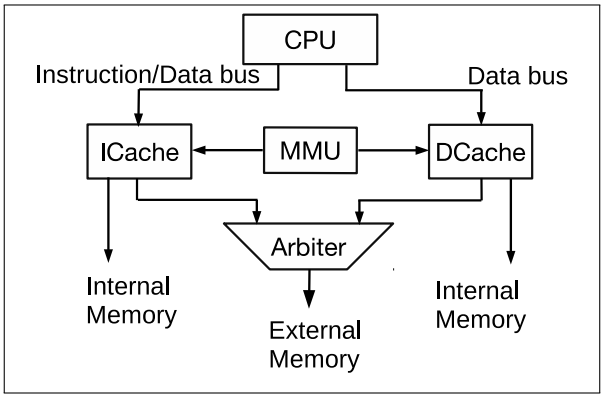


Figure 1-2. Cache Structure

1.3.3.3 Cache Operations

ESP32-S2 caches support the following operations:

1. **Invalidate**: The cache clears the valid bit of a tag. The CPU needs to access the external memory in order to read/write the data. There are two types of invalidation operations: manual invalidation and automatic invalidation. Manual invalidation performs only on data in the specified area in the cache, while automatic invalidation performs on all data in the cache. Both ICache and DCache have this function.
2. **Clean**: The cache clears the dirty bit of the tag and retains the valid bit. The CPU can then read/write the data directly from the cache. Only DCache has this function.

3. **Write-back**: The cache clears the dirty block flag of the tag and retains the valid bit. It also forces the data in the corresponding address to be written back to the external memory. The CPU can then read/write the data directly from the cache. Only DCache has this function.
4. **Preload**: To preload a cache is to load instructions and data into the cache in advance. The minimum unit of a preloading is one block. There are two types of preloading: manual preloading and automatic preloading. Manual preloading means that the hardware prefetches a piece of continuous data according to the virtual address specified by the software. Automatic preloading means the hardware prefetches a piece of continuous data according to the current hit / miss address (depending on configuration).
5. **Lock / Unlock**: 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. When manual lock is enabled, the cache checks the data that has been filled into the cache memory and locks the data that falls in the specified area. The data in the locked area is always stored in the cache and will not be replaced. But when all the ways within the cache are locked, the cache will replace data, as if the ways were not locked. Unlocking is the reverse of locking. The manual invalidation, clean, and write-back operations are only available after unlocking.

1.3.4 DMA

With DMA, the ESP32-S2 can perform data transfers between:

- modules / peripherals and internal memory;
- different types of internal memory;
- modules / peripherals and external memory;
- internal and external memory.

DMA uses the same addressing as the data bus to read and write Internal SRAM 0 and Internal SRAM 1. Specifically, DMA uses address range 0x3FFB_0000 ~ 0x3FFB_7FFF to access Internal SRAM 0, and 0x3FFB_8000 ~ 0x3FFF_FFFF to access Internal SRAM 1. Note that DMA cannot access the internal memory occupied by the cache.

In addition, DMA addresses the external RAM from 0x3F50_0000 ~ 0x3FF7_FFFF, the same used by the CPU to access DCache. When DCache and DMA access the external memory simultaneously, data consistency is required.

Nine modules / peripherals on the ESP32-S2 support DMA, as shown in Table 1-4. With DMA, some of them can only access internal memory, some can access both internal and external memory.

Table 1-4. Peripherals with DMA Support

UART0	UART1
SPI2	SPI3
I2S0	
ADC Controller	
Copy DMA	
AES Accelerator	SHA Accelerator

1.3.5 Modules / Peripherals

The CPU can access modules / peripherals via address range 0x3F40_0000 ~ 0x3F4F_FFFF on the data bus, or via 0x6000_0000 ~ 0x600B_FFFF and 0x6180_0000 ~ 0x6180_3FFF shared by the data bus and instruction bus.

1.3.5.1 Naming Conventions for Peripheral Buses

There are two peripheral buses defined as follows:

- **PeriBus1**: Which refers to the address range 0x3F40_0000 ~ 0x3F4F_FFFF on the bus. 0x3F40_0000 is the base address.
- **PeriBus2**: Which refers to the address ranges 0x6000_0000 ~ 0x600B_FFFF and 0x6180_0000 ~ 0x6180_3FFF on the bus. 0x6000_0000 is the base address.

All references to "PeriBus1" and "PeriBus2" in this document indicate the corresponding address range(s).

1.3.5.2 Differences Between PeriBus1 and PeriBus2

The CPU can access modules / peripherals more efficiently through PeriBus1 than through PeriBus2. However, PeriBus1 features speculative reads, which means it cannot guarantee that each read is valid. Therefore, the CPU has to use PeriBus2 to access some special registers, for example, FIFO registers.

In addition, PeriBus1 will upset the order of r/w operations on the bus to improve performance, which may cause programs that have strict requirements on the r/w order to crash. In such cases, please add volatile before the program statement, or use PeriBus2 instead.

1.3.5.3 Module / Peripheral Address Mapping

Table 1-5 lists all the modules / peripherals and their respective address ranges. Note that addresses in column "Boundary Address" are offsets relative to the base address, instead of absolute addresses. The absolute addresses are the addition of bus base address and the corresponding offsets.

Table 1-5. Module / Peripheral Address Mapping

Target	Boundary Address		Size	Notes
	Low Address	High Address		
UART0	0x0000_0000	0x0000_0FFF	4 KB	1, 2, 3
Reserved	0x0000_1000	0x0000_1FFF		
SPI1	0x0000_2000	0x0000_2FFF	4 KB	1, 2
SPI0	0x0000_3000	0x0000_3FFF	4 KB	1, 2
GPIO	0x0000_4000	0x0000_4FFF	4 KB	1, 2
Reserved	0x0000_5000	0x0000_6FFF		
TIMER	0x0000_7000	0x0000_7FFF	4 KB	1, 2
RTC	0x0000_8000	0x0000_8FFF	4 KB	1, 2
IO MUX	0x0000_9000	0x0000_9FFF	4 KB	1, 2
Reserved	0x0000_A000	0x0000_EFFF		

Target	Boundary Address		Size	Notes
	Low Address	High Address		
I2S0	0x0000_F000	0x0000_FFFF	4 KB	1, 2, 3
UART1	0x0001_0000	0x0001_0FFF	4 KB	1, 2, 3
Reserved	0x0001_1000	0x0001_2FFF		
I2C0	0x0001_3000	0x0001_3FFF	4 KB	1, 2, 3
UHCI0	0x0001_4000	0x0001_4FFF	4 KB	1, 2
Reserved	0x0001_5000	0x0001_5FFF		
RMT	0x0001_6000	0x0001_6FFF	4 KB	1, 2, 3
PCNT	0x0001_7000	0x0001_7FFF	4 KB	1, 2
Reserved	0x0001_8000	0x0001_8FFF		
LED PWM Controller	0x0001_9000	0x0001_9FFF	4 KB	1, 2
eFuse Controller	0x0001_A000	0x0001_AFFF	4 KB	1, 2
Reserved	0x0001_B000	0x0001_EFFF		
Timer Group 0	0x0001_F000	0x0001_FFFF	4 KB	1, 2
Timer Group 1	0x0002_0000	0x0002_0FFF	4 KB	1, 2
RTC SLOW Memory	0x0002_1000	0x0002_2FFF	8 KB	1, 2, 3
System Timer	0x0002_3000	0x0002_3FFF	4 KB	1, 2
SPI2	0x0002_4000	0x0002_4FFF	4 KB	1, 2
SPI3	0x0002_5000	0x0002_5FFF	4 KB	1, 2
APB Controller	0x0002_6000	0x0002_6FFF	4 KB	1, 2
I2C1	0x0002_7000	0x0002_7FFF	4 KB	1, 2, 3
Reserved	0x0002_8000	0x0002_AFFF		
TWAI Controller	0x0002_B000	0x0002_BFFF	4 KB	1, 2
Reserved	0x0002_C000	0x0003_8FFF		
USB OTG	0x0003_9000	0x0003_9FFF	4 KB	1, 2, 3, 4
AES Accelerator	0x0003_A000	0x0003_AFFF	4 KB	1, 2
SHA Accelerator	0x0003_B000	0x0003_BFFF	4 KB	1, 2
RSA Accelerator	0x0003_C000	0x0003_CFFF	4 KB	1, 2
Digital Signature	0x0003_D000	0x0003_DFFF	4 KB	1, 2
HMAC	0x0003_E000	0x0003_EFFF	4 KB	1, 2
Crypto DMA	0x0003_F000	0x0003_FFFF	4 KB	1, 2
Reserved	0x0004_4000	0x000C_DFFF		
ADC Controller	0x0004_0000	0x0004_0FFF	4 KB	1, 2
Reserved	0x0004_1000	0x0007_FFFF		
USB OTG	0x0008_0000	0x000B_FFFF	256 KB	1, 2, 3, 4
System Registers	0x000C_0000	0x000C_0FFF	4 KB	1
Sensitive Register	0x000C_1000	0x000C_1FFF	4 KB	1
Interrupt Matrix	0x000C_2000	0x000C_2FFF	4 KB	1
Copy DMA	0x000C_3000	0x000C_3FFF	4 KB	1
Reserved	0x000C_4000	0x000C_EFFF		
Dedicated GPIO	0x000C_F000	0x000C_FFFF	4 KB	1
Reserved	0x000D_1000	0x000F_FFFF		
Configure Cache	0x0180_0000	0x0180_3FFF	16 KB	2

Note:

1. This module / peripheral can be accessed from PeriBus1.
2. This module / peripheral can be accessed from PeriBus2.
3. Some special addresses in this module / peripheral are not accessible from PeriBus1 (see Section 1.3.5.4).
4. The address space in this module / peripheral is not continuous.

1.3.5.4 Addresses with Restricted Access from PeriBus1

As mentioned in Section 1.3.5.2, PeriBus1 features speculative reads, which means it is forbidden to read FIFO registers. Table 1-6 below lists the address (range) with restricted access from PeriBus1.

There are four reserved user-defined registers that can be configured as needed to add more addresses with restricted access.

Table 1-6. Addresses with Restricted Access

Peripherals	Addresses with Restricted Access
UART0	0x3F40_0000
UART1	0x3F41_0000
I2S0	0x3F40_F004
RMT	0x3F41_6000 ~ 0x3F41_600F
I2C0	0x3F41_301C
I2C1	0x3F42_701C
USB OTG	0x3F48_0020, 0x3F48_1000 ~ 0x3F49_0FFF

2. Reset and Clock

2.1 Reset

2.1.1 Introduction

ESP32-S2 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 2-1 shows the scopes of affected subsystems when different types of reset occur.

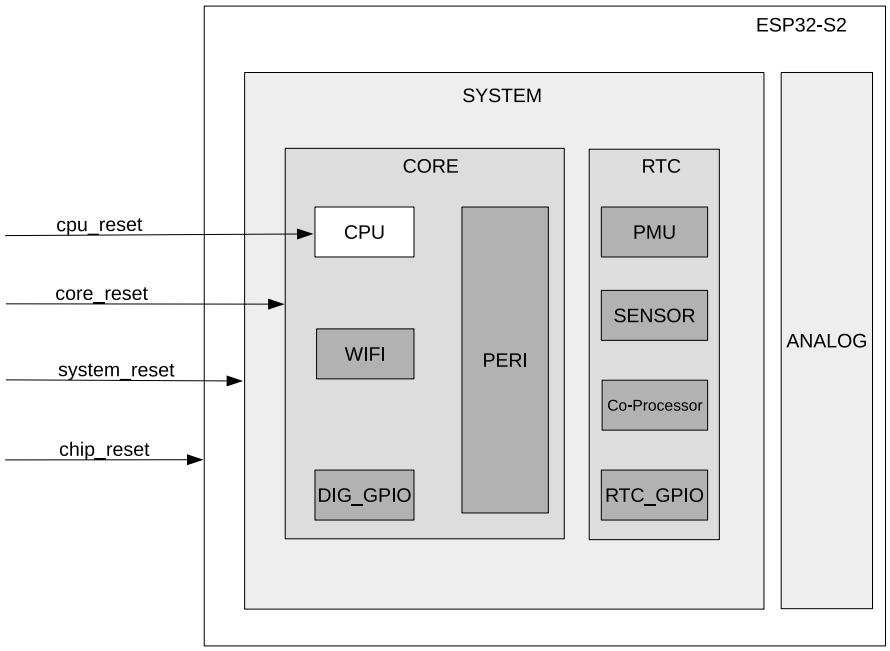


Figure 2-1. System Reset

- CPU Reset: Only resets CPU core. Once such reset is released, programs will be executed from CPU reset vector.
- Core Reset: Resets the whole digital system except RTC, including CPU, peripherals, Wi-Fi, and digital GPIOs.
- System Reset: Resets the whole digital system, including RTC.
- Chip Reset: Resets the whole chip.

2.1.2 Reset Source

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 2-1 lists different reset sources and the types of reset they trigger.

Table 2-1. Reset Source

Code	Source	Reset Type	Comments
0x01	Chip reset	Chip Reset	See the note below
0x0F	Brown-out system reset	System Reset	Triggered by brown-out detector
0x10	RWDT system reset	System Reset	See Chapter 10 *Watchdog Timers*
0x13	GLITCH reset	System Reset	-
0x03	Software system reset	Core Reset	Triggered by configuring RTC_CNTL_SW_SYS_RST
0x05	Deep-sleep reset	Core Reset	-
0x07	MWDT0 global reset	Core Reset	See Chapter 10 *Watchdog Timers*
0x08	MWDT1 global reset	Core Reset	See Chapter 10 *Watchdog Timers*
0x09	RWDT core reset	Core Reset	See Chapter 10 *Watchdog Timers*
0x0B	MWDT0 CPU reset	CPU Reset	See Chapter 10 *Watchdog Timers*
0x0C	Software CPU reset	CPU Reset	Triggered by configuring RTC_CNTL_SW_PROCPU_RST
0x0D	RWDT CPU reset	CPU Reset	See Chapter 10 *Watchdog Timers*
0x11	MWDT1 CPU reset	CPU Reset	See Chapter 10 *Watchdog Timers*

Note:

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

2.2 Clock

2.2.1 Introduction

ESP32-S2 provides multiple clock sources, which allow CPU, peripherals and RTC to work at different frequencies, thus providing more flexibility in meeting the requirements of various application scenarios. Figure 2-2 shows the system clock structure.

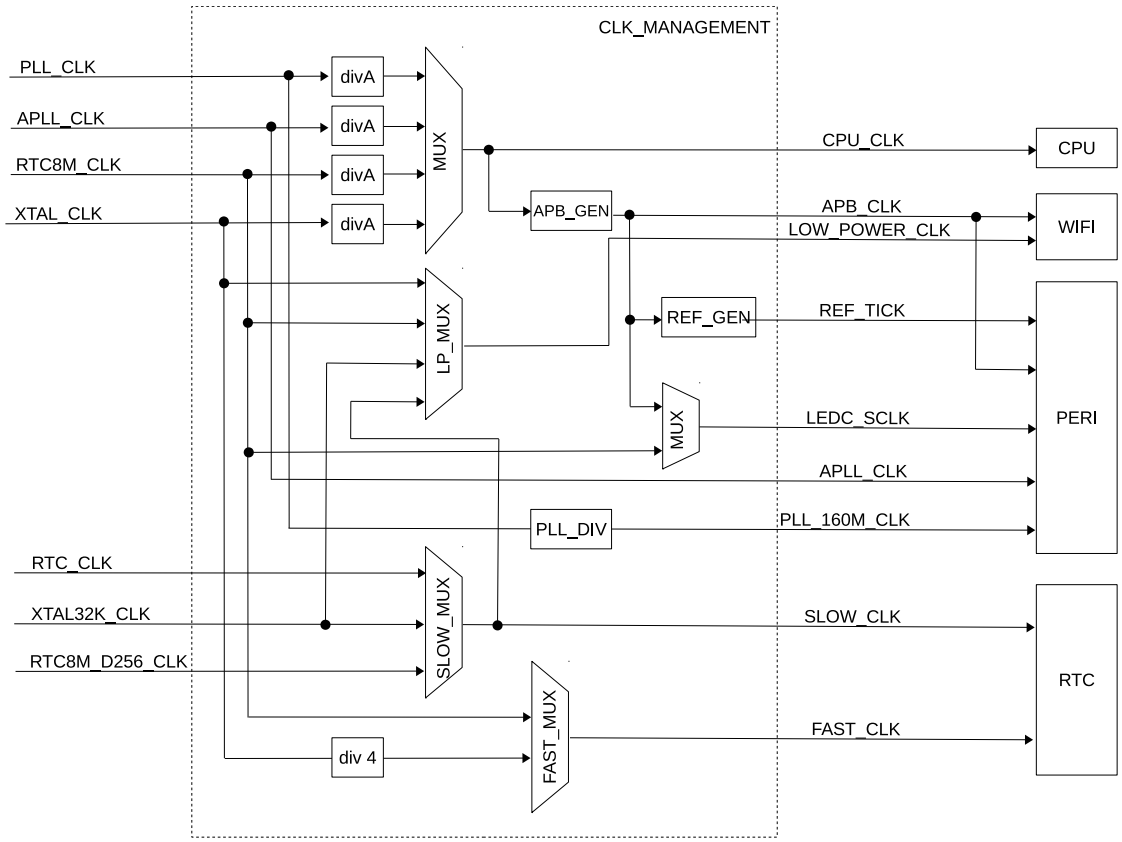


Figure 2-2. System Clock

2.2.2 Clock Source

ESP32-S2 uses external crystal, internal PLL, or internal oscillator working as clock sources to generate different kinds of clocks, which can be classified in three types depending on their clock speed.

- High speed clock 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 clock for low-power devices, such as power management unit and low-power peripherals
  - XTAL32K_CLK (32 kHz): external crystal clock
  - RTC8M_CLK (8 MHz by default): internal oscillator with adjustable frequency
  - RTC8M_D256_CLK (31.250 kHz by default): internal clock derived from RTC8M_CLK divided by 256
  - RTC_CLK (150 kHz by default): internal oscillator with adjustable frequency
- Audio Clock for audio-related devices
  - APLL_CLK (16 MHz ~ 128 MHz): internal Audio PLL clock

2.2.3 CPU Clock

As Figure 2-2 shows, CPU_CLK is the master clock for CPU and it can be as high as 240 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, APLL_CLK, RTC8M_CLK or XTAL_CLK as CPU_CLK clock source by configuring register SYSTEM_SOC_CLK_SEL, see Table 2-2 and Table 2-3.

Table 2-2. CPU_CLK Source

SYSTEM_SOC_CLK_SEL Value	Clock Source
0	XTAL_CLK
1	PLL_CLK
2	RTC8M_CLK
3	APLL_CLK

Table 2-3. CPU_CLK Selection

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 (480 MHz)	1	1	2	CPU_CLK = PLL_CLK / 2 CPU_CLK frequency is 240 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
RTC8M_CLK	2	-	-	CPU_CLK = RTC8M_CLK / (SYSTEM_PRE_DIV_CNT + 1) SYSTEM_PRE_DIV_CNT ranges from 0 ~ 1023. Default is 1
APLL_CLK	3	0	0	CPU_CLK = APLL_CLK / 4
APLL_CLK	3	0	1	CPU_CLK = APLL_CLK / 2

*SEL_0: The value of register SYSTEM_SOC_CLK_SEL.

*SEL_1: The value of register SYSTEM_PLL_FREQ_SEL.

*SEL_2: The value of register SYSTEM_CPUPERIOD_SEL.

Note:

- When users select XTAL_CLK as CPU clock source and adjust the divider value by configuring register SYSTEM_PRE_DIV_CNT, the rules below should be followed.
  - If current divider value is 2 (SYSTEM_PRE_DIV_CNT = 1) and the target is x (x ≠ 1), users should set the divider first to 1 (SYSTEM_PRE_DIV_CNT = 0) and then to x (SYSTEM_PRE_DIV_CNT = x - 1).
  - If current divider value is x (SYSTEM_PRE_DIV_CNT = x - 1) and the target is 2, users should set the divider first to 1 (SYSTEM_PRE_DIV_CNT = 0) and then to 2 (SYSTEM_PRE_DIV_CNT = 1).
  - For other target divider value x, users can adjust the register directly (SYSTEM_PRE_DIV_CNT = x - 1).

2.2.4 Peripheral Clock

Peripheral clocks include APB_CLK, REF_TICK, LEDC_PWM_CLK, APLL_CLK and PLL_160M_CLK. Table 2-4 shows which clock can be used by which peripheral.

Table 2-4. Peripheral Clock Usage

Peripheral	APB_CLK	REF_TICK	LEDC_PWM_CLK	APLL_CLK	PLL_160M_CLK
TIMG	Y	Y			
I²S	Y			Y	Y
UHCI	Y				
UART	Y	Y			
RMT	Y	Y			
LED_PWM	Y	Y	Y		
I²C	Y	Y			
SPI	Y				
PCNT	Y				
eFuse Controller	Y				
SARADC/DAC	Y			Y	
USB	Y				
CRYPTO					Y
TWAI Controller	Y				
System Timer	Y				

2.2.4.1 APB_CLK Source

APB_CLK is determined by the clock source of CPU_CLK as shown in Table 2-5.

Table 2-5. APB_CLK Source

CPU_CLK Source	APB_CLK
PLL_CLK	80 MHz
APLL_CLK	CPU_CLK / 2
XTAL_CLK	CPU_CLK
RTC8M_CLK	CPU_CLK

2.2.4.2 REF_TICK Source

REF_TICK is derived from XTAL_CLK or RTC8M_CLK via a divider. When PLL_CLK, APLL_CLK or XTAL_CLK is set as CPU clock source, REF_TICK will be divided from XTAL_CLK. When RTC8M_CLK is set as CPU clock source, REF_TICK will be divided from RTC8M_CLK. In such way, REF_TICK frequency remains unchanged when APB_CLK changes its clock source. Table 2-6 shows the configuration of these clock divider registers.

Table 2-6. REF_TICK Source

CPU_CLK Source	Clock Divider Register
PLL_CLK | XTAL_CLK | APLL_CLK	APB_CTRL_XTAL_TICK_NUM
RTC8M_CLK	APB_CTRL_CK8M_TICK_NUM

Normally, one REF_TICK cycle lasts for 1 μs, so APB_CTRL_XTAL_TICK_NUM should be configured to 39 (default), and APB_CTRL_CK8M_TICK_NUM to 7 (default).

2.2.4.3 LEDC_PWM_CLK Source

LEDC_PWM_CLK clock source is selected by configuring register LEDC_APB_CLK_SEL, as shown in Table 2-7.

Table 2-7. LEDC_PWM_CLK Source

LEDC_APB_CLK_SEL Value	LEDC_PWM_CLK Source
0 (Default)	-
1	APB_CLK
2	RTC8M_CLK
3	XTAL_CLK

2.2.4.4 APLL_SCLK Source

APLL_CLK is sourced from PLL_CLK, and its output frequency is configured using APLL configuration registers. See Section 2.2.7 for more information.

2.2.4.5 PLL_160M_CLK Source

PLL_160M_CLK is divided from PLL_CLK according to current PLL frequency.

2.2.4.6 Clock Source Considerations

Peripherals that need to work with other clocks, such as RMT and I²C, generally operate using PLL_CLK frequency as a reference. When this frequency changes, peripherals should update their clock configuration to operate at the same frequency after the change. Peripherals accessing REF_TICK can continue operating normally without changing their clock configuration when switching clock sources. Please see Table 2-4.

LED module uses RTC8M_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 (APB_CLK is turned off), but LED can work normally via RTC8M_CLK.

2.2.5 Wi-Fi Clock

Wi-Fi can work only when APB_CLK uses PLL_CLK as its clock source. Suspending PLL_CLK requires that Wi-Fi has entered low-power mode first.

LOW_POWER_CLK uses XTAL32K_CLK, XTAL_CLK, RTC8M_CLK or SLOW_CLK (the low clock selected by RTC) as its clock source for Wi-Fi in low-power mode.

2.2.6 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 RTC8M_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 RTC8M_CLK.

2.2.7 Audio PLL Clock

The operation of audio and other time-critical data-transfer applications requires highly-configurable, low-jitter and accurate clock sources. The clock sources derived from system clocks that serve digital peripherals may carry jitter and, therefore, are not suitable for a high-precision clock frequency setting.

Providing an integrated precision clock source can minimize system cost. To this end, ESP32-S2 integrates an audio PLL to clock I²S module.

Audio PLL formula is as follows:

$$f_{\text{out}} = \frac{f_{\text{xtal}}(\text{sdm2} + \frac{\text{sdm1}}{2^8} + \frac{\text{sdm0}}{2^{16}} + 4)}{2(odiv + 2)}$$

Parameters are defined below:

- f_{xtal}: the frequency of crystal oscillator, usually 40 MHz;
- sdm0: the value is 0 ~ 255;
- sdm1: the value is 0 ~ 255;
- sdm2: the value is 0 ~ 63;
- odiv: odiv: the value is 0 ~ 31;
- The operating frequency range of the numerator is 350 MHz ~ 500 MHz.

$$350MHz < f_{\text{xtal}}(\text{sdm2} + \frac{\text{sdm1}}{2^8} + \frac{\text{sdm0}}{2^{16}} + 4) < 500MHz$$

Audio PLL can be manually enabled or disabled via registers RTC_CNTL_PLLA_FORCE_PU and RTC_CNTL_PLLA_FORCE_PD, respectively. Disabling it takes priority over enabling it. When RTC_CNTL_PLLA_FORCE_PU and RTC_CNTL_PLLA_FORCE_PD are 0, PLL follows the state of the system. When the system enters sleep mode, PLL will be disabled automatically; when the system wakes up, PLL will be enabled automatically.

3. Chip Boot Control

3.1 Overview

ESP32-S2 has three strapping pins:

- GPIO0
- GPIO45
- GPIO46

Software can read the values of the three strapping pins from register GPIO_STRAPPING. During chip reset triggered by power-on, brown-out or by analog super watchdog (see Chapter 2 *Reset and Clock*), hardware samples and stores the voltage level of strapping pins as strapping bit of “0” or “1” in latches, and hold these bits until the chip is powered down or shut down.

By default, GPIO0, GPIO45 and GPIO46 are connected to internal pull-up/pull-down during chip reset. Consequently, if the three GPIOs are unconnected or their connected external circuits are high-impedance, the internal weak pull-up/pull-down will determine the default input level of the strapping pins (see Table 3-1).

Table 3-1. Default configuration of strapping pins

Pin	GPIO0	GPIO45	GPIO46
Default	Pull-up	Pull-down	Pull-down

To change the default configuration of strapping pins, 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-S2. After the reset is released, the strapping pins work as normal-function pins.

3.2 Boot Mode

GPIO0 and GPIO46 control the boot mode after reset.

Table 3-2. Boot Mode

Pin	SPI Boot	Download Boot
GPIO0	1	0
GPIO46	x	0

Table 3-2 shows the strapping pin values and the associated boot modes. ”x” means that this value is ignored. Currently only the two boot modes shown are supported. The strapping combination of GPIO0 = 0 and GPIO46 = 1 is not supported and will trigger unexpected behavior.

In SPI boot mode, the CPU boots the system by reading the program stored in SPI flash.

In download boot mode, users can download code to SRAM or Flash using UART0, UART1, QPI 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 behavior:

- EFUSE_DIS_FORCE_DOWNLOAD. If this eFuse is 0 (default), software can switch the chip from SPI boot mode to download boot mode by setting register RTC_FORCE_DOWNLOAD_BOOT and triggering a CPU reset. If this eFuse is 1, this register 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 doesn't support any SRAM or register operations. This eFuse is ignored if download boot mode is disabled.

3.3 ROM Code Printing to UART

GPIO46 controls ROM code printing of information during the early boot process. This GPIO is used together with the UART_PRINT_CONTROL eFuse.

Table 3-3. ROM Code Printing Control

UART_PRINT_CONTROL	GPIO46	ROM Code Printing
0	-	ROM code will always print information to UART during boot. GPIO46 is not used.
1	0	Print is enabled during boot
	1	Print is disabled
2	0	Print is disabled
	1	Print is enabled during boot
3	-	Print is always disabled during boot. GPIO46 is not used.

ROM code will print to pin U0TXD (default) or to pin DAC_1, depending on the eFuse bit UART_PRINT_CHANNEL (0: UART0; 1: DAC_1).

3.4 VDD_SPI Voltage

GPIO45 is used to select the VDD_SPI power supply voltage at reset:

- GPIO45 = 0, VDD_SPI pin is powered directly from VDD3P3_RTC_IO via resistor R_{SPI}. Typically this voltage is 3.3 V.
- GPIO45 = 1, VDD_SPI pin is powered from internal 1.8 V LDO.

This functionality can be overridden by setting eFuse bit VDD_SPI_FORCE to 1, in which case the VDD_SPI_TIEH eFuse value determines the VDD_SPI voltage:

- VDD_SPI_FORCE = 1 and VDD_SPI_TIEH = 0, VDD_SPI connects to 1.8 V LDO.
- VDD_SPI_FORCE = 1 and VDD_SPI_TIEH = 1, VDD_SPI connects to VDD3P3_RTC_IO.

4. Interrupt Matrix

4.1 Overview

The interrupt matrix embedded in ESP32-S2 independently allocates peripheral interrupt sources to the CPU peripheral interrupts, so as to timely inform the CPU to process the interrupts once the interrupt signals are generated. This flexible function is applicable to a variety of application scenarios.

4.2 Features

- Accept 95 peripheral interrupt sources as input
- Generate 26 peripheral interrupts to the CPU as output
- Disable CPU non-maskable interrupt (NMI) sources
- Query current interrupt status of peripheral interrupt sources

The structure of the interrupt matrix is shown in Figure 4-1.

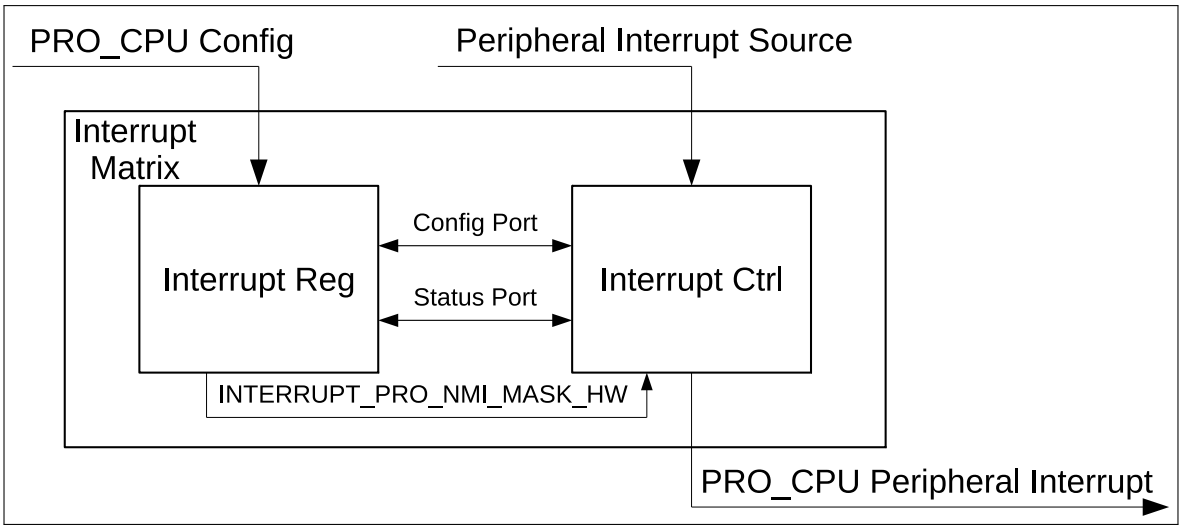


Figure 4-1. Interrupt Matrix Structure

4.3 Functional Description

4.3.1 Peripheral Interrupt Sources

ESP32-S2 has 95 peripheral interrupt sources in total, all of which can be allocated to the CPU. For the peripheral interrupt sources and their configuration/status registers, please refer to Table 4-1.

Table 4-1. CPU peripheral interrupt configuration/status registers and peripheral interrupt sources

No.	Source	Configuration Register	Status Register	
			Bit	Name
0	MAC_INTR	INTERRUPT_PRO_MAC_INTR_MAP_REG	0	INTERRUPT_PRO_INTR_STATUS_REG_0_REG
1	MAC_NMI	INTERRUPT_PRO_MAC_NMI_MAP_REG	1	
2	PWR_INTR	INTERRUPT_PRO_PWR_INTR_MAP_REG	2	
3	BB_INT	INTERRUPT_PRO_BB_INT_MAP_REG	3	
4	BT_MAC_INT	INTERRUPT_PRO_BT_MAC_INT_MAP_REG	4	
5	BT_BB_INT	INTERRUPT_PRO_BT_BB_INT_MAP_REG	5	
6	BT_BB_NMI	INTERRUPT_PRO_BT_BB_NMI_MAP_REG	6	
7	RWBT_IRQ	INTERRUPT_PRO_RWBT_IRQ_MAP_REG	7	
8	RWBLE_IRQ	INTERRUPT_PRO_RWBLE_IRQ_MAP_REG	8	
9	RWBT_NMI	INTERRUPT_PRO_RWBT_NMI_MAP_REG	9	
10	RWBLE_NMI	INTERRUPT_PRO_RWBLE_NMI_MAP_REG	10	
11	SLC0_INTR	INTERRUPT_PRO_SLC0_INTR_MAP_REG	11	
12	SLC1_INTR	INTERRUPT_PRO_SLC1_INTR_MAP_REG	12	
13	UHCI0_INTR	INTERRUPT_PRO_UHCI0_INTR_MAP_REG	13	
14	UHCI1_INTR	INTERRUPT_PRO_UHCI1_INTR_MAP_REG	14	
15	TG_T0_LEVEL_INT	INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP_REG	15	
16	TG_T1_LEVEL_INT	INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP_REG	16	
17	TG_WDT_LEVEL_INT	INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP_REG	17	
18	TG_LACT_LEVEL_INT	INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP_REG	18	
19	TG1_T0_LEVEL_INT	INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP_REG	19	
20	TG1_T1_LEVEL_INT	INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP_REG	20	
21	TG1_WDT_LEVEL_INT	INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP_REG	21	
22	TG1_LACT_LEVEL_INT	INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP_REG	22	
23	GPIO_INTERRUPT_PRO	INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP_REG	23	
24	GPIO_INTERRUPT_PRO_NMI	INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP_REG	24	
25	GPIO_INTERRUPT_APP	INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP_REG	25	
26	GPIO_INTERRUPT_APP_NMI	INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP_REG	26	
27	DEDICATED_GPIO_IN_INTR	INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP_REG	27	
28	CPU_INTR_FROM_CPU_0	INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP_REG	28	
29	CPU_INTR_FROM_CPU_1	INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP_REG	29	
30	CPU_INTR_FROM_CPU_2	INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP_REG	30	
31	CPU_INTR_FROM_CPU_3	INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP_REG	31	
32	SPI_INTR_1	INTERRUPT_PRO_SPI_INTR_1_MAP_REG	0	INTERRUPT_PRO_INTR_STATUS_REG_1_REG
33	SPI_INTR_2	INTERRUPT_PRO_SPI_INTR_2_MAP_REG	1	
34	SPI_INTR_3	INTERRUPT_PRO_SPI_INTR_3_MAP_REG	2	
35	I2S0_INT	INTERRUPT_PRO_I2S0_INT_MAP_REG	3	
36	I2S1_INT	INTERRUPT_PRO_I2S1_INT_MAP_REG	4	
37	UART_INTR	INTERRUPT_PRO_UART_INTR_MAP_REG	5	
38	UART1_INTR	INTERRUPT_PRO_UART1_INTR_MAP_REG	6	

No.	Source	Configuration Register	Status Register	
			Bit	Name
39	UART2_INTR	INTERRUPT_PRO_UART2_INTR_MAP_REG	7	
40	SDIO_HOST_INTERRUPT	INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP_REG	8	
41	PWM0_INTR	INTERRUPT_PRO_PWM0_INTR_MAP_REG	9	
42	PWM1_INTR	INTERRUPT_PRO_PWM1_INTR_MAP_REG	10	
43	PWM2_INTR	INTERRUPT_PRO_PWM2_INTR_MAP_REG	11	
44	PWM3_INTR	INTERRUPT_PRO_PWM3_INTR_MAP_REG	12	
45	LEDC_INT	INTERRUPT_PRO_LEDC_INT_MAP_REG	13	
46	EFUSE_INT	INTERRUPT_PRO_EFUSE_INT_MAP_REG	14	
47	CAN_INT	INTERRUPT_PRO_CAN_INT_MAP_REG	15	
48	USB_INTR	INTERRUPT_PRO_USB_INTR_MAP_REG	16	
49	RTC_CORE_INTR	INTERRUPT_PRO_RTC_CORE_INTR_MAP_REG	17	
50	RMT_INTR	INTERRUPT_PRO_RMT_INTR_MAP_REG	18	
51	PCNT_INTR	INTERRUPT_PRO_PCNT_INTR_MAP_REG	19	INTERRUPT_PRO_INTR_STATUS_REG_1_REG
52	I2C_EXT0_INTR	INTERRUPT_PRO_I2C_EXT0_INTR_MAP_REG	20	
53	I2C_EXT1_INTR	INTERRUPT_PRO_I2C_EXT1_INTR_MAP_REG	21	
54	RSA_INTR	INTERRUPT_PRO_RSA_INTR_MAP_REG	22	
55	SHA_INTR	INTERRUPT_PRO_SHA_INTR_MAP_REG	23	
56	AES_INTR	INTERRUPT_PRO_AES_INTR_MAP_REG	24	
57	SPI2_DMA_INT	INTERRUPT_PRO_SPI2_DMA_INT_MAP_REG	25	
58	SPI3_DMA_INT	INTERRUPT_PRO_SPI3_DMA_INT_MAP_REG	26	
59	WDG_INT	INTERRUPT_PRO_WDG_INT_MAP_REG	27	
60	TIMER_INT	INTERRUPT_PRO_TIMER_INT1_MAP_REG	28	
61	TIMER_INT2	INTERRUPT_PRO_TIMER_INT2_MAP_REG	29	
62	TG_T0_EDGE_INT	INTERRUPT_PRO_TG_T0_EDGE_INT_MAP_REG	30	
63	TG_T1_EDGE_INT	INTERRUPT_PRO_TG_T1_EDGE_INT_MAP_REG	31	
64	TG_WDT_EDGE_INT	INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP_REG	0	
65	TG_LACT_EDGE_INT	INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP_REG	1	
66	TG1_T0_EDGE_INT	INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP_REG	2	
67	TG1_T1_EDGE_INT	INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP_REG	3	
68	TG1_WDT_EDGE_INT	INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP_REG	4	
69	TG1_LACT_EDGE_INT	INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP_REG	5	
70	CACHE_IA_INT	INTERRUPT_PRO_CACHE_IA_INT_MAP_REG	6	
71	SYSTIMER_TARGET0_INT	INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP_REG	7	
72	SYSTIMER_TARGET1_INT	INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP_REG	8	INTERRUPT_PRO_INTR_STATUS_REG_2_REG
73	SYSTIMER_TARGET2_INT	INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP_REG	9	
74	ASSIST_DEBUG_INTR	INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP_REG	10	
75	PMS_PRO_IRAM0_ILG_INTR	INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP_REG	11	
76	PMS_PRO_DRAM0_ILG_INTR	INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP_REG	12	
77	PMS_PRO_DPORT_ILG_INTR	INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP_REG	13	
78	PMS_PRO_AHB_ILG_INTR	INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP_REG	14	
79	PMS_PRO_CACHE_ILG_INTR	INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP_REG	15	

No.	Source	Configuration Register	Status Register	
			Bit	Name
80	PMS_DMA_APB_I_ILG_INTR	INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP_REG	16	
81	PMS_DMA_RX_I_ILG_INTR	INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP_REG	17	
82	PMS_DMA_TX_I_ILG_INTR	INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP_REG	18	
83	SPI_MEM_REJECT_INTR	INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP_REG	19	
84	DMA_COPY_INTR	INTERRUPT_PRO_DMA_COPY_INTR_MAP_REG	20	
85	SPI4_DMA_INT	INTERRUPT_PRO_SPI4_DMA_INT_MAP_REG	21	
86	DCACHE_PRELOAD_INT	INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG	22	
87	DCACHE_PRELOAD_INT	INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG	23	INTERRUPT_PRO_INTR_STATUS_REG_2_REG
88	ICACHE_PRELOAD_INT	INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP_REG	24	
89	APB_ADC_INT	INTERRUPT_PRO_APB_ADC_INT_MAP_REG	25	
90	CRYPTO_DMA_INT	INTERRUPT_PRO_CRYPTO_DMA_INT_MAP_REG	26	
91	CPU_PERI_ERROR_INT	INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP_REG	27	
92	APB_PERI_ERROR_INT	INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP_REG	28	
93	DCACHE_SYNC_INT	INTERRUPT_PRO_DCACHE_SYNC_INT_MAP_REG	29	
94	ICACHE_SYNC_INT	INTERRUPT_PRO_ICACHE_SYNC_INT_MAP_REG	30	

4.3.2 CPU Interrupts

The CPU has 32 interrupts, including 26 peripheral interrupts and six internal interrupts. Table 4-2 lists all the interrupts.

- Peripheral Interrupts:
  - Level-triggered interrupts: triggered by high level signal. The interrupt sources should hold the level till the CPU handles the interrupts.
  - Edge-triggered interrupts: triggered on rising edge. The CPU responds to this kind of interrupts immediately.
  - NMI interrupt: once triggered, the NMI interrupt can not be masked by software using the CPU registers.
- Internal Interrupts:
  - Timer interrupts: triggered by internal timers and are used to generate periodic interrupts.
  - Software interrupts: triggered when software writes to special registers.
  - Profiling interrupt: triggered for performance monitoring and analysis.

ESP32-S2 supports the above-mentioned 32 interrupts at six levels as shown in the table below. A higher level corresponds to a higher priority. NMI has the highest interrupt priority and once triggered, the CPU must handle such interrupt.

Table 4-2. CPU Interrupts

No.	Category	Type	Priority Level
0	Peripheral	Level-triggered	1
1	Peripheral	Level-triggered	1
2	Peripheral	Level-triggered	1
3	Peripheral	Level-triggered	1
4	Peripheral	Level-triggered	1
5	Peripheral	Level-triggered	1
6	Internal	Timer.0	1
7	Internal	Software	1
8	Peripheral	Level-triggered	1
9	Peripheral	Level-triggered	1
10	Peripheral	Edge-triggered	1
11	Internal	Profiling	3
12	Peripheral	Level-triggered	1
13	Peripheral	Level-triggered	1
14	Peripheral	NMI	NMI
15	Internal	Timer.1	3
16	Internal	Timer.2	5
17	Peripheral	Level-triggered	1
18	Peripheral	Level-triggered	1
19	Peripheral	Level-triggered	2
20	Peripheral	Level-triggered	2

No.	Category	Type	Priority Level
21	Peripheral	Level-triggered	2
22	Peripheral	Edge-triggered	3
23	Peripheral	Level-triggered	3
24	Peripheral	Level-triggered	4
25	Peripheral	Level-triggered	4
26	Peripheral	Level-triggered	5
27	Peripheral	Level-triggered	3
28	Peripheral	Edge-triggered	4
29	Internal	Software	3
30	Peripheral	Edge-triggered	4
31	Peripheral	Level-triggered	5

4.3.3 Allocate Peripheral Interrupt Source to CPU Interrupt

In this section, the following terms are used to describe the operation of the interrupt matrix.

- Source_X: stands for a particular peripheral interrupt source, wherein, X means the number of this interrupt source in Table 4-1.
- INTERRUPT_PRO_X_MAP_REG: stands for a peripheral interrupt configuration register, corresponding to the peripheral interrupt source Source_X. In Table 4-1, the registers listed in column ”Configuration Register” correspond to the peripheral interrupt sources listed in column “Source”. For example, the configuration register for source MAC_INTR is INTERRUPT_PRO_MAC_INTR_MAP_REG.
- Interrupt_P: stands for the CPU peripheral interrupt numbered as Num_P. The value of Num_P can be 0 ~ 5, 8 ~ 10, 12 ~ 14, 17 ~ 28 and 30 ~ 31 (see Table 4-2).
- Interrupt_I: stands for the CPU internal interrupt numbered as Num_I. The value of Num_I can be 6, 7, 11, 15, 16 and 29 (see Table 4-2).

4.3.3.1 Allocate one peripheral interrupt source Source_X to CPU

Setting the corresponding configuration register INTERRUPT_PRO_X_MAP_REG of Source_X to Num_P will allocate this interrupt source to Interrupt_P. Num_P here can be any value from 0 ~ 5, 8 ~ 10, 12 ~ 14, 17 ~ 28 and 30 ~ 31. Note that one CPU interrupt can be shared by multiple peripherals.

4.3.3.2 Allocate multiple peripheral interrupt sources Source_X*n* to CPU

Setting the corresponding configuration register INTERRUPT_PRO_X*n*_MAP_REG of each interrupt source to the same Num_P will allocate all the sources to the same Interrupt_P. Any of these sources will trigger CPU Interrupt_P. When an interrupt signal is generated, software should check the interrupt status registers to figure out which peripheral the signal comes from.

4.3.3.3 Disable CPU peripheral interrupt source Source_X

Setting the corresponding configuration register INTERRUPT_PRO_X_MAP_REG of the source to any Num_I will disable this interrupt Source_X. The choice of Num_I does not matter, as none of Num_I is connected to the CPU. Therefore this functionality can be used to disable peripheral interrupt sources.

4.3.4 Disable CPU NMI Interrupt Sources

The interrupt matrix is able to mask all peripheral interrupt sources allocated to CPU No.14 NMI interrupt using hardware, depending on the internal signal INTERRUPT_PRO_NMI_MASK_HW. The signal comes from "Interrupt Reg" register configuration submodule inside interrupt matrix, see Figure 4-1. If the signal is set to high level, CPU will not respond to NMI interrupt.

4.3.5 Query Current Interrupt Status of Peripheral Interrupt Source

Current interrupt status of a peripheral interrupt source can be read via the bit value in INTERRUPT_PRO_INTR_STATUS_REG_*n*. For the mapping between INTERRUPT_PRO_INTR_STATUS_REG_*n* and peripheral interrupt sources, please refer to Table 4-1.

4.4 Base Address

Users can access interrupt matrix with the base address as shown in Table 4-3. For more information, see Chapter 1 *System and Memory*.

Table 4-3. Interrupt Matrix Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F4C2000

4.5 Register Summary

The address in the following table represents the address offset (relative address) with the respect to the peripheral base address, not the absolute address. For detailed information about the interrupt matrix base address, please refer to Section 4.4.

Name	Description	Address	Access
Configuration registers			
INTERRUPT_PRO_MAC_INTR_MAP_REG	MAC_INTR interrupt configuration register	0x0000	R/W
INTERRUPT_PRO_MAC_NMI_MAP_REG	MAC_NMI interrupt configuration register	0x0004	R/W
INTERRUPT_PRO_PWR_INTR_MAP_REG	PWR_INTR interrupt configuration register	0x0008	R/W
INTERRUPT_PRO_BB_INT_MAP_REG	BB_INT interrupt configuration register	0x000C	R/W
INTERRUPT_PRO_BT_MAC_INT_MAP_REG	BT_MAC_INT interrupt configuration register	0x0010	R/W
INTERRUPT_PRO_BT_BB_INT_MAP_REG	BT_BB_INT interrupt configuration register	0x0014	R/W
INTERRUPT_PRO_BT_BB_NMI_MAP_REG	BT_BB_NMI interrupt configuration register	0x0018	R/W
INTERRUPT_PRO_RWBT_IRQ_MAP_REG	RWBT_IRQ interrupt configuration register	0x001C	R/W
INTERRUPT_PRO_RWBLE_IRQ_MAP_REG	RWBLE_IRQ interrupt configuration register	0x0020	R/W
INTERRUPT_PRO_RWBT_NMI_MAP_REG	RWBT_NMI interrupt configuration register	0x0024	R/W
INTERRUPT_PRO_RWBLE_NMI_MAP_REG	RWBLE_NMI interrupt configuration register	0x0028	R/W
INTERRUPT_PRO_SLC0_INTR_MAP_REG	SLC0_INTR interrupt configuration register	0x002C	R/W
INTERRUPT_PRO_SLC1_INTR_MAP_REG	SLC1_INTR interrupt configuration register	0x0030	R/W
INTERRUPT_PRO_UHCI0_INTR_MAP_REG	UHCI0_INTR interrupt configuration register	0x0034	R/W
INTERRUPT_PRO_UHCI1_INTR_MAP_REG	UHCI1_INTR interrupt configuration register	0x0038	R/W
INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP_REG	TG_T0_LEVEL_INT interrupt configuration register	0x003C	R/W
INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP_REG	TG_T1_LEVEL_INT interrupt configuration register	0x0040	R/W
INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP_REG	TG_WDT_LEVEL_INT interrupt configuration register	0x0044	R/W
INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP_REG	TG_LACT_LEVEL_INT interrupt configuration register	0x0048	R/W
INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP_REG	TG1_T0_LEVEL_INT interrupt configuration register	0x004C	R/W
INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP_REG	TG1_T1_LEVEL_INT interrupt configuration register	0x0050	R/W
INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP_REG	TG1_WDT_LEVEL_INT interrupt configuration register	0x0054	R/W
INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP_REG	TG1_LACT_LEVEL_INT interrupt configuration register	0x0058	R/W
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP_REG	GPIO_INTERRUPT_PRO interrupt configuration register	0x005C	R/W
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP_REG	GPIO_INTERRUPT_PRO_NMI interrupt configuration register	0x0060	R/W
INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP_REG	GPIO_INTERRUPT_APP interrupt configuration register	0x0064	R/W
INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP_REG	GPIO_INTERRUPT_APP_NMI interrupt configuration register	0x0068	R/W
INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP_REG	DEDICATED_GPIO_IN_INTR interrupt configuration register	0x006C	R/W
INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP_REG	CPU_INTR_FROM_CPU_0 interrupt configuration register	0x0070	R/W
INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP_REG	CPU_INTR_FROM_CPU_1 interrupt configuration register	0x0074	R/W

Name	Description	Address	Access
INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP_REG	CPU_INTR_FROM_CPU_2 interrupt configuration register	0x0078	R/W
INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP_REG	CPU_INTR_FROM_CPU_3 interrupt configuration register	0x007C	R/W
INTERRUPT_PRO_SPI_INTR_1_MAP_REG	SPI_INTR_1 interrupt configuration register	0x0080	R/W
INTERRUPT_PRO_SPI_INTR_2_MAP_REG	SPI_INTR_2 interrupt configuration register	0x0084	R/W
INTERRUPT_PRO_SPI_INTR_3_MAP_REG	SPI_INTR_3 interrupt configuration register	0x0088	R/W
INTERRUPT_PRO_I2S0_INT_MAP_REG	I2S0_INT interrupt configuration register	0x008C	R/W
INTERRUPT_PRO_I2S1_INT_MAP_REG	I2S1_INT interrupt configuration register	0x0090	R/W
INTERRUPT_PRO_UART_INTR_MAP_REG	UART_INT interrupt configuration register	0x0094	R/W
INTERRUPT_PRO_UART1_INTR_MAP_REG	UART1_INT interrupt configuration register	0x0098	R/W
INTERRUPT_PRO_UART2_INTR_MAP_REG	UART2_INT interrupt configuration register	0x009C	R/W
INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP_REG	SDIO_HOST_INTERRUPT configuration register	0x00A0	R/W
INTERRUPT_PRO_PWM0_INTR_MAP_REG	PWM0_INTR interrupt configuration register	0x00A4	R/W
INTERRUPT_PRO_PWM1_INTR_MAP_REG	PWM1_INTR interrupt configuration register	0x00A8	R/W
INTERRUPT_PRO_PWM2_INTR_MAP_REG	PWM2_INTR interrupt configuration register	0x00AC	R/W
INTERRUPT_PRO_PWM3_INTR_MAP_REG	PWM3_INTR interrupt configuration register	0x00B0	R/W
INTERRUPT_PRO_LEDC_INT_MAP_REG	LEDC_INTR interrupt configuration register	0x00B4	R/W
INTERRUPT_PRO_EFUSE_INT_MAP_REG	EFUSE_INT interrupt configuration register	0x00B8	R/W
INTERRUPT_PRO_CAN_INT_MAP_REG	CAN_INT interrupt configuration register	0x00BC	R/W
INTERRUPT_PRO_USB_INTR_MAP_REG	USB_INT interrupt configuration register	0x00C0	R/W
INTERRUPT_PRO_RTC_CORE_INTR_MAP_REG	RTC_CORE_INTR interrupt configuration register	0x00C4	R/W
INTERRUPT_PRO_RMT_INTR_MAP_REG	RMT_INTR interrupt configuration register	0x00C8	R/W
INTERRUPT_PRO_PCNT_INTR_MAP_REG	PCNT_INTR interrupt configuration register	0x00CC	R/W
INTERRUPT_PRO_I2C_EXT0_INTR_MAP_REG	I2C_EXT0_INTR interrupt configuration register	0x00D0	R/W
INTERRUPT_PRO_I2C_EXT1_INTR_MAP_REG	I2C_EXT1_INTR interrupt configuration register	0x00D4	R/W
INTERRUPT_PRO_RSA_INTR_MAP_REG	RSA_INTR interrupt configuration register	0x00D8	R/W
INTERRUPT_PRO_SHA_INTR_MAP_REG	SHA_INTR interrupt configuration register	0x00DC	R/W
INTERRUPT_PRO_AES_INTR_MAP_REG	AES_INTR interrupt configuration register	0x00E0	R/W
INTERRUPT_PRO_SPI2_DMA_INT_MAP_REG	SPI2_DMA_INT interrupt configuration register	0x00E4	R/W
INTERRUPT_PRO_SPI3_DMA_INT_MAP_REG	SPI3_DMA_INT interrupt configuration register	0x00E8	R/W
INTERRUPT_PRO_WDG_INT_MAP_REG	WDG_INT interrupt configuration register	0x00EC	R/W
INTERRUPT_PRO_TIMER_INT1_MAP_REG	TIMER_INT1 interrupt configuration register	0x00F0	R/W

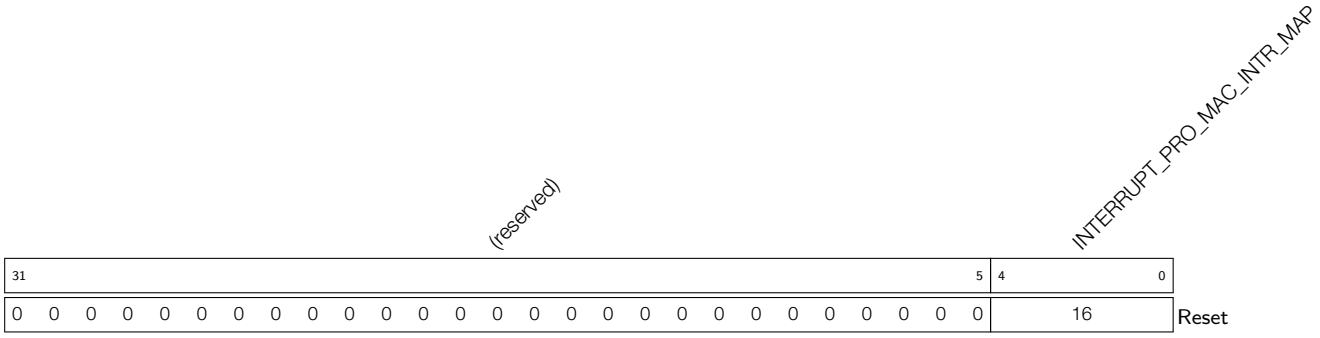
Name	Description	Address	Access
INTERRUPT_PRO_TIMER_INT2_MAP_REG	TIMER_INT2 interrupt configuration register	0x00F4	R/W
INTERRUPT_PRO_TG_T0_EDGE_INT_MAP_REG	TG_T0_EDGE_INT interrupt configuration register	0x00F8	R/W
INTERRUPT_PRO_TG_T1_EDGE_INT_MAP_REG	TG_T1_EDGE_INT interrupt configuration register	0x00FC	R/W
INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP_REG	TG_WDT_EDGE_INT interrupt configuration register	0x0100	R/W
INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP_REG	TG_LACT_EDGE_INT interrupt configuration register	0x0104	R/W
INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP_REG	TG1_T0_EDGE_INT interrupt configuration register	0x0108	R/W
INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP_REG	TG1_T1_EDGE_INT interrupt configuration register	0x010C	R/W
INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP_REG	TG1_WDT_EDGE_INT interrupt configuration register	0x0110	R/W
INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP_REG	TG1_LACT_EDGE_INT interrupt configuration register	0x0114	R/W
INTERRUPT_PRO_CACHE_IA_INT_MAP_REG	CACHE_IA_INT interrupt configuration register	0x0118	R/W
INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP_REG	SYSTIMER_TARGET0_INT interrupt configuration register	0x011C	R/W
INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP_REG	SYSTIMER_TARGET1_INT interrupt configuration register	0x0120	R/W
INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP_REG	SYSTIMER_TARGET2 interrupt configuration register	0x0124	R/W
INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP_REG	ASSIST_DEBUG_INTR interrupt configuration register	0x0128	R/W
INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP_REG	PMS_PRO_IRAM0_ILG interrupt configuration register	0x012C	R/W
INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP_REG	PMS_PRO_DRAM0_ILG interrupt configuration register	0x0130	R/W
INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP_REG	PMS_PRO_DPORT_ILG interrupt configuration register	0x0134	R/W
INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP_REG	PMS_PRO_AHB_ILG interrupt configuration register	0x0138	R/W
INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP_REG	PMS_PRO_CACHE_ILG interrupt configuration register	0x013C	R/W
INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP_REG	PMS_DMA_APB_I_ILG interrupt configuration register	0x0140	R/W
INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP_REG	PMS_DMA_RX_I_ILG interrupt configuration register	0x0144	R/W
INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP_REG	PMS_DMA_TX_I_ILG interrupt configuration register	0x0148	R/W
INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP_REG	SPI_MEM_REJECT_INTR interrupt configuration register	0x014C	R/W
INTERRUPT_PRO_DMA_COPY_INTR_MAP_REG	DMA_COPY_INTR interrupt configuration register	0x0150	R/W
INTERRUPT_PRO_SPI4_DMA_INT_MAP_REG	SPI4_DMA_INT interrupt configuration register	0x0154	R/W
INTERRUPT_PRO_SPI_INTR_4_MAP_REG	SPI_INTR_4 interrupt configuration register	0x0158	R/W
INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG	DCACHE_PRELOAD_INT interrupt configuration register	0x015C	R/W
INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP_REG	ICACHE_PRELOAD_INT interrupt configuration register	0x0160	R/W
INTERRUPT_PRO_APB_ADC_INT_MAP_REG	APB_ADC_INT interrupt configuration register	0x0164	R/W
INTERRUPT_PRO_CRYPTO_DMA_INT_MAP_REG	CRYPTO_DMA_INT interrupt configuration register	0x0168	R/W
INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP_REG	CPU_PERI_ERROR_INT interrupt configuration register	0x016C	R/W

Name	Description	Address	Access
INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP_REG	APB_PERI_ERROR_INT interrupt configuration register	0x0170	R/W
INTERRUPT_PRO_DCACHE_SYNC_INT_MAP_REG	DCACHE_SYNC_INT interrupt configuration register	0x0174	R/W
INTERRUPT_PRO_ICACHE_SYNC_INT_MAP_REG	ICACHE_SYNC_INT interrupt configuration register	0x0178	R/W
INTERRUPT_CLOCK_GATE_REG	NMI interrupt signals mask register	0x0188	R/W
Interrupt status registers			
INTERRUPT_PRO_INTR_STATUS_REG_0_REG	Interrupt status register 0	0x017C	RO
INTERRUPT_PRO_INTR_STATUS_REG_1_REG	Interrupt status register 1	0x0180	RO
INTERRUPT_PRO_INTR_STATUS_REG_2_REG	Interrupt status register 2	0x0184	RO
Version register			
INTERRUPT_REG_DATE_REG	Version control register	0x0FFC	R/W

4.6 Registers

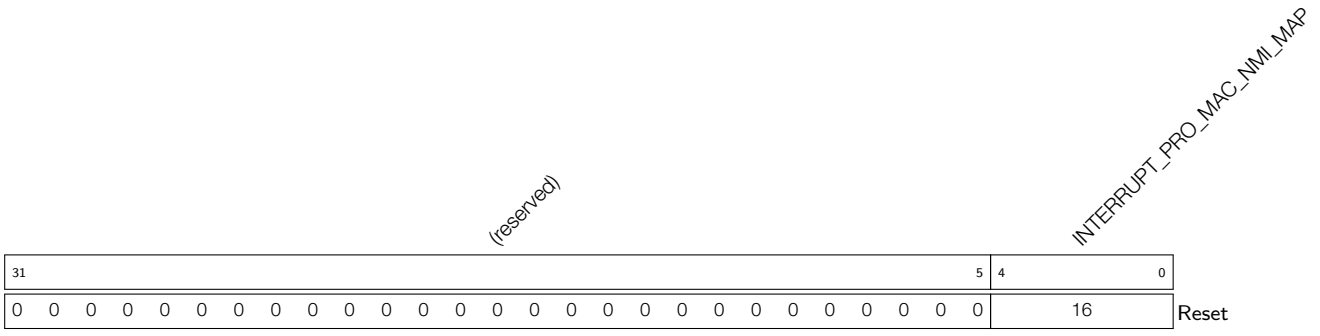
The address in the following part represents the address offset (relative address) with respect to the peripheral base address, not the absolute address. For detailed information about the interrupt matrix base address, please refer to Section 4.4.

Register 4.1: INTERRUPT_PRO_MAC_INTR_MAP_REG (0x0000)



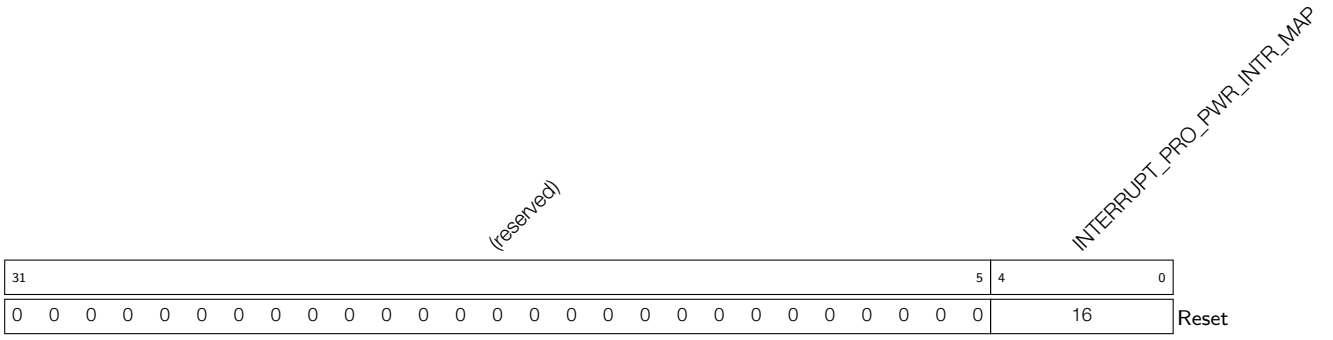
INTERRUPT_PRO_MAC_INTR_MAP This register is used to map MAC_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.2: INTERRUPT_PRO_MAC_NMI_MAP_REG (0x0004)



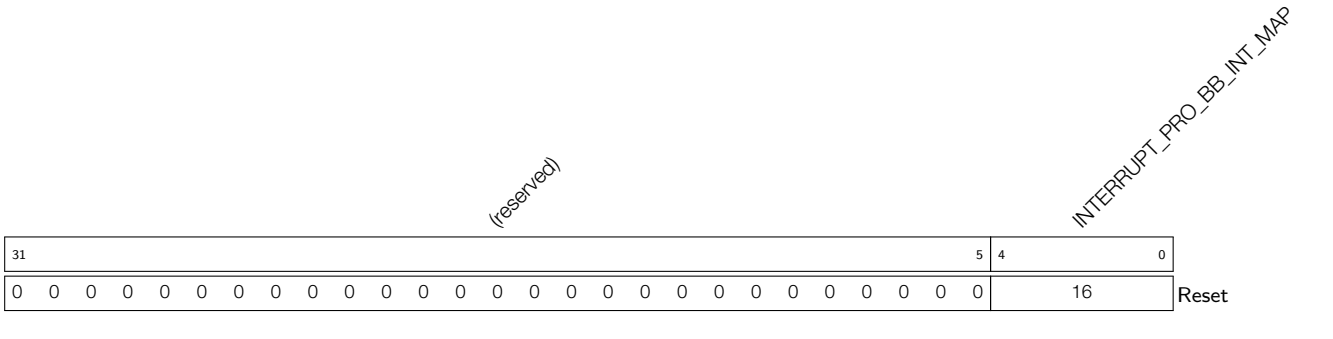
INTERRUPT_PRO_MAC_NMI_MAP This register is used to map MAC_NMI interrupt signal to one of the CPU interrupts. (R/W)

Register 4.3: INTERRUPT_PRO_PWR_INTR_MAP_REG (0x0008)



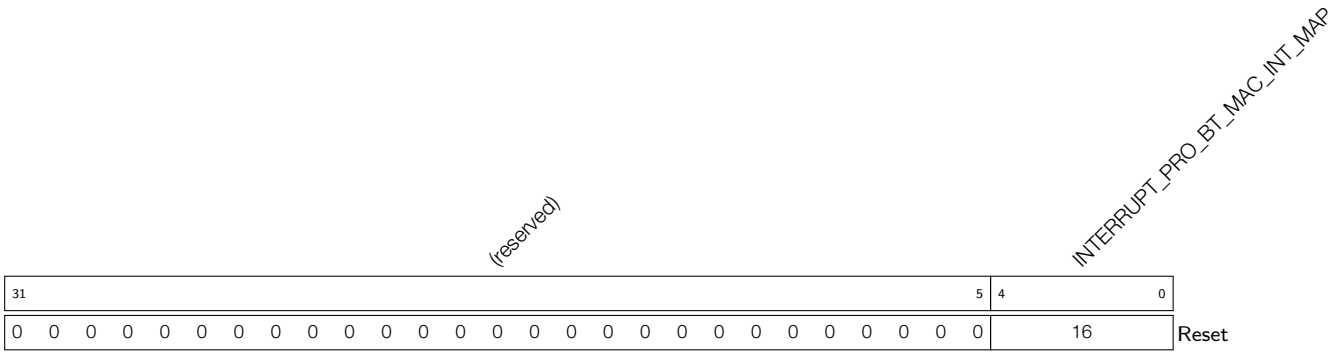
INTERRUPT_PRO_PWR_INTR_MAP This register is used to map PWR_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.4: INTERRUPT_PRO_BB_INT_MAP_REG (0x000C)



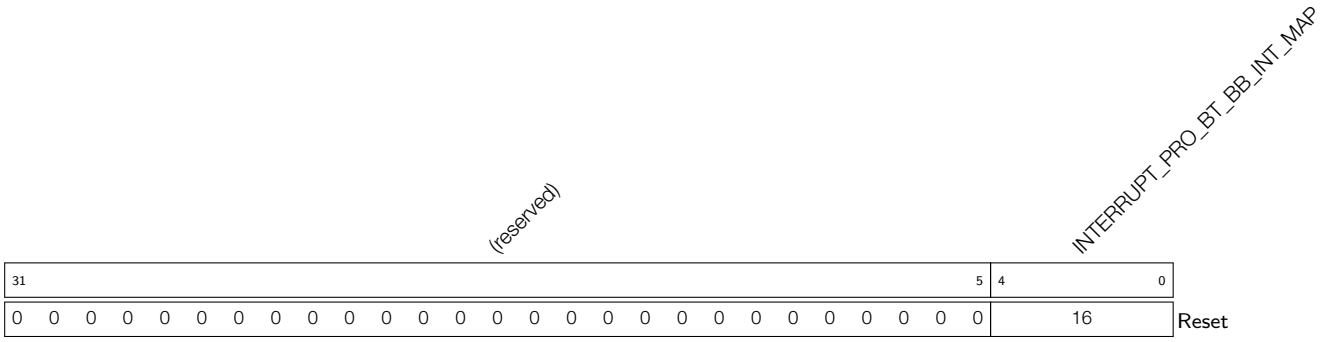
INTERRUPT_PRO_BB_INT_MAP This register is used to map BB_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.5: INTERRUPT_PRO_BT_MAC_INT_MAP_REG (0x0010)



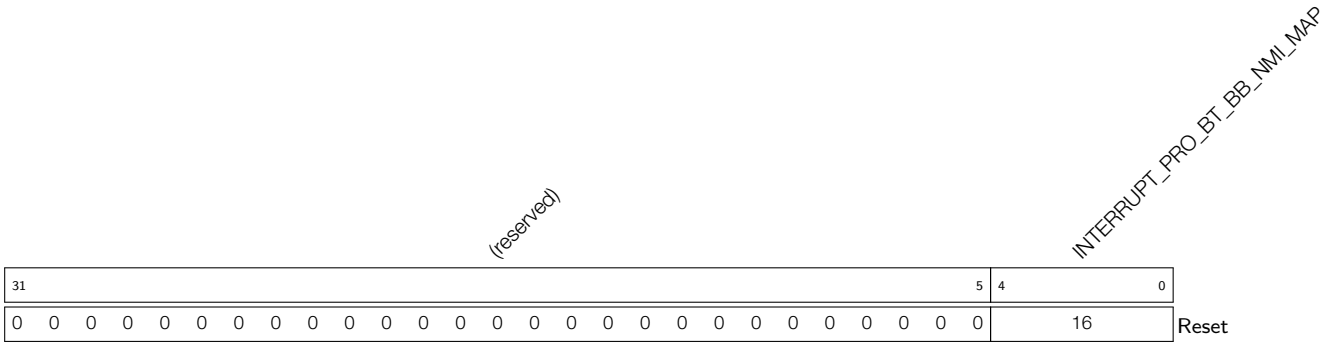
INTERRUPT_PRO_BT_MAC_INT_MAP This register is used to map BT_MAC_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.6: INTERRUPT_PRO_BT_BB_INT_MAP_REG (0x0014)



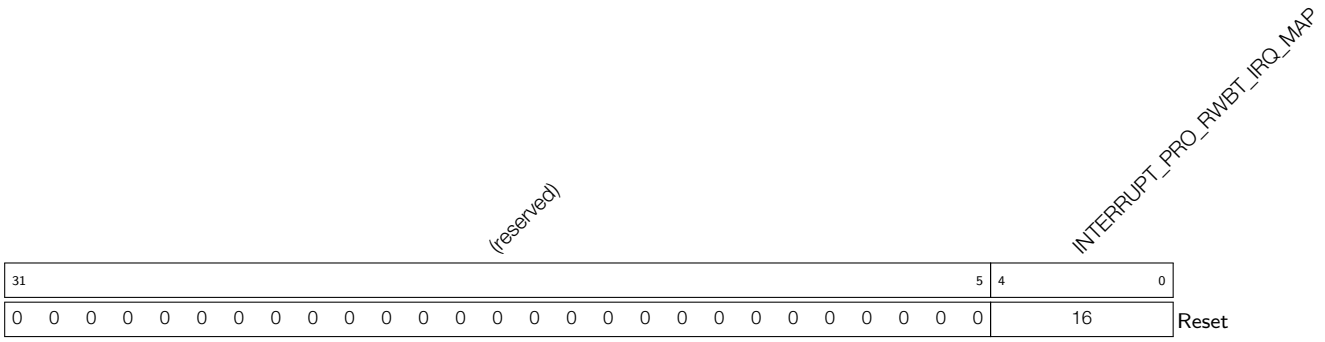
INTERRUPT_PRO_BT_BB_INT_MAP This register is used to map BT_BB_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.7: INTERRUPT_PRO_BT_BB_NMI_MAP_REG (0x0018)



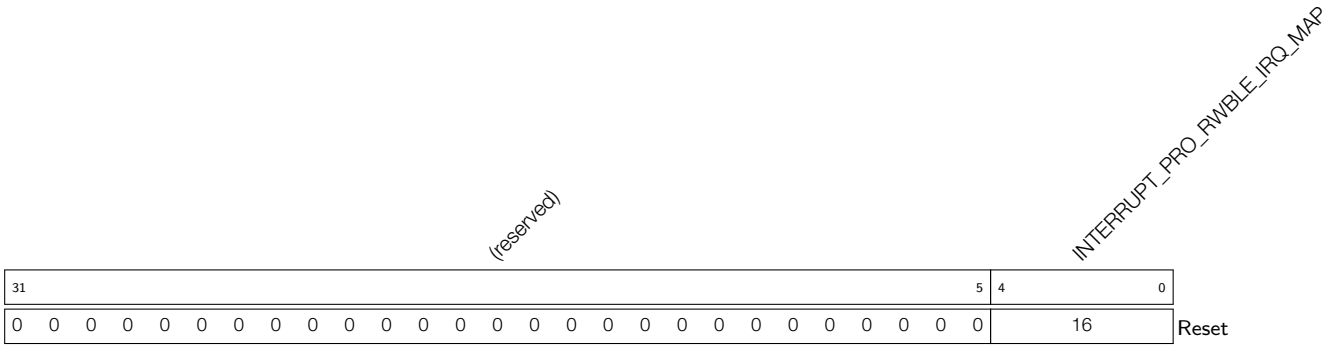
INTERRUPT_PRO_BT_BB_NMI_MAP This register is used to map BT_BB_NMI interrupt signal to one of the CPU interrupts. (R/W)

Register 4.8: INTERRUPT_PRO_RWBT_IRQ_MAP_REG (0x001C)



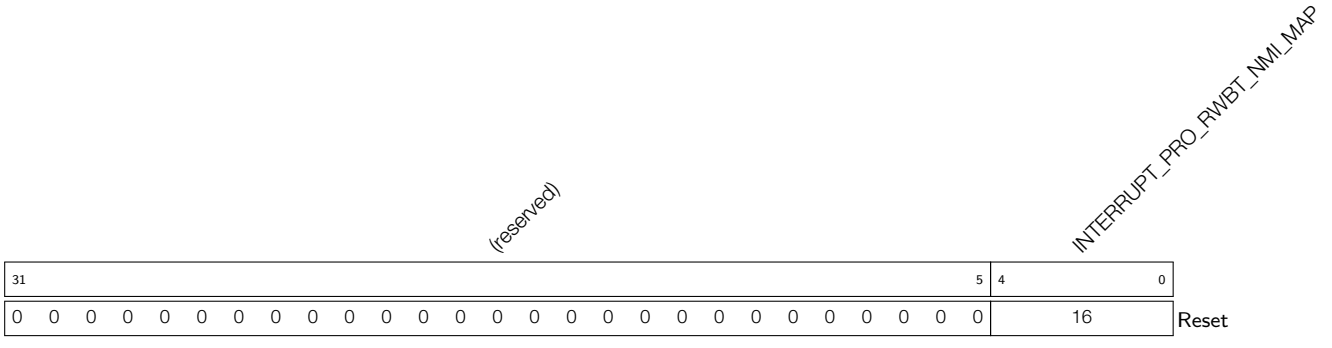
INTERRUPT_PRO_RWBT_IRQ_MAP This register is used to map RWBT_IRQ interrupt signal to one of the CPU interrupts. (R/W)

Register 4.9: INTERRUPT_PRO_RWBLE_IRQ_MAP_REG (0x0020)



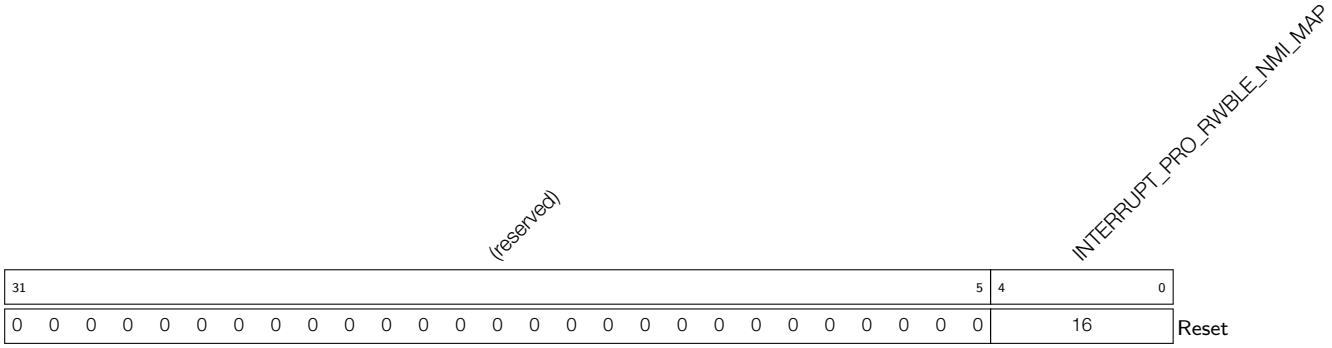
INTERRUPT_PRO_RWBLE_IRQ_MAP This register is used to map RWBLE_IRQ interrupt signal to one of the CPU interrupts. (R/W)

Register 4.10: INTERRUPT_PRO_RWBT_NMI_MAP_REG (0x0024)



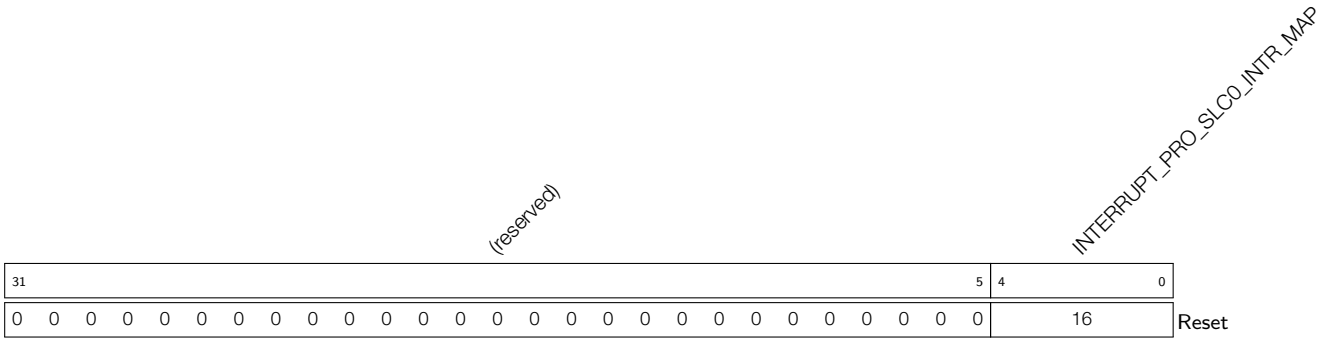
INTERRUPT_PRO_RWBT_NMI_MAP This register is used to map RWBT_NMI interrupt signal to one of the CPU interrupts. (R/W)

Register 4.11: INTERRUPT_PRO_RWBLE_NMI_MAP_REG (0x0028)



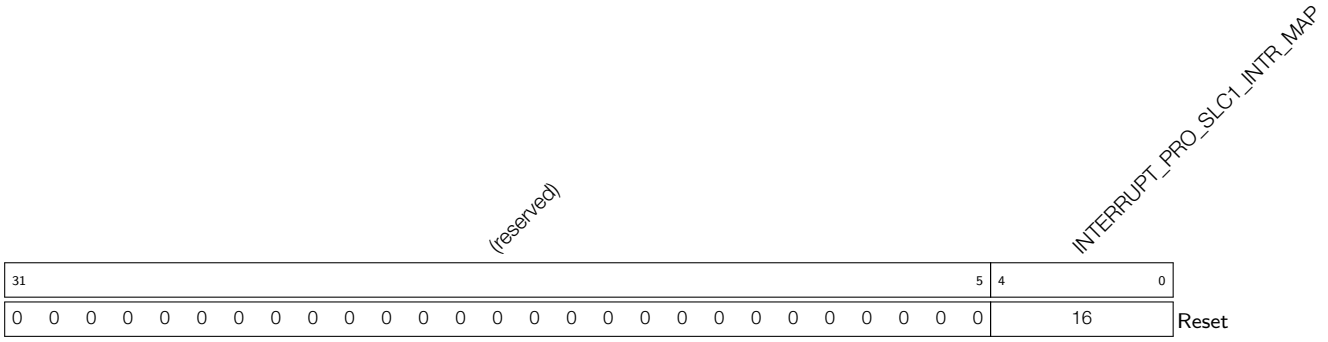
INTERRUPT_PRO_RWBLE_NMI_MAP This register is used to map RWBLE_NMI interrupt signal to one of the CPU interrupts. (R/W)

Register 4.12: INTERRUPT_PRO_SLC0_INTR_MAP_REG (0x002C)



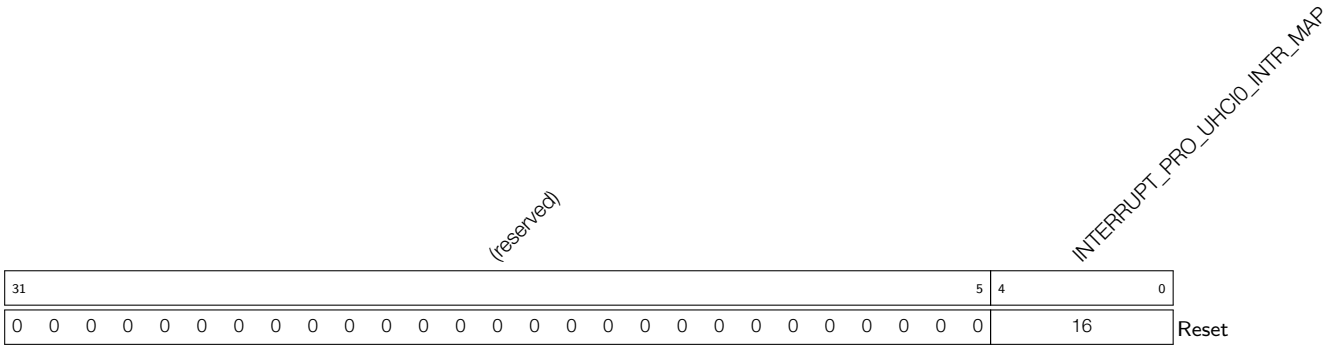
INTERRUPT_PRO_SLC0_INTR_MAP This register is used to map SLC0_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.13: INTERRUPT_PRO_SLC1_INTR_MAP_REG (0x0030)



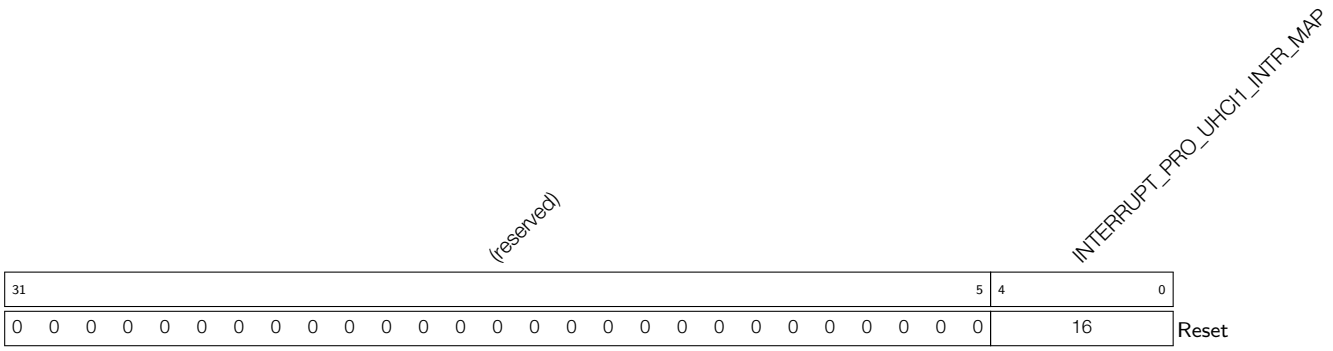
INTERRUPT_PRO_SLC1_INTR_MAP This register is used to map SLC1_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.14: INTERRUPT_PRO_UHCI0_INTR_MAP_REG (0x0034)



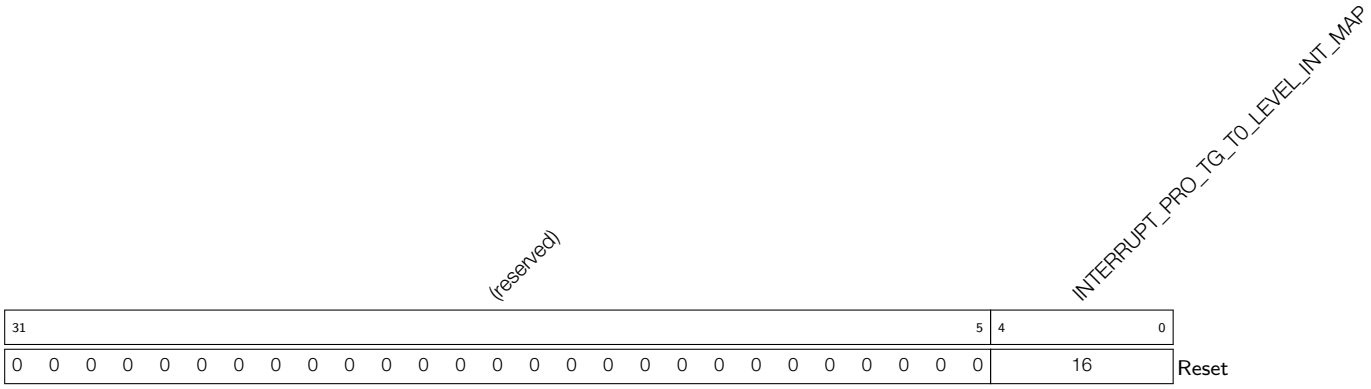
INTERRUPT_PRO_UHCI0_INTR_MAP This register is used to map UHCI0_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.15: INTERRUPT_PRO_UHCI1_INTR_MAP_REG (0x0038)



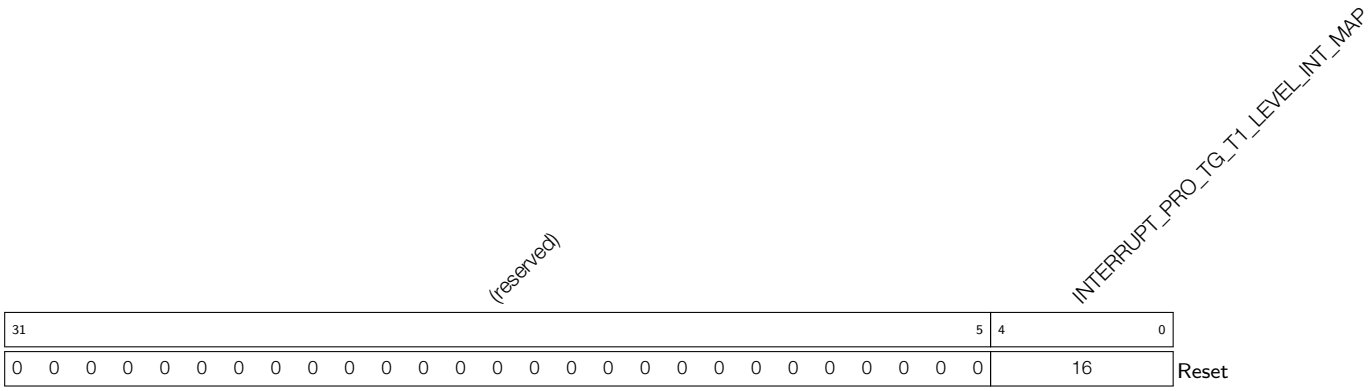
INTERRUPT_PRO_UHCI1_INTR_MAP This register is used to map UHCI1_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.16: INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP_REG (0x003C)



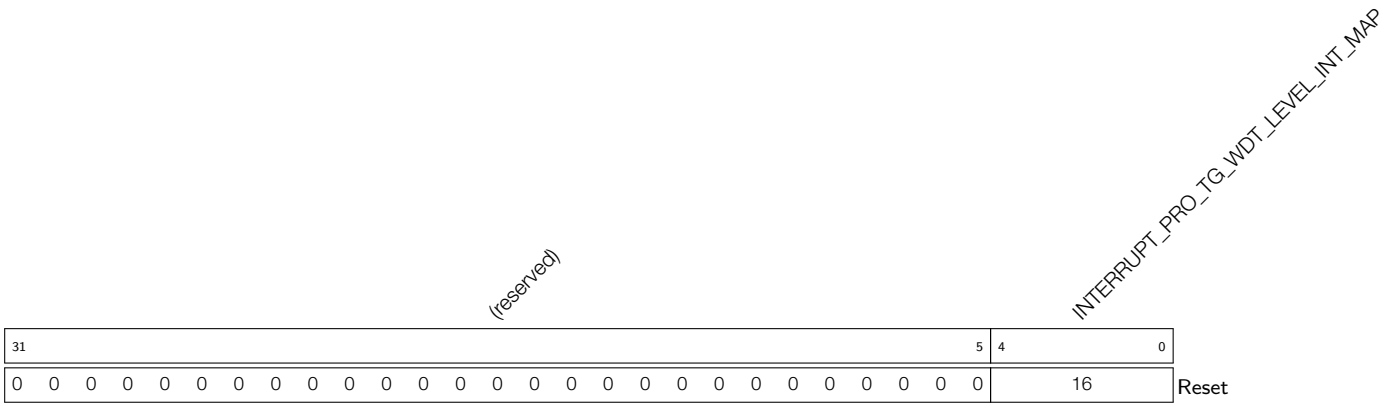
INTERRUPT_PRO_TG_T0_LEVEL_INT_MAP This register is used to map TG_T0_LEVEL_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.17: INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP_REG (0x0040)



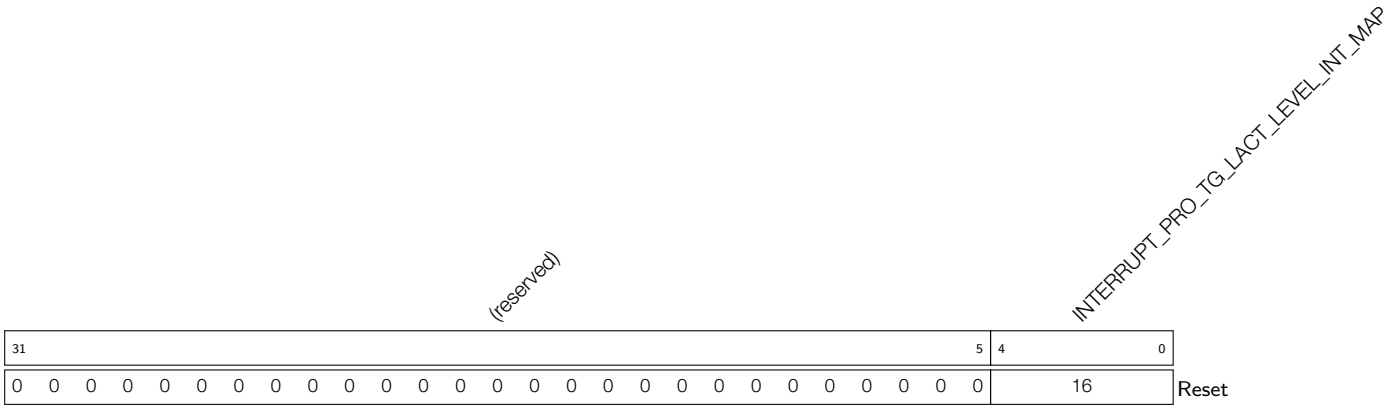
INTERRUPT_PRO_TG_T1_LEVEL_INT_MAP This register is used to map TG_T1_LEVEL_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.18: INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP_REG (0x0044)



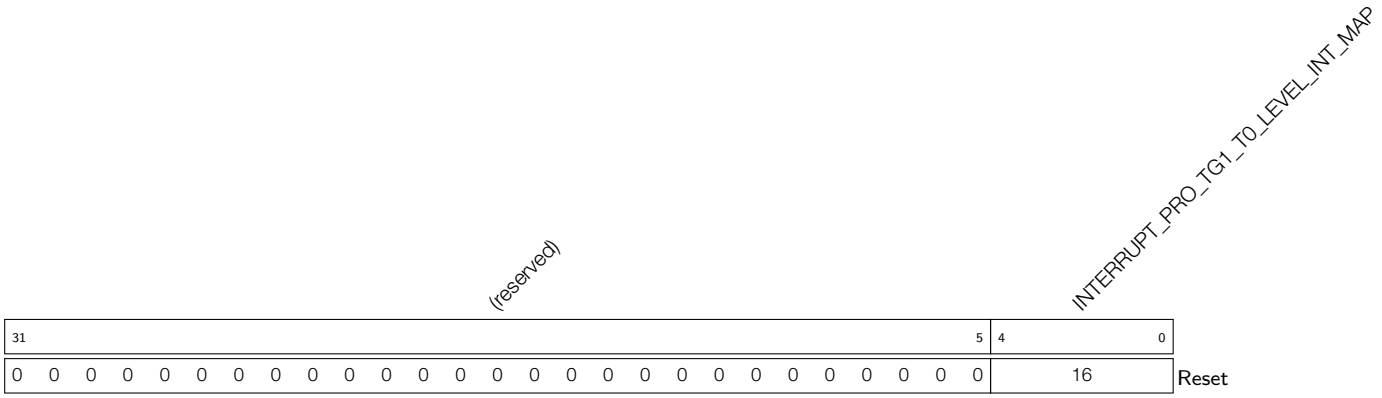
INTERRUPT_PRO_TG_WDT_LEVEL_INT_MAP This register is used to map TG_WDT_LEVEL_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.19: INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP_REG (0x0048)



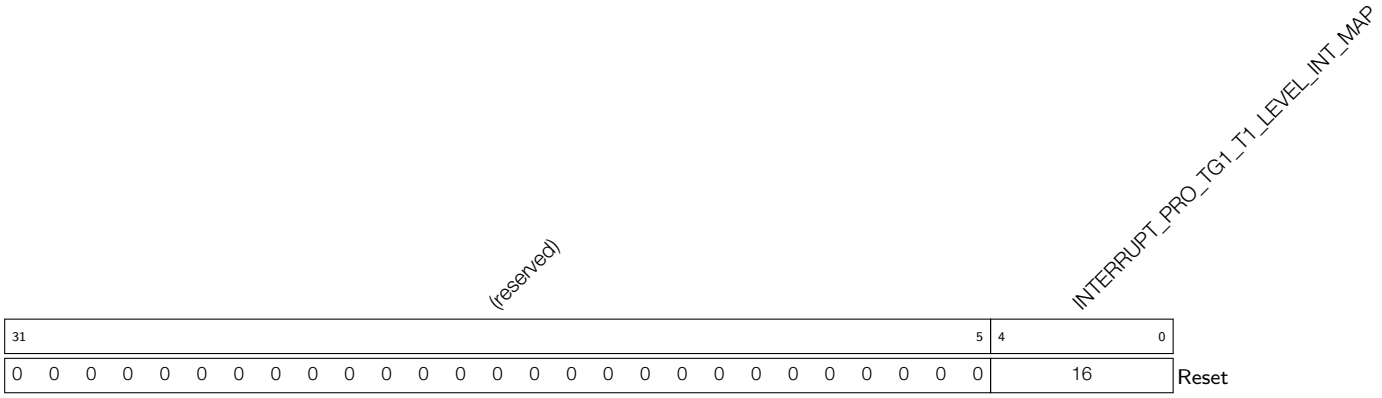
INTERRUPT_PRO_TG_LACT_LEVEL_INT_MAP This register is used to map TG_LACT_LEVEL_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.20: INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP_REG (0x004C)



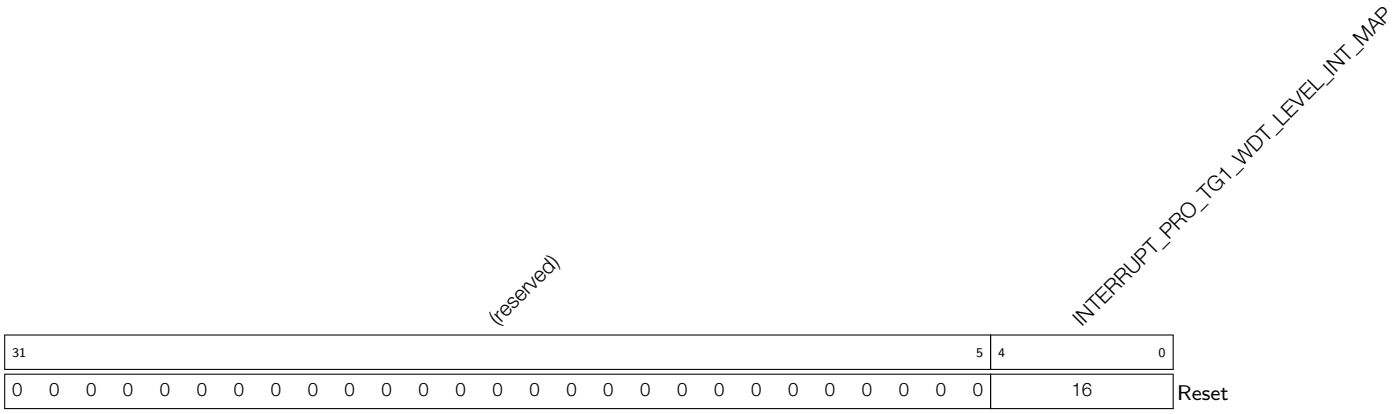
INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP This register is used to map TG1_T0_LEVEL_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.21: INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP_REG (0x0050)



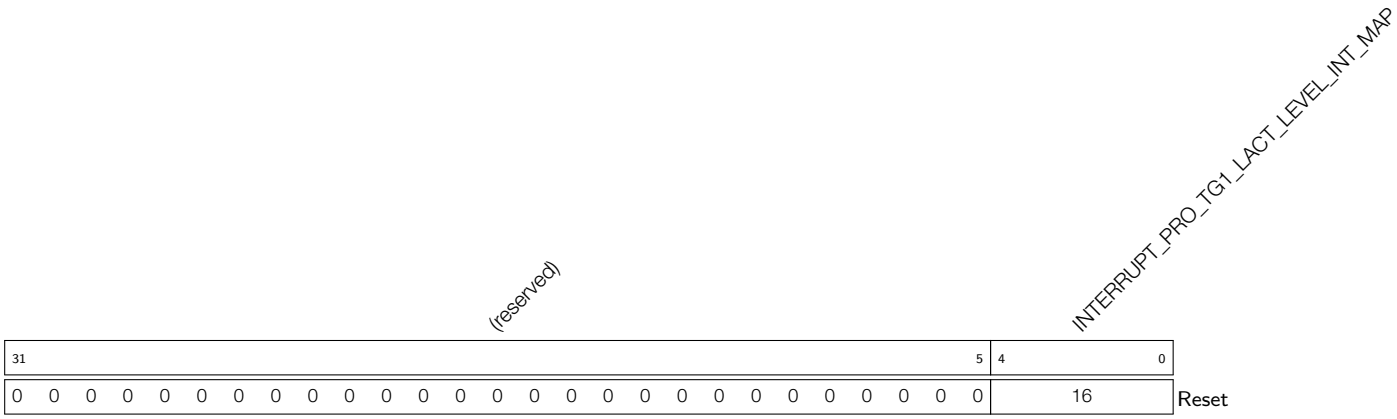
INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP This register is used to map TG1_T1_LEVEL_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.22: INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP_REG (0x0054)



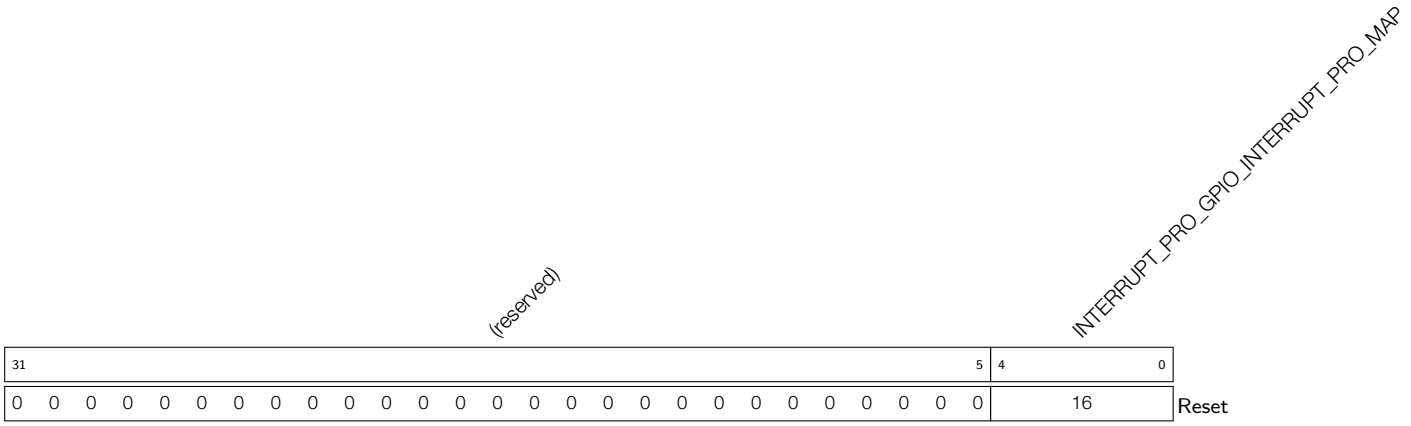
INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP This register is used to map TG1_WDT_LEVEL_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.23: INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP_REG (0x0058)



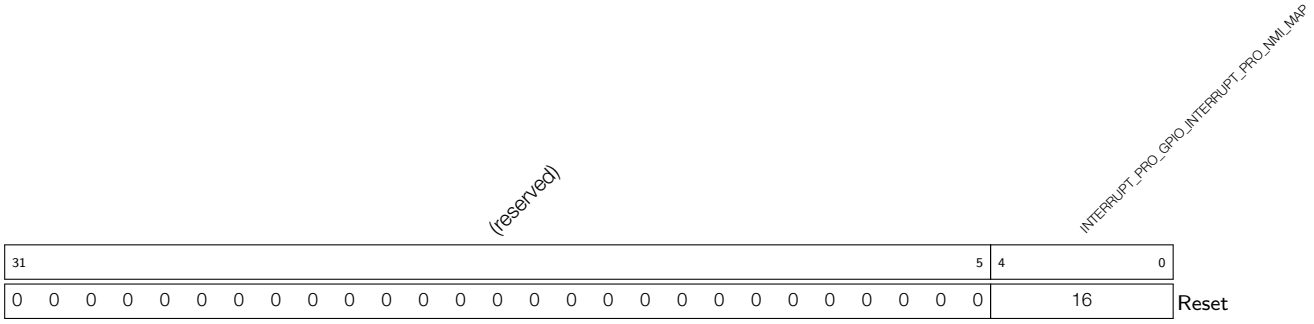
INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP This register is used to map TG1_LACT_LEVEL_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.24: INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP_REG (0x005C)



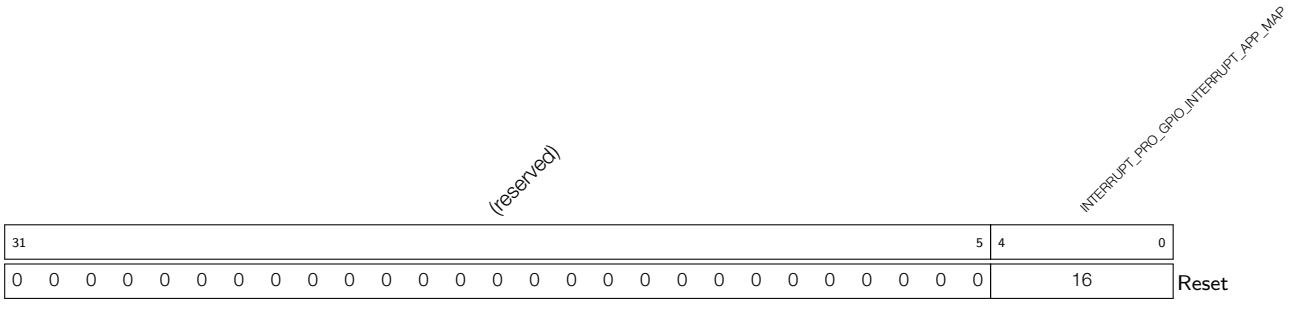
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_MAP This register is used to map GPIO_INTERRUPT_PRO interrupt signal to one of the CPU interrupts. (R/W)

Register 4.25: INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP_REG (0x0060)



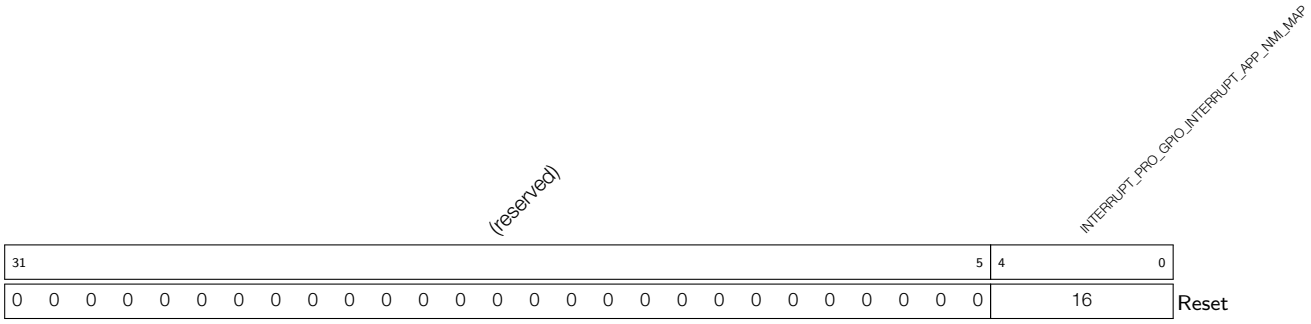
INTERRUPT_PRO_GPIO_INTERRUPT_PRO_NMI_MAP This register is used to map GPIO_INTERRUPT_PRO_NMI interrupt signal to one of the CPU interrupts. (R/W)

Register 4.26: INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP_REG (0x0064)



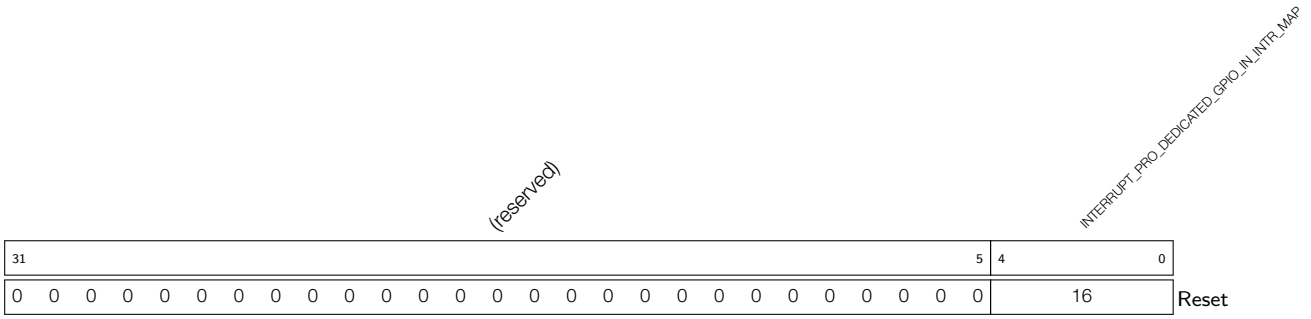
INTERRUPT_PRO_GPIO_INTERRUPT_APP_MAP This register is used to map GPIO_INTERRUPT_APP interrupt signal to one of the CPU interrupts. (R/W)

Register 4.27: INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP_REG (0x0068)



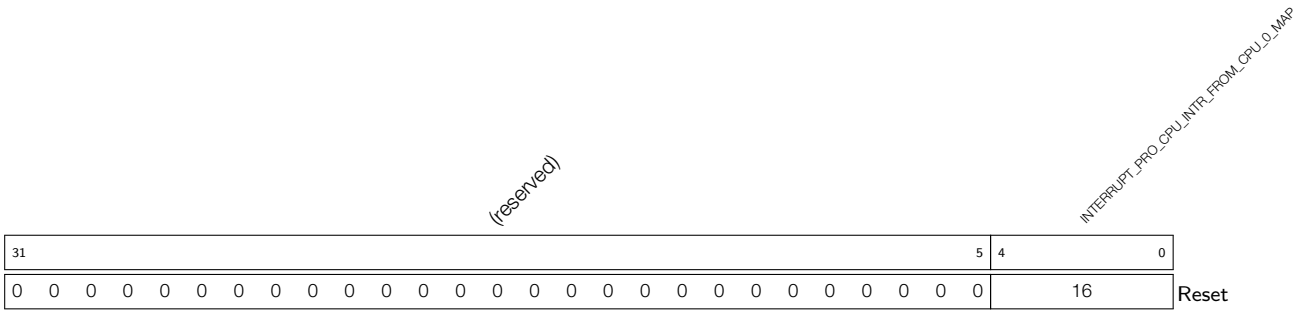
INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP This register is used to map GPIO_INTERRUPT_APP_NMI interrupt signal to one of the CPU interrupts. (R/W)

Register 4.28: INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP_REG (0x006C)



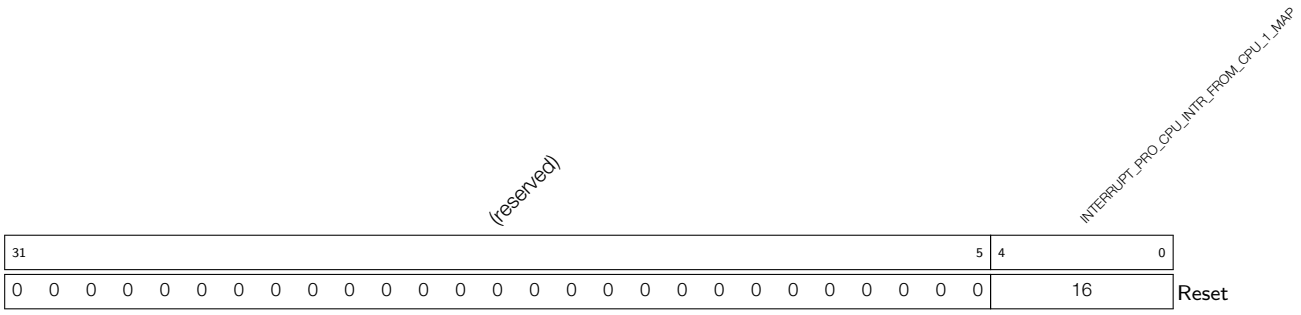
INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP This register is used to map DEDICATED_GPIO_IN_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.29: INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP_REG (0x0070)



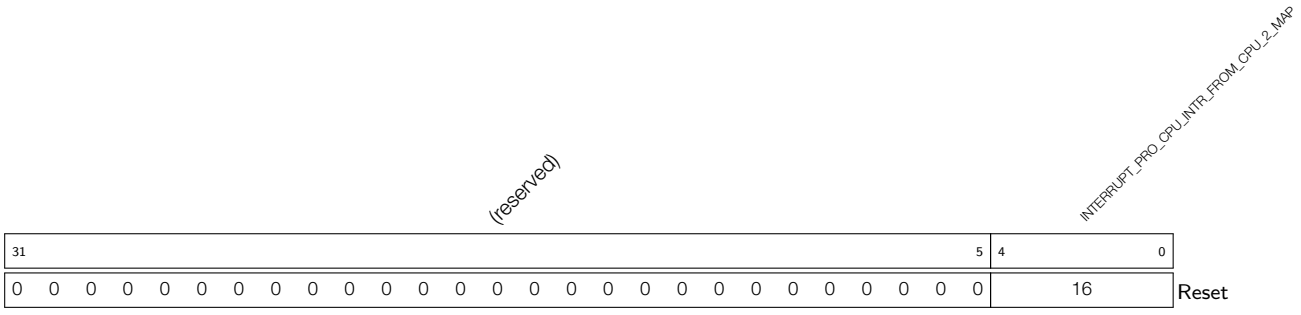
INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP This register is used to map CPU_INTR_FROM_CPU_0 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.30: INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP_REG (0x0074)



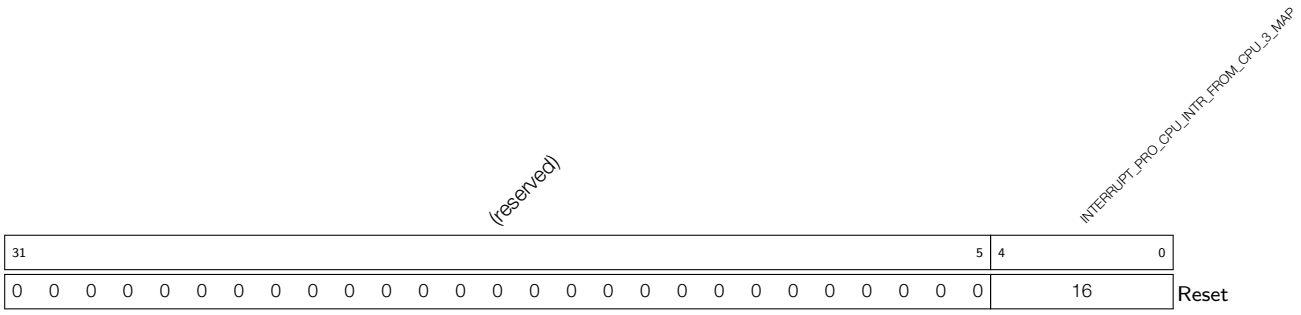
INTERRUPT_PRO_CPU_INTR_FROM_CPU_1_MAP This register is used to map CPU_INTR_FROM_CPU_1 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.31: INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP_REG (0x0078)



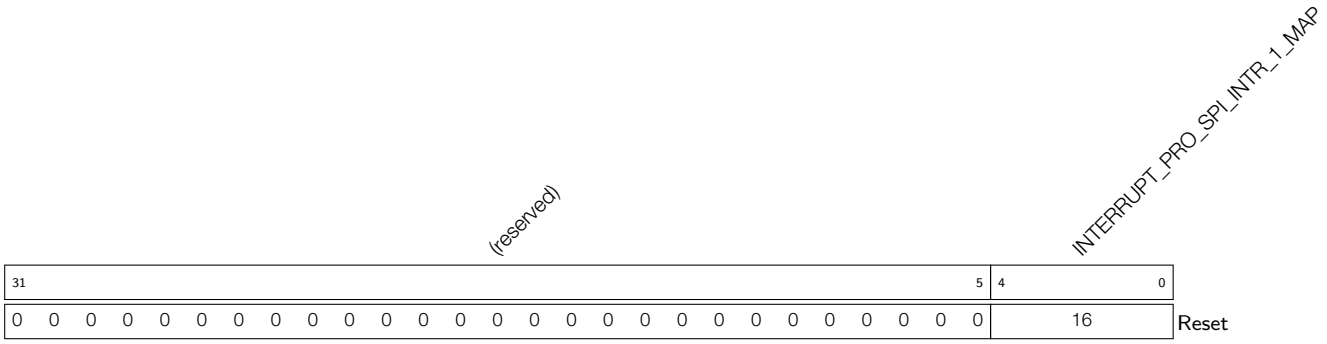
INTERRUPT_PRO_CPU_INTR_FROM_CPU_2_MAP This register is used to map CPU_INTR_FROM_CPU_2 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.32: INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP_REG (0x007C)



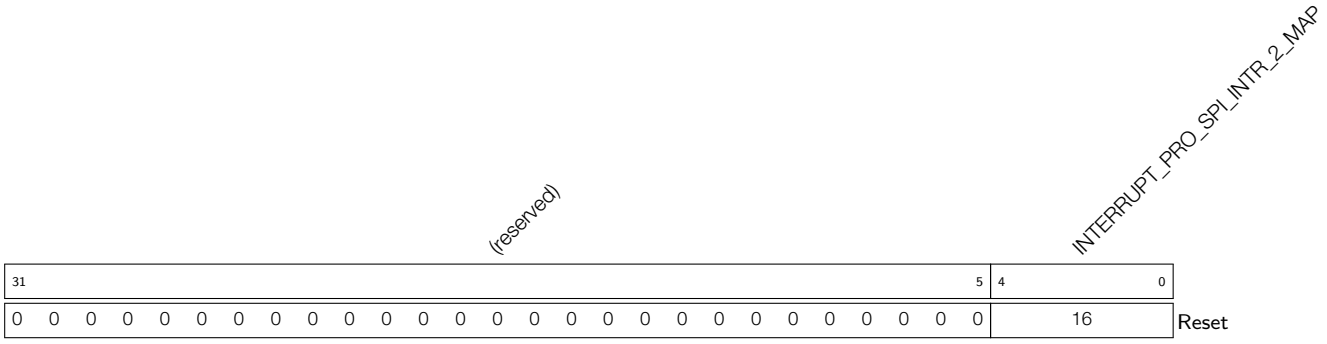
INTERRUPT_PRO_CPU_INTR_FROM_CPU_3_MAP This register is used to map CPU_INTR_FROM_CPU_3 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.33: INTERRUPT_PRO_SPI_INTR_1_MAP_REG (0x0080)



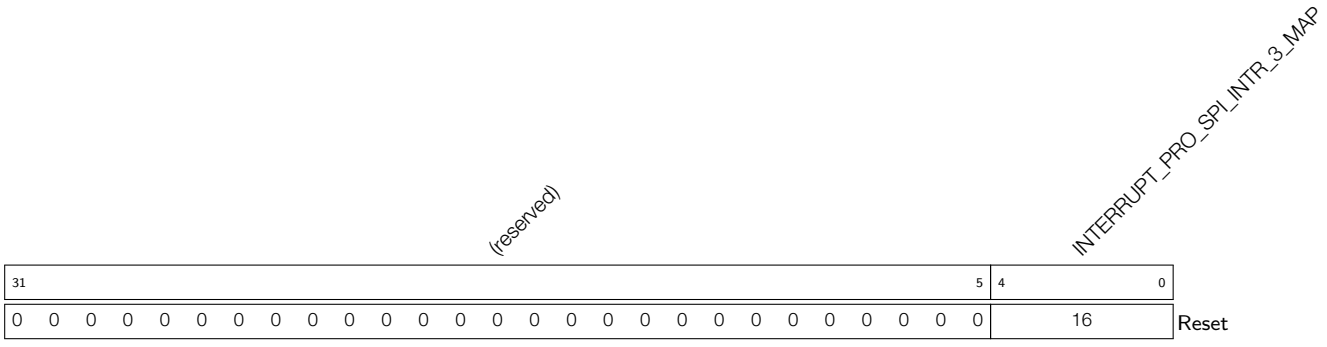
INTERRUPT_PRO_SPI_INTR_1_MAP This register is used to map SPI_INTR_1 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.34: INTERRUPT_PRO_SPI_INTR_2_MAP_REG (0x0084)



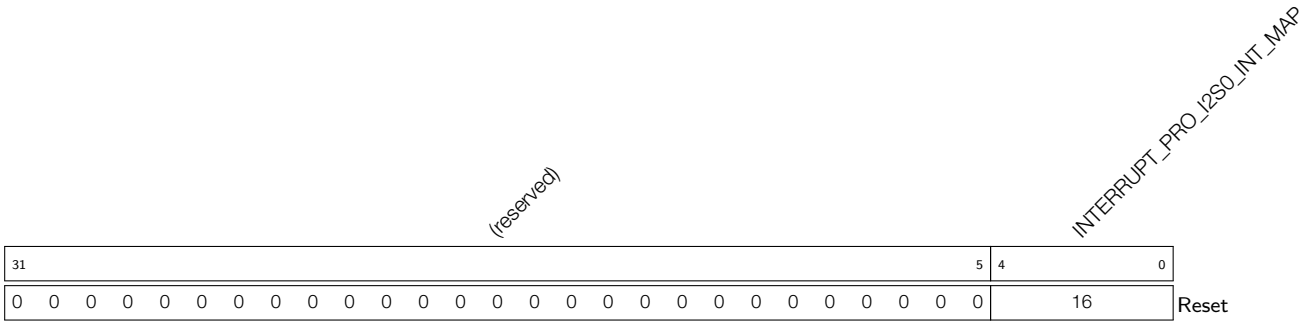
INTERRUPT_PRO_SPI_INTR_2_MAP This register is used to map SPI_INTR_2 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.35: INTERRUPT_PRO_SPI_INTR_3_MAP_REG (0x0088)



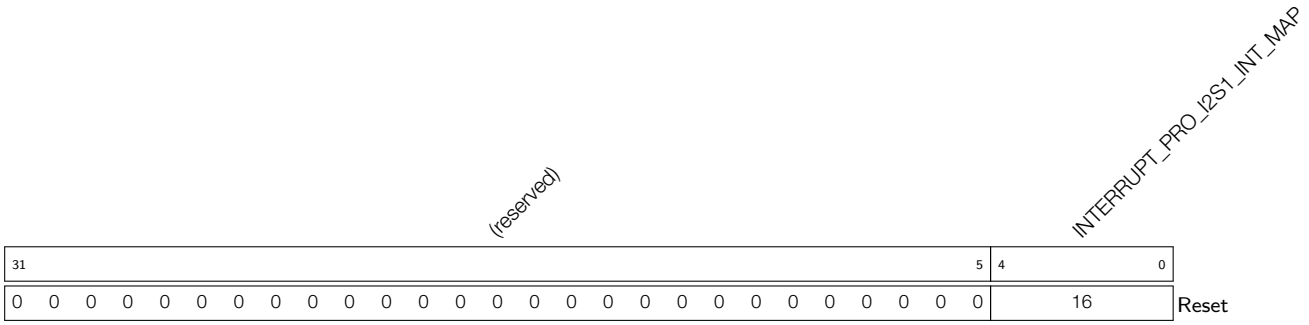
INTERRUPT_PRO_SPI_INTR_3_MAP This register is used to map SPI_INTR_3 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.36: INTERRUPT_PRO_I2S0_INT_MAP_REG (0x008C)



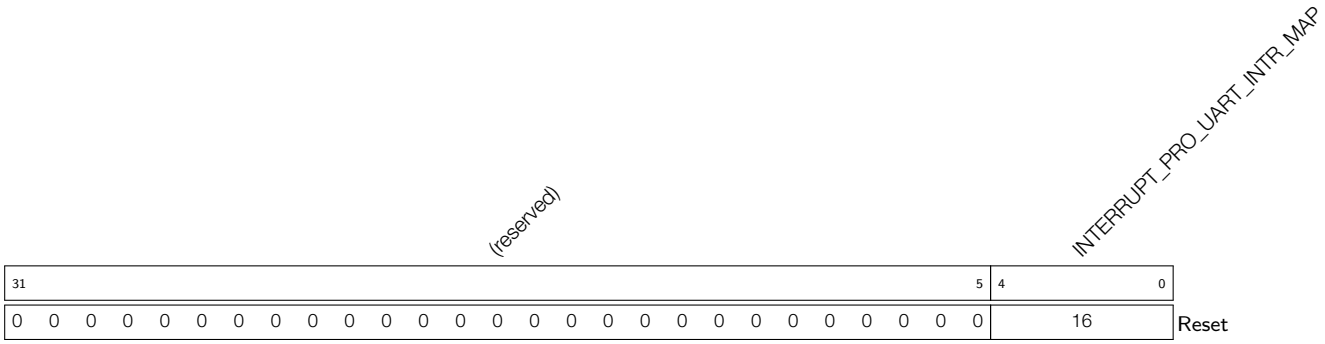
INTERRUPT_PRO_I2S0_INT_MAP This register is used to map I2S0_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.37: INTERRUPT_PRO_I2S1_INT_MAP_REG (0x0090)



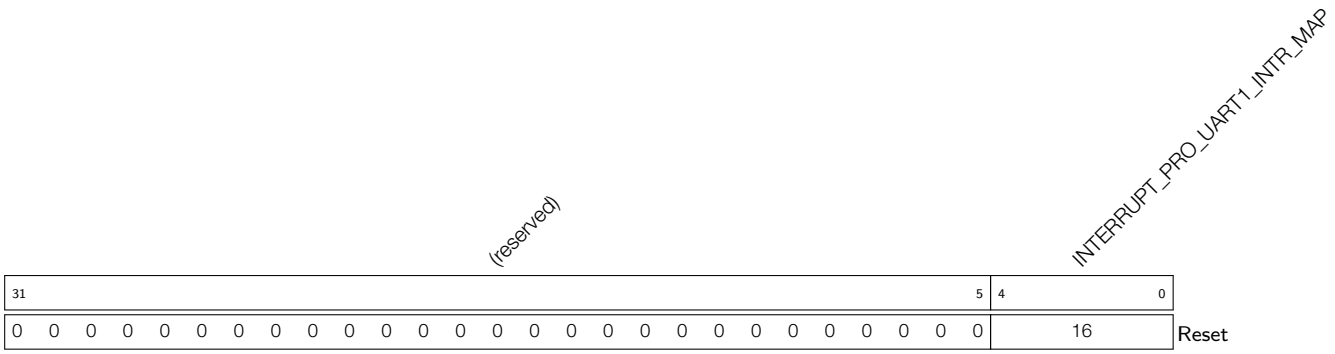
INTERRUPT_PRO_I2S1_INT_MAP This register is used to map I2S1_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.38: INTERRUPT_PRO_UART_INTR_MAP_REG (0x0094)



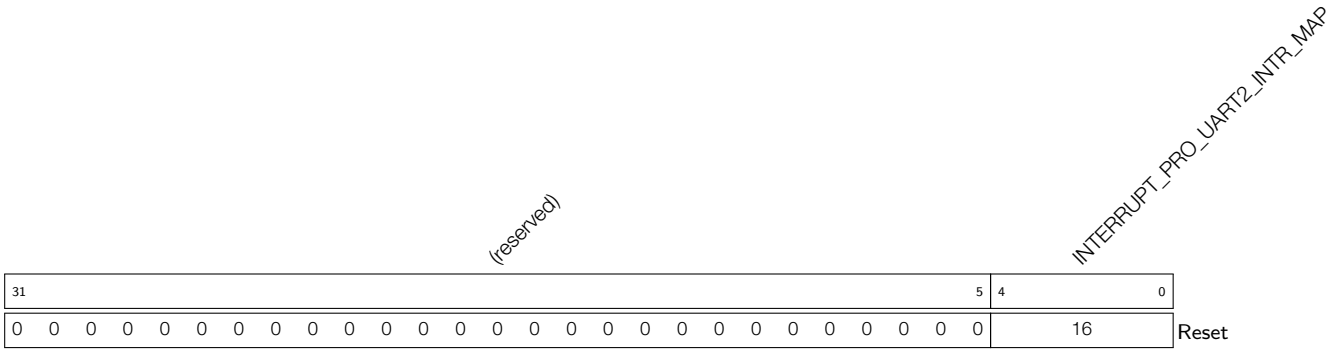
INTERRUPT_PRO_UART_INTR_MAP This register is used to map UART_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.39: INTERRUPT_PRO_UART1_INTR_MAP_REG (0x0098)



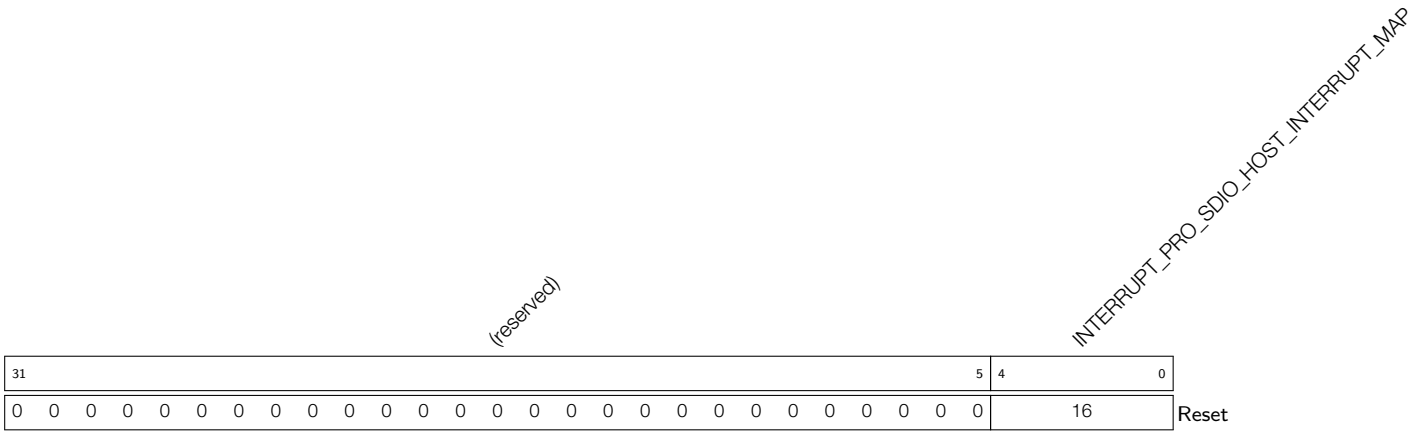
INTERRUPT_PRO_UART1_INTR_MAP This register is used to map UART1_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.40: INTERRUPT_PRO_UART2_INTR_MAP_REG (0x009C)



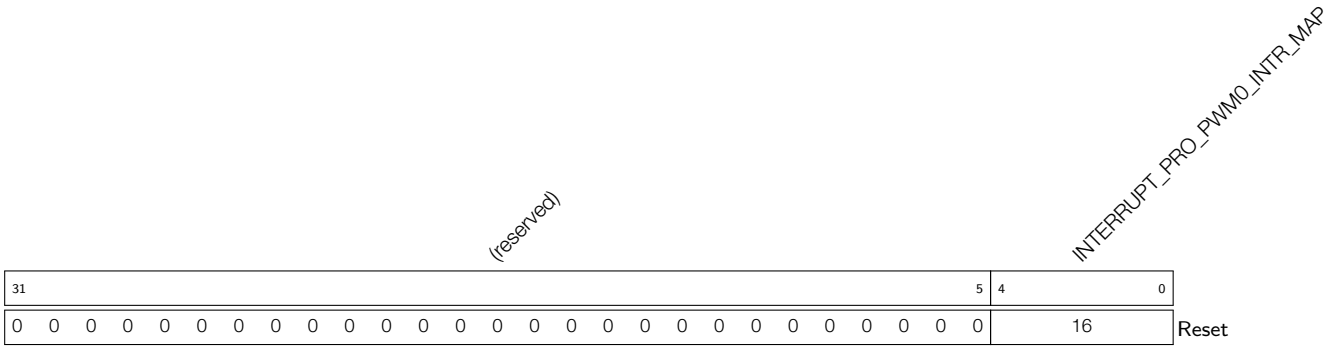
INTERRUPT_PRO_UART2_INTR_MAP This register is used to map UART2_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.41: INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP_REG (0x00A0)



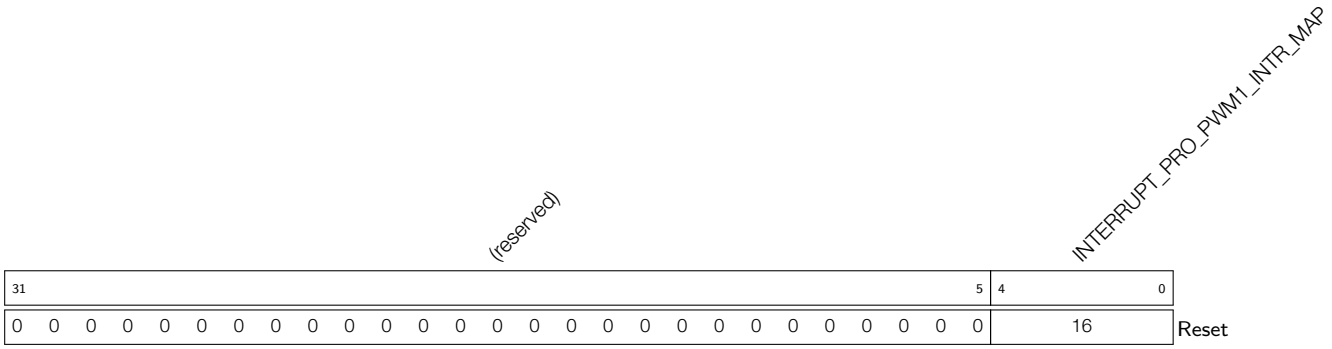
INTERRUPT_PRO_SDIO_HOST_INTERRUPT_MAP This register is used to map SDIO_HOST_INTERRUPT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.42: INTERRUPT_PRO_PWM0_INTR_MAP_REG (0x00A4)



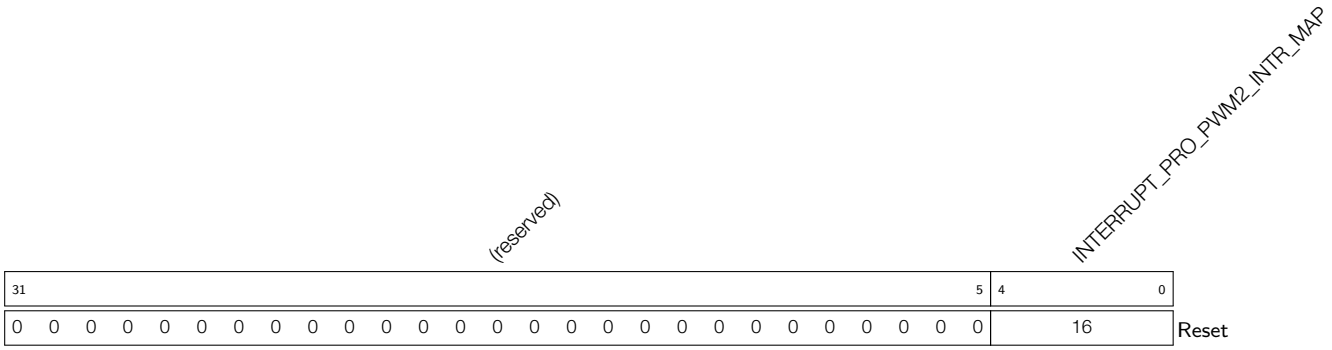
INTERRUPT_PRO_PWM0_INTR_MAP This register is used to map PWM0_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.43: INTERRUPT_PRO_PWM1_INTR_MAP_REG (0x00A8)



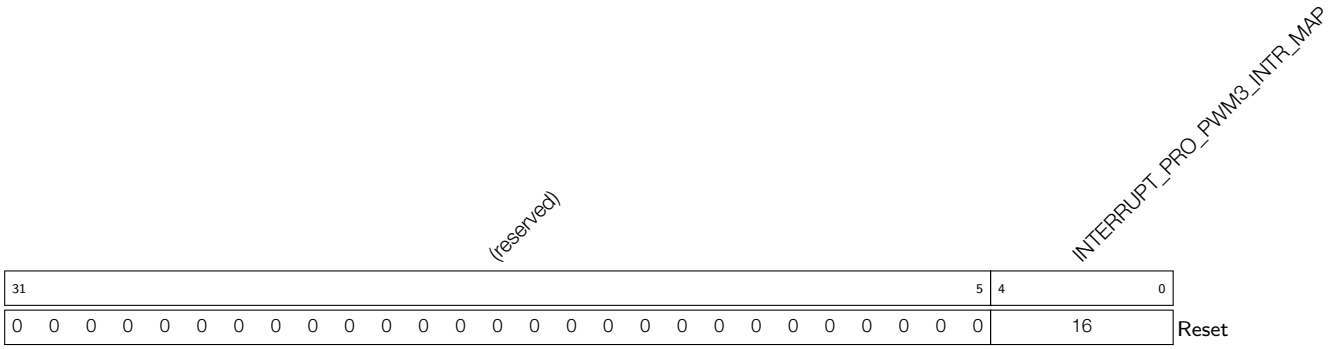
INTERRUPT_PRO_PWM1_INTR_MAP This register is used to map PWM1_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.44: INTERRUPT_PRO_PWM2_INTR_MAP_REG (0x00AC)



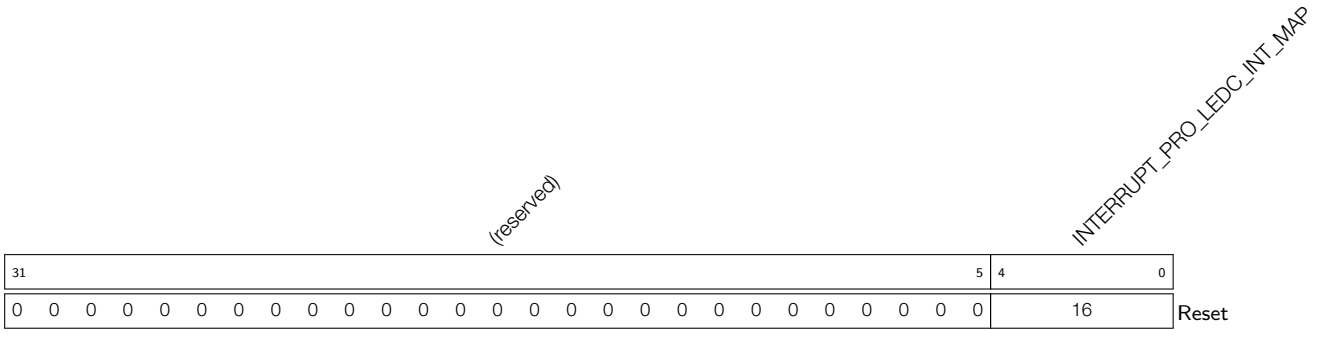
INTERRUPT_PRO_PWM2_INTR_MAP This register is used to map PWM2_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.45: INTERRUPT_PRO_PWM3_INTR_MAP_REG (0x00B0)



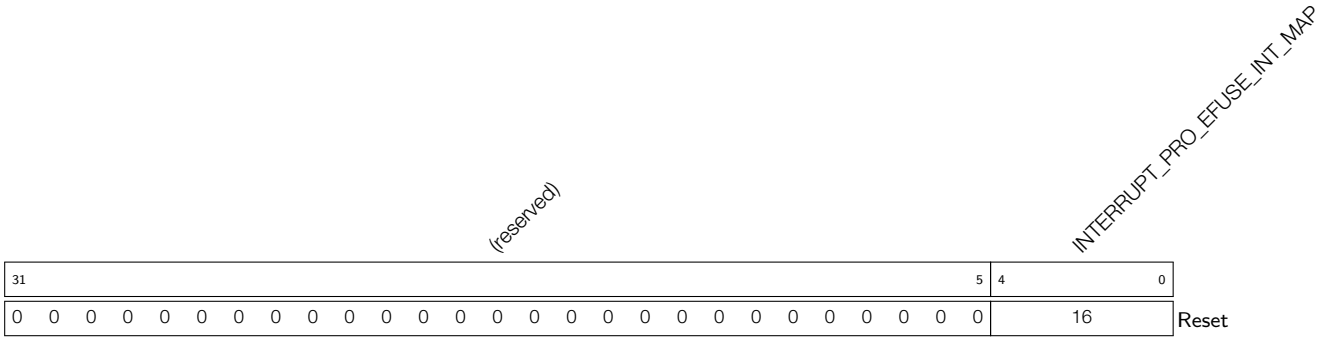
INTERRUPT_PRO_PWM3_INTR_MAP This register is used to map PWM3_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.46: INTERRUPT_PRO_LEDC_INT_MAP_REG (0x00B4)



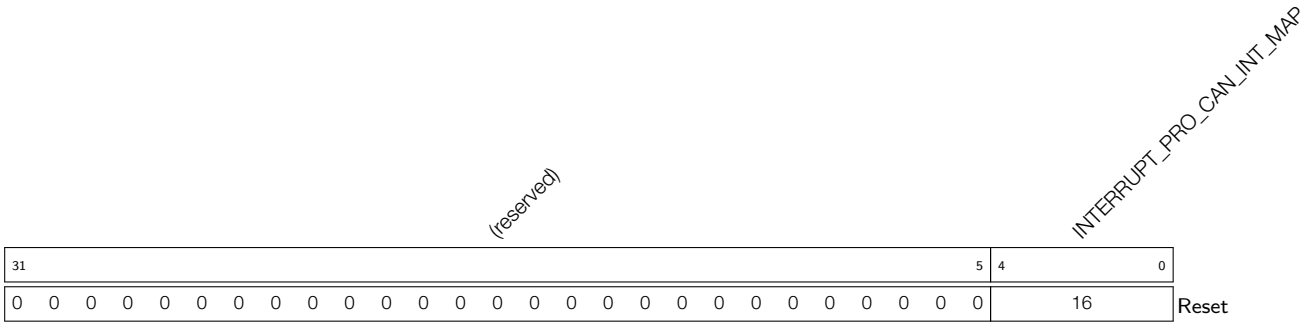
INTERRUPT_PRO_LEDC_INT_MAP This register is used to map LEDC_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.47: INTERRUPT_PRO_EFUSE_INT_MAP_REG (0x00B8)



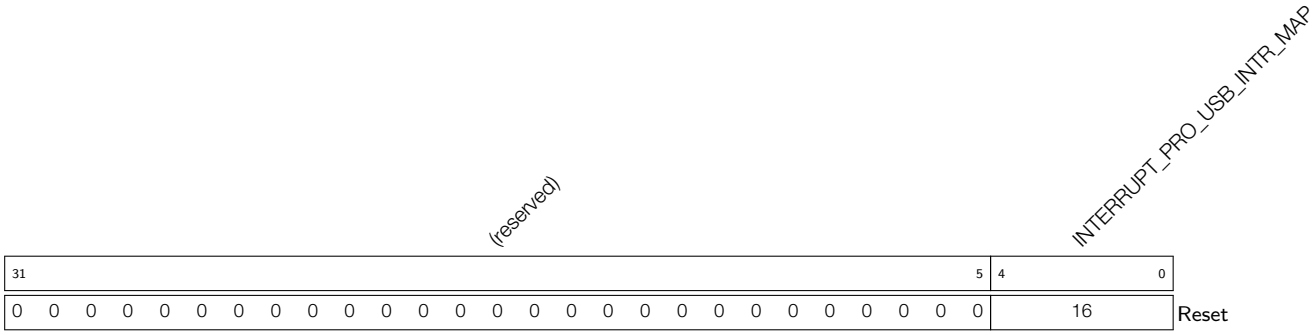
INTERRUPT_PRO_EFUSE_INT_MAP This register is used to map EFUSE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.48: INTERRUPT_PRO_CAN_INT_MAP_REG (0x00BC)



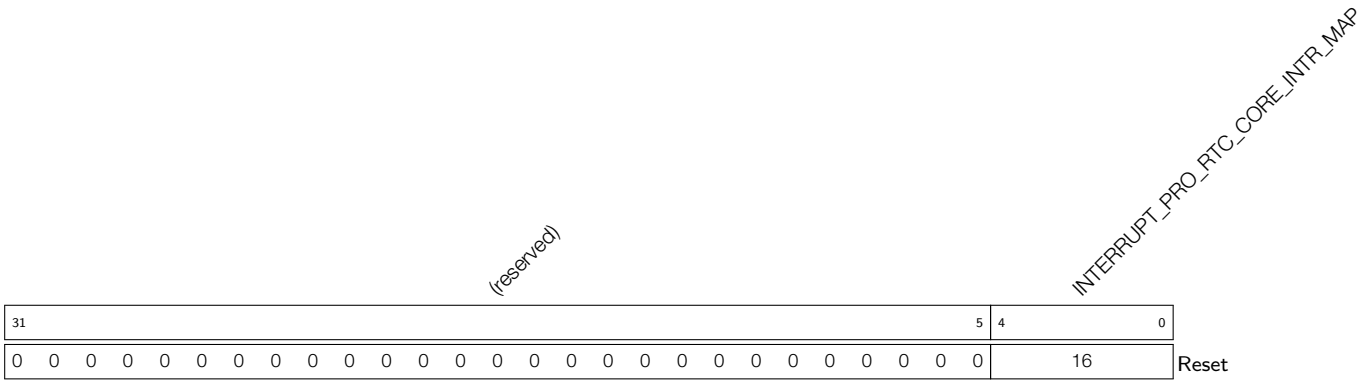
INTERRUPT_PRO_CAN_INT_MAP This register is used to map CAN_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.49: INTERRUPT_PRO_USB_INTR_MAP_REG (0x00C0)



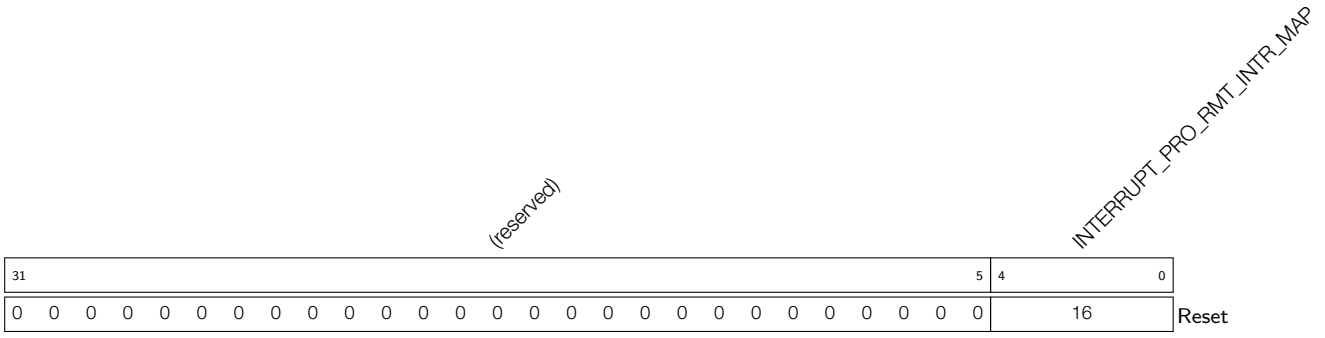
INTERRUPT_PRO_USB_INTR_MAP This register is used to map USB_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.50: INTERRUPT_PRO_RTC_CORE_INTR_MAP_REG (0x00C4)



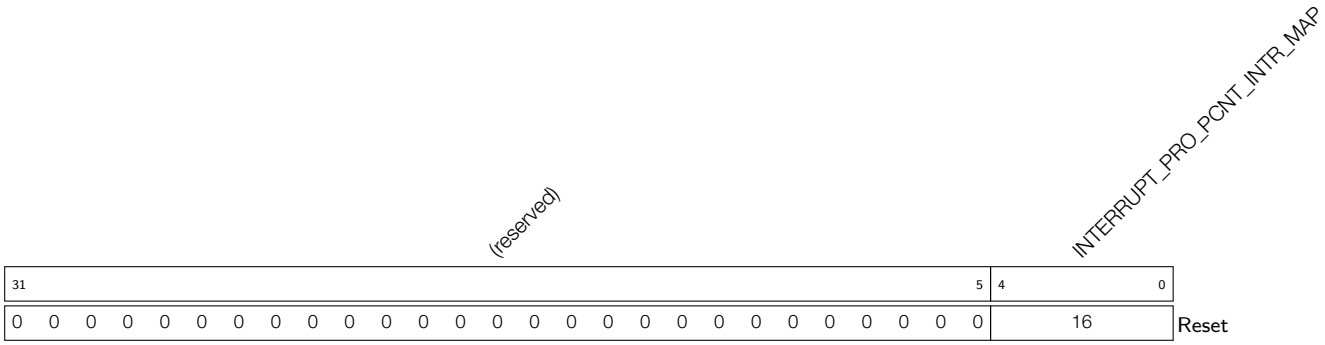
INTERRUPT_PRO_RTC_CORE_INTR_MAP This register is used to map RTC_CORE_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.51: INTERRUPT_PRO_RMT_INTR_MAP_REG (0x00C8)



INTERRUPT_PRO_RMT_INTR_MAP This register is used to map RMT_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.52: INTERRUPT_PRO_PCNT_INTR_MAP_REG (0x00CC)

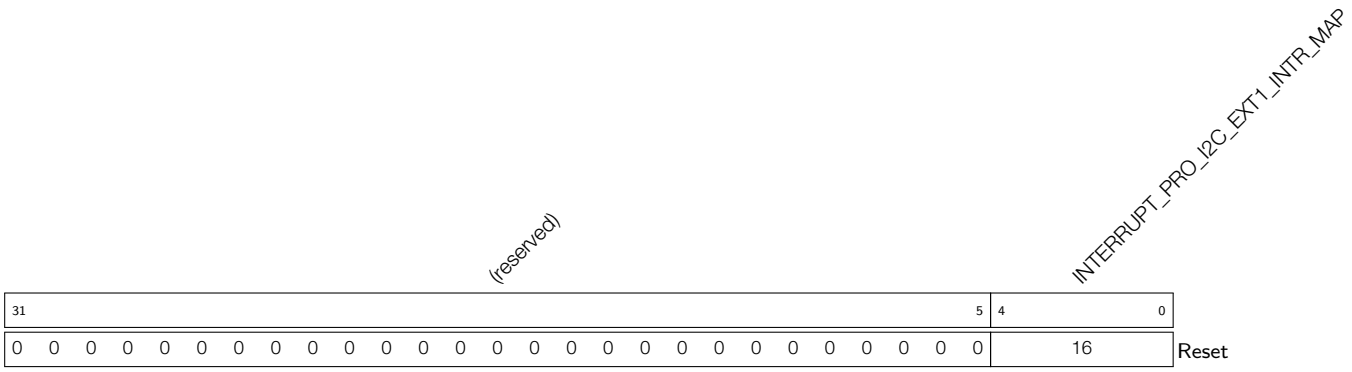


INTERRUPT_PRO_PCNT_INTR_MAP This register is used to map PCNT_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.53: INTERRUPT_PRO_I2C_EXT0_INTR_MAP_REG (0x00D0)

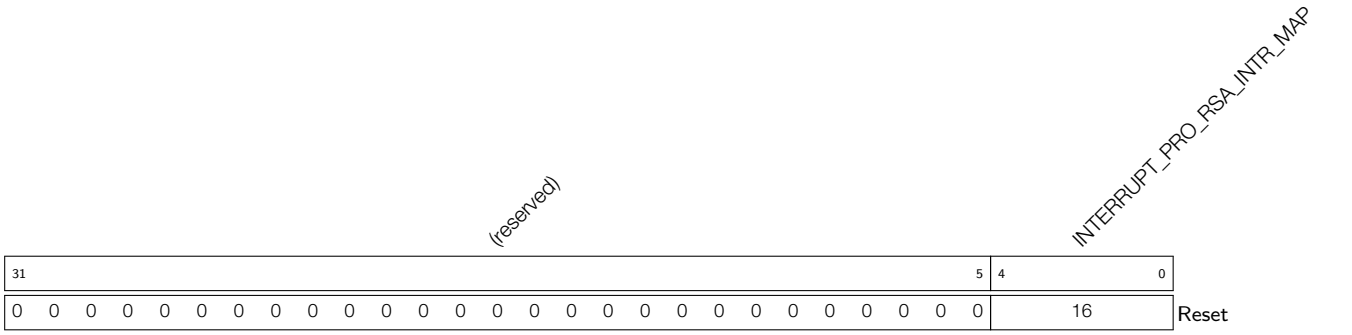
INTERRUPT_PRO_I2C_EXT0_INTR_MAP This register is used to map I2C_EXT0_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.54: INTERRUPT_PRO_I2C_EXT1_INTR_MAP_REG (0x00D4)



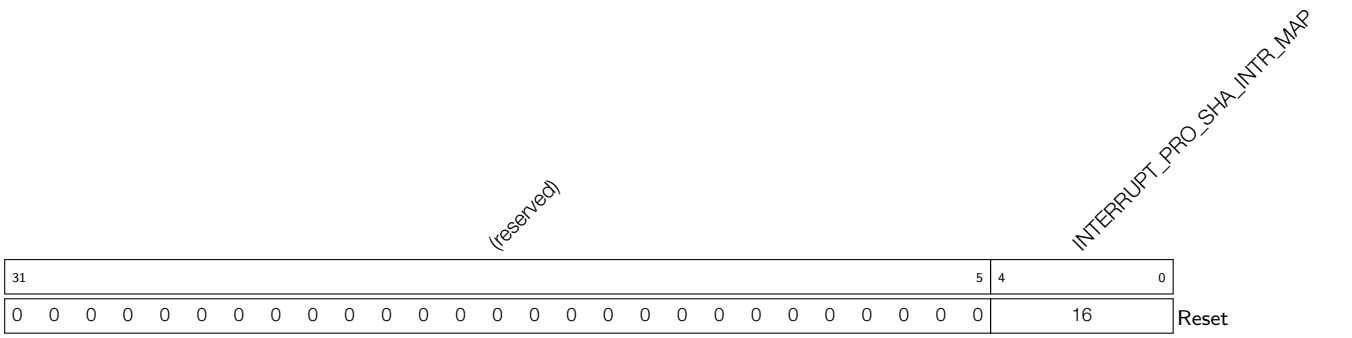
INTERRUPT_PRO_I2C_EXT1_INTR_MAP This register is used to map I2C_EXT1_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.55: INTERRUPT_PRO_RSA_INTR_MAP_REG (0x00D8)



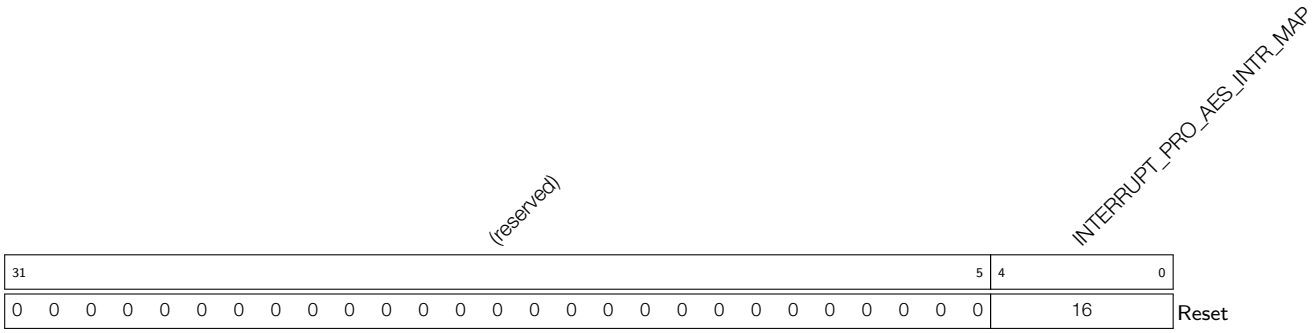
INTERRUPT_PRO_RSA_INTR_MAP This register is used to map RSA_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.56: INTERRUPT_PRO_SHA_INTR_MAP_REG (0x00DC)



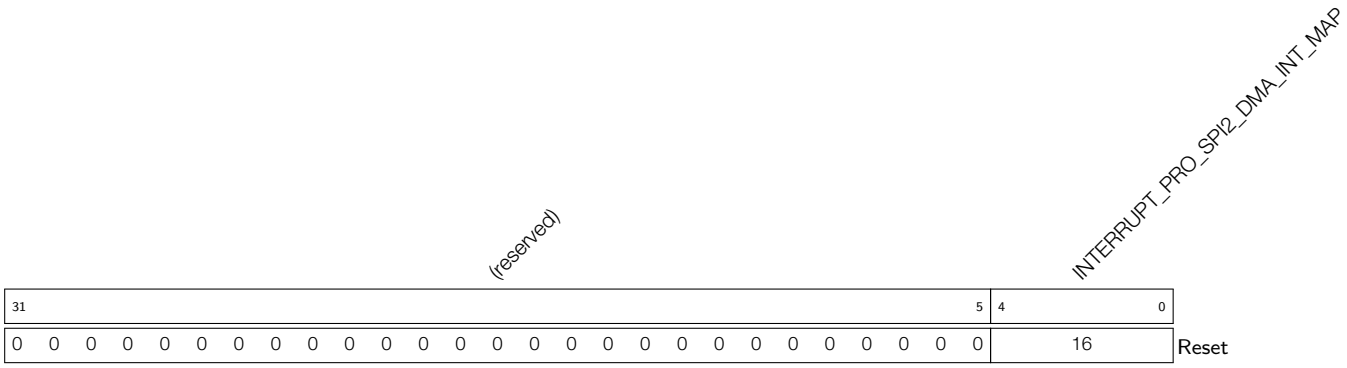
INTERRUPT_PRO_SHA_INTR_MAP This register is used to map SHA_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.57: INTERRUPT_PRO_AES_INTR_MAP_REG (0x00E0)



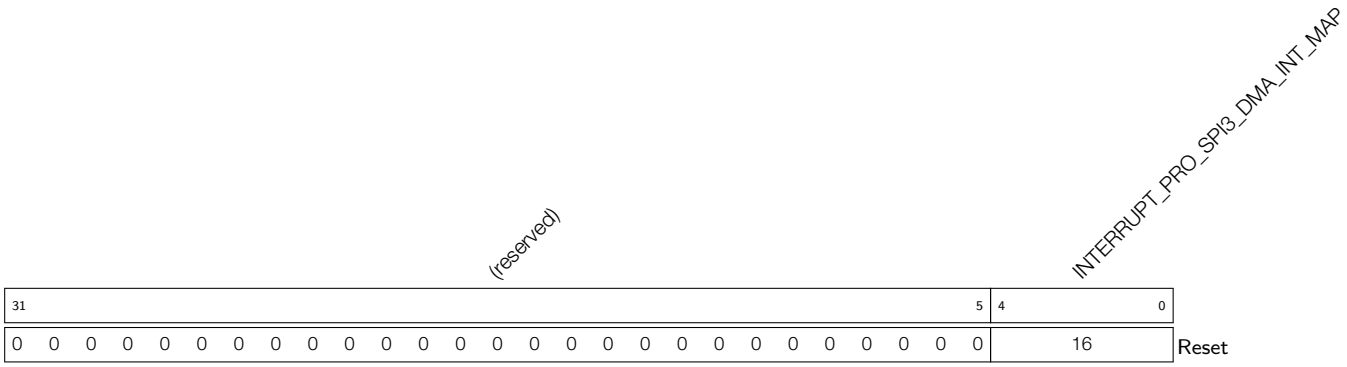
INTERRUPT_PRO_AES_INTR_MAP This register is used to map AES_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.58: INTERRUPT_PRO_SPI2_DMA_INT_MAP_REG (0x00E4)



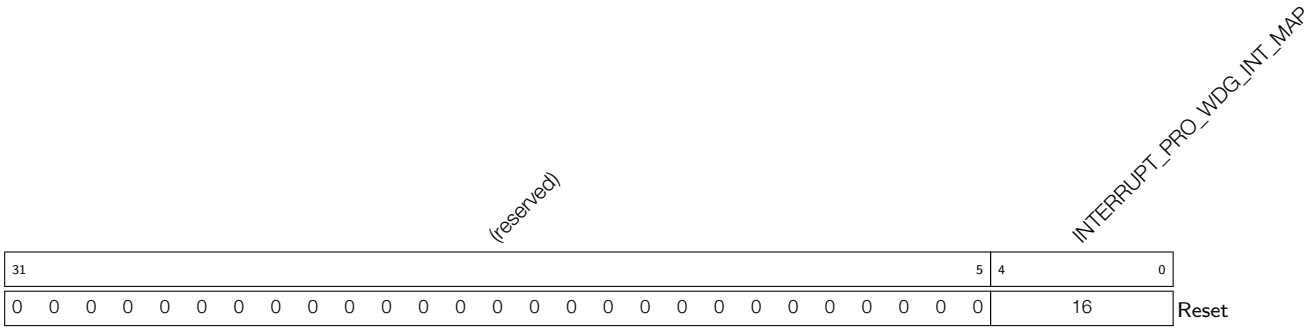
INTERRUPT_PRO_SPI2_DMA_INT_MAP This register is used to map SPI2_DMA_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.59: INTERRUPT_PRO_SPI3_DMA_INT_MAP_REG (0x00E8)



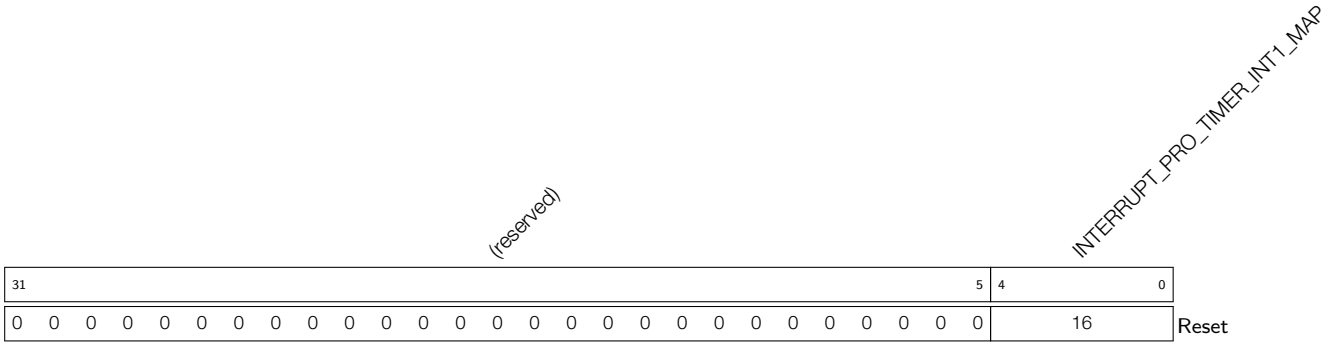
INTERRUPT_PRO_SPI3_DMA_INT_MAP This register is used to map SPI3_DMA_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.60: INTERRUPT_PRO_WDG_INT_MAP_REG (0x00EC)



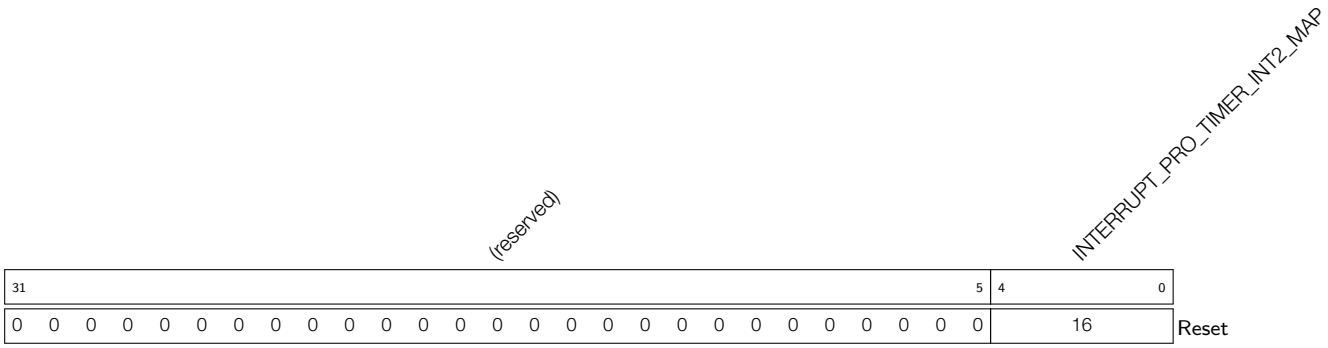
INTERRUPT_PRO_WDG_INT_MAP This register is used to map WDG_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.61: INTERRUPT_PRO_TIMER_INT1_MAP_REG (0x00F0)



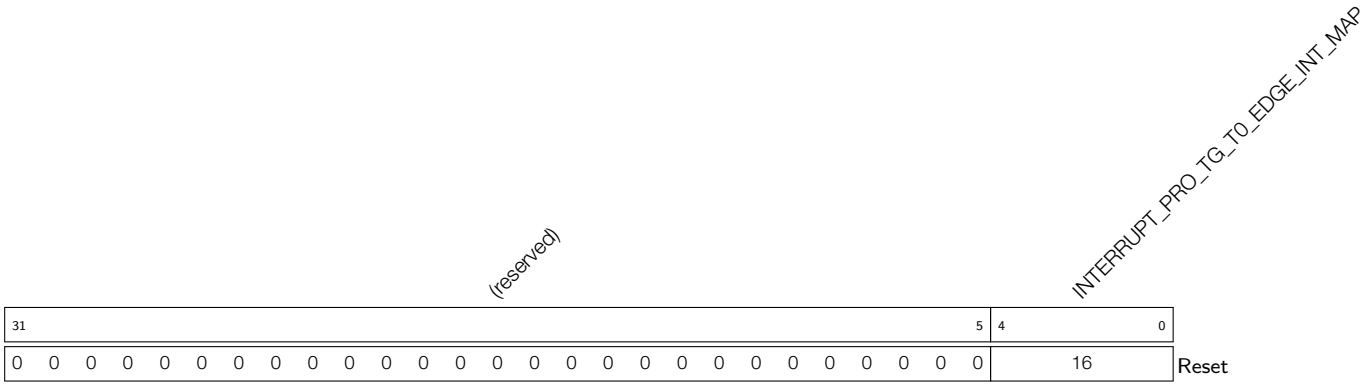
INTERRUPT_PRO_TIMER_INT1_MAP This register is used to map TIMER_INT1 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.62: INTERRUPT_PRO_TIMER_INT2_MAP_REG (0x00F4)



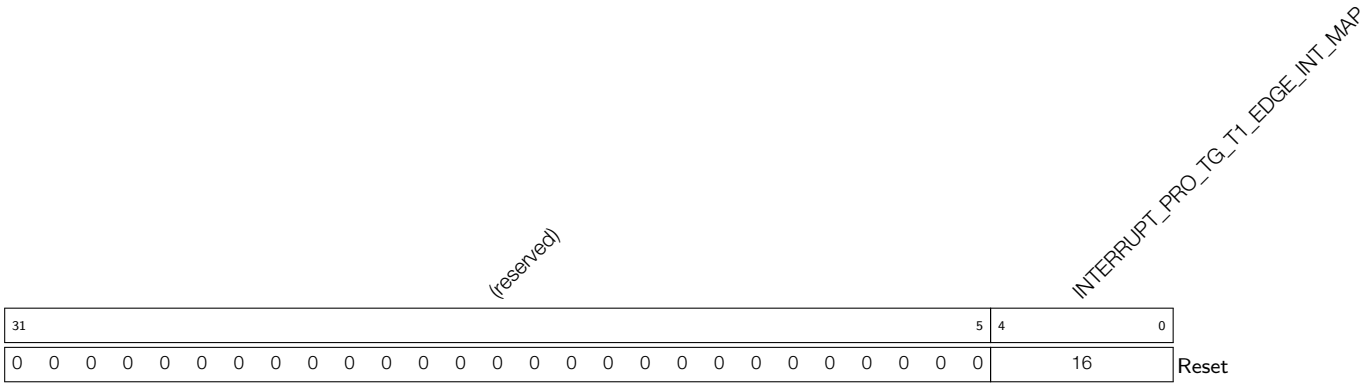
INTERRUPT_PRO_TIMER_INT2_MAP This register is used to map TIMER_INT2 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.63: INTERRUPT_PRO_TG_T0_EDGE_INT_MAP_REG (0x00F8)



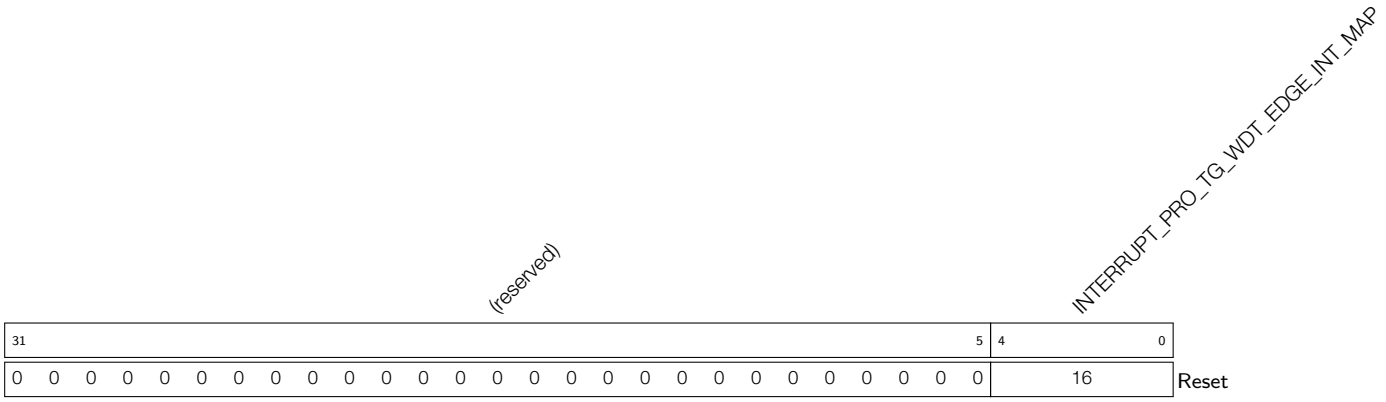
INTERRUPT_PRO_TG_T0_EDGE_INT_MAP This register is used to map TG_T0_EDGE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.64: INTERRUPT_PRO_TG_T1_EDGE_INT_MAP_REG (0x00FC)



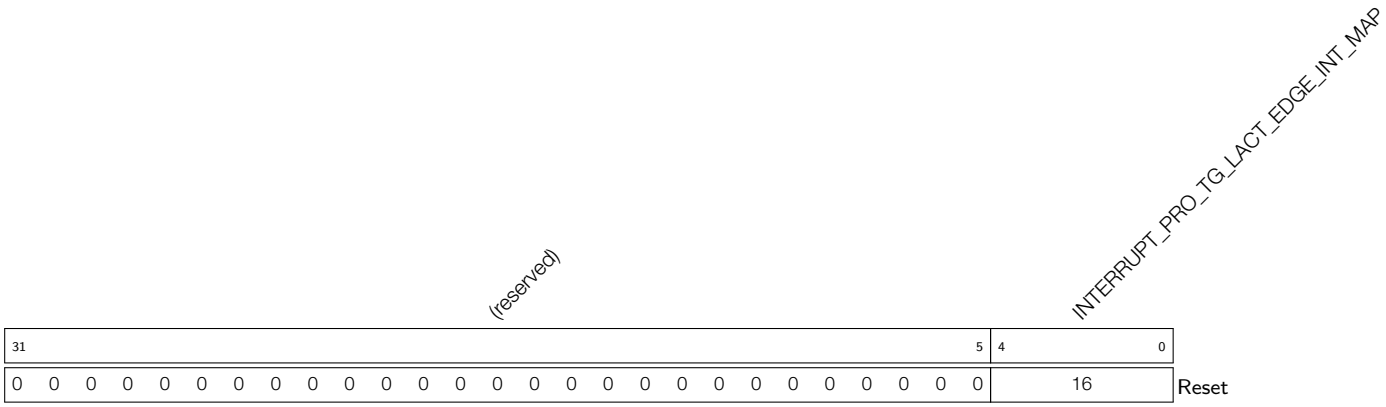
INTERRUPT_PRO_TG_T1_EDGE_INT_MAP This register is used to map TG_T1_EDGE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.65: INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP_REG (0x0100)



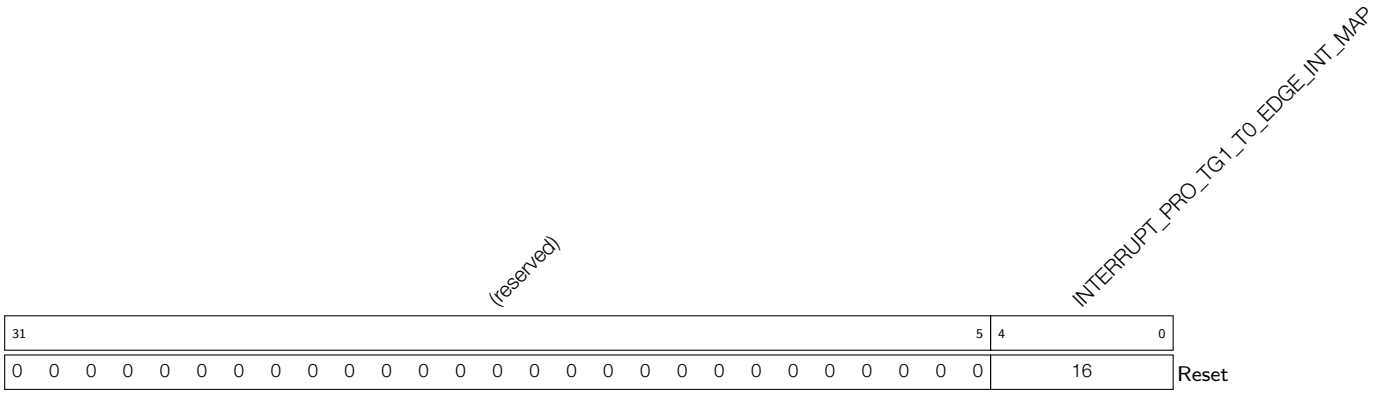
INTERRUPT_PRO_TG_WDT_EDGE_INT_MAP This register is used to map TG_WDT_EDGE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.66: INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP_REG (0x0104)



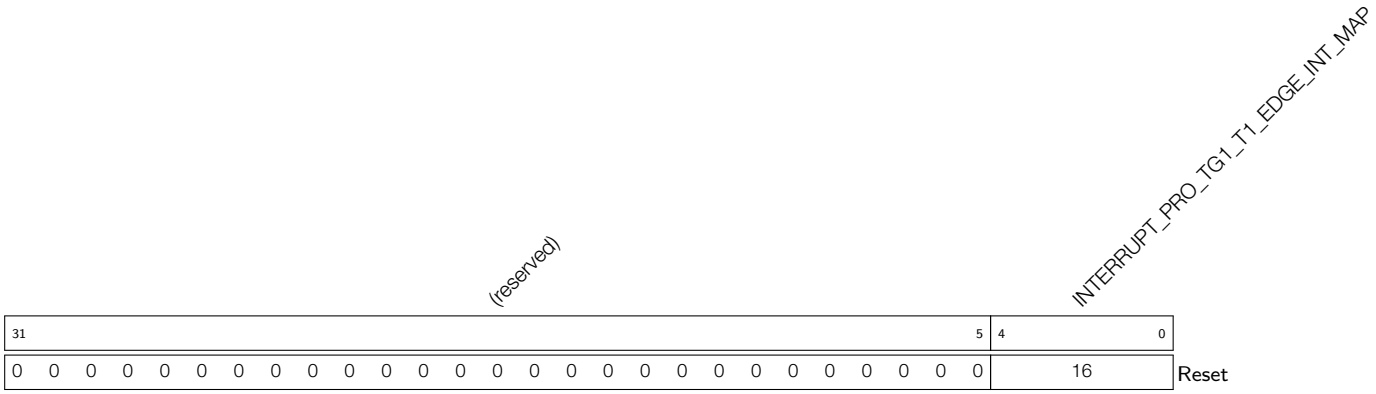
INTERRUPT_PRO_TG_LACT_EDGE_INT_MAP This register is used to map TG_LACT_EDGE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.67: INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP_REG (0x0108)



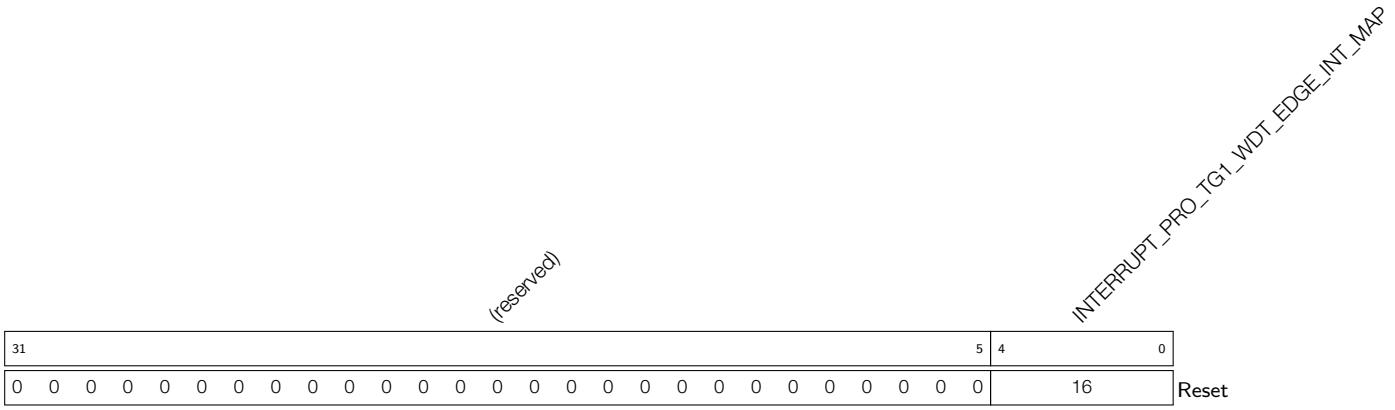
INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP This register is used to map TG1_T0_EDGE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.68: INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP_REG (0x010C)



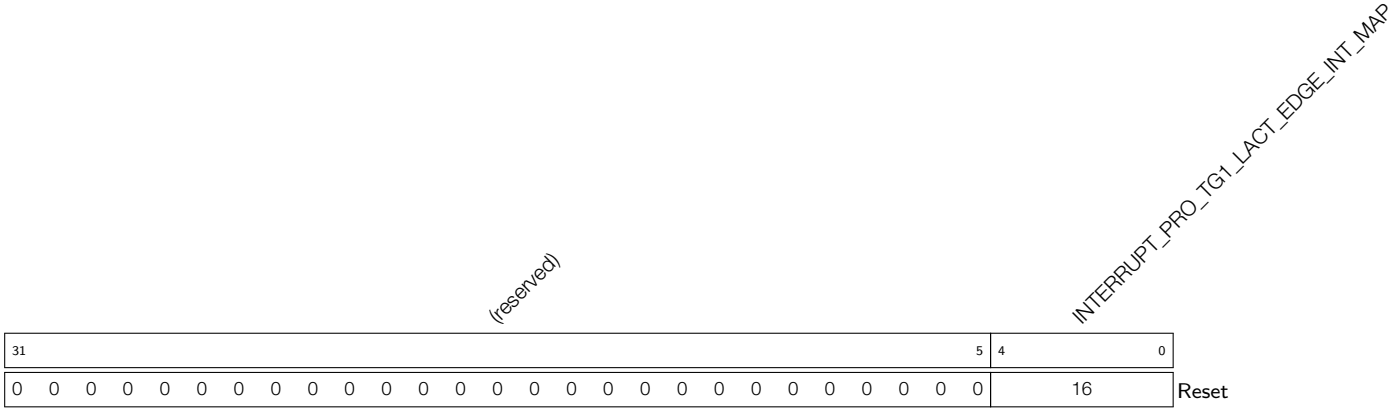
INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP This register is used to map TG1_T1_EDGE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.69: INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP_REG (0x0110)



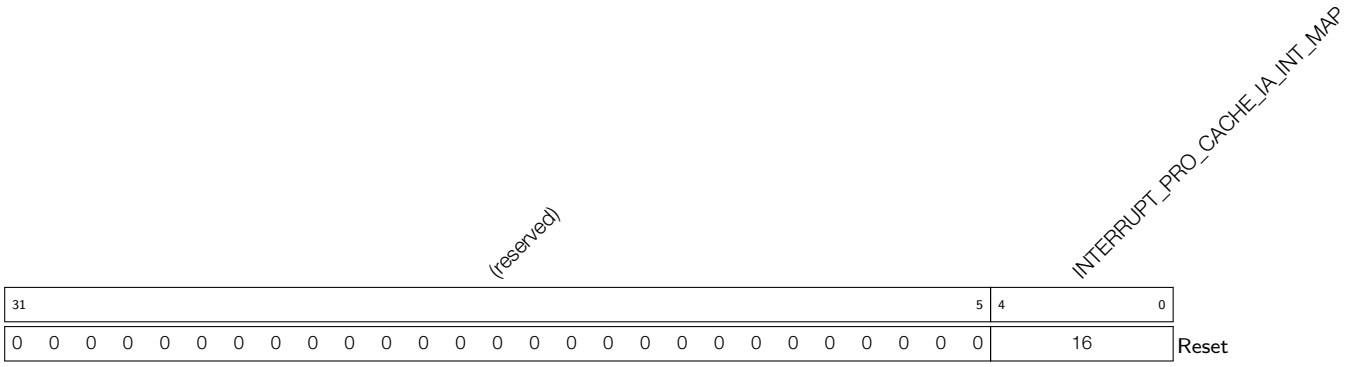
INTERRUPT_PRO_TG1_WDT_EDGE_INT_MAP This register is used to map TG1_WDT_EDGE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.70: INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP_REG (0x0114)



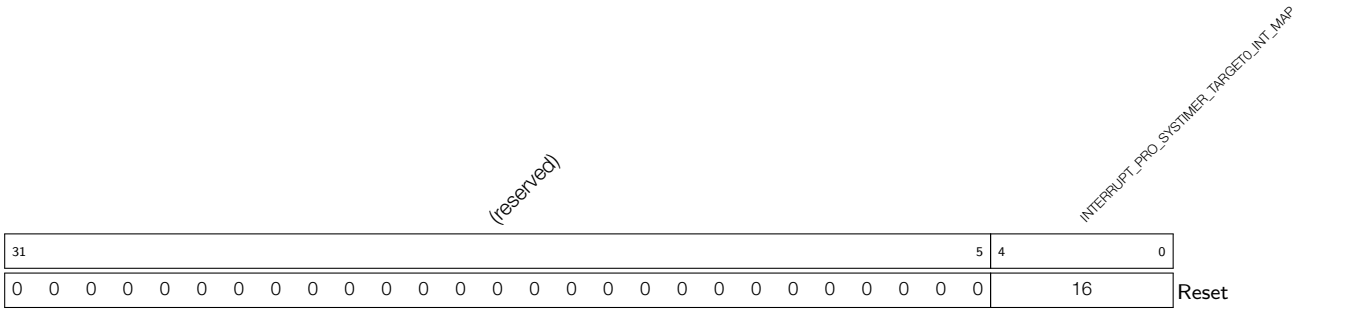
INTERRUPT_PRO_TG1_LACT_EDGE_INT_MAP This register is used to map TG1_LACT_EDGE_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.71: INTERRUPT_PRO_CACHE_IA_INT_MAP_REG (0x0118)



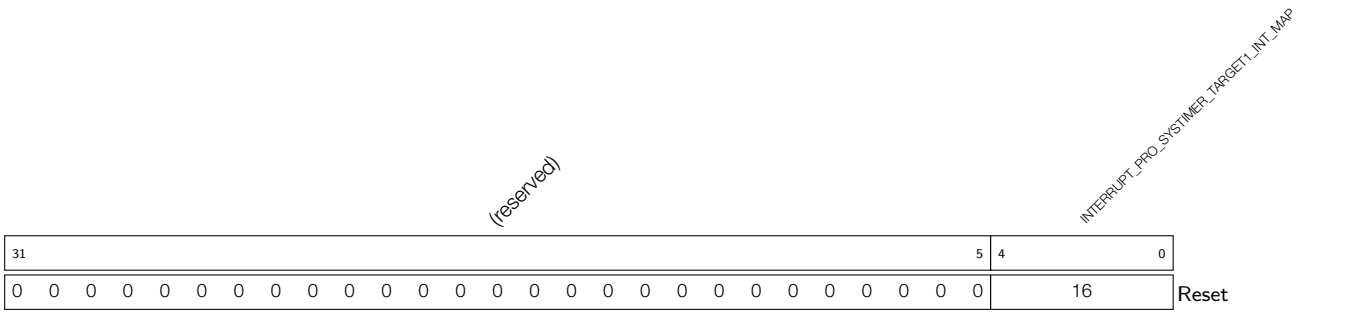
INTERRUPT_PRO_CACHE_IA_INT_MAP This register is used to map CACHE_IA_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.72: INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP_REG (0x011C)



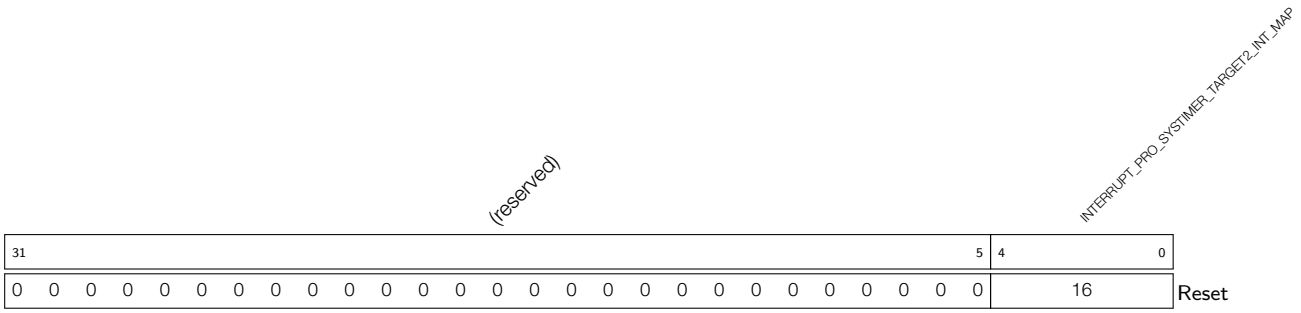
INTERRUPT_PRO_SYSTIMER_TARGET0_INT_MAP This register is used to map SYSTIMER_TARGET0_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.73: INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP_REG (0x0120)



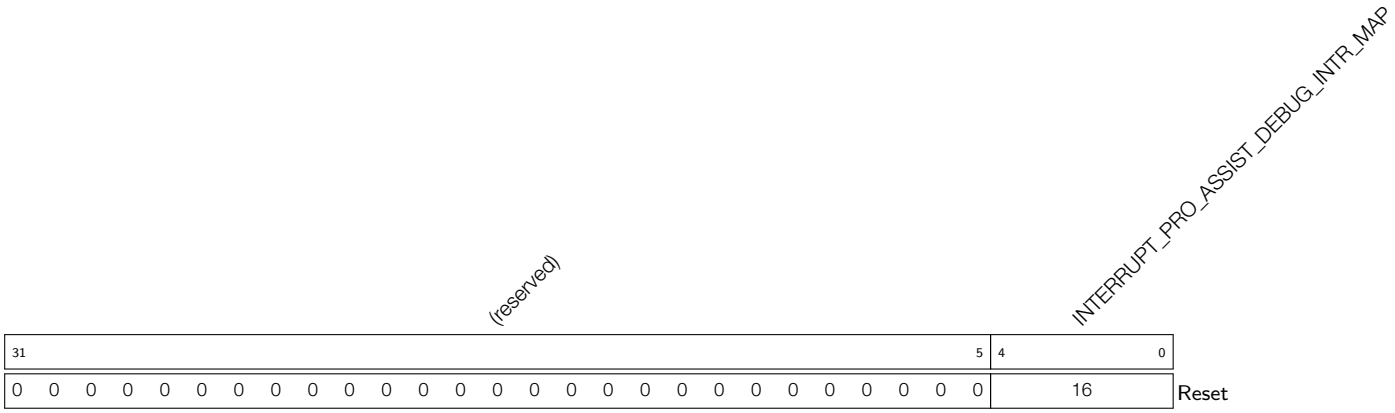
INTERRUPT_PRO_SYSTIMER_TARGET1_INT_MAP This register is used to map SYSTIMER_TARGET1_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.74: INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP_REG (0x0124)



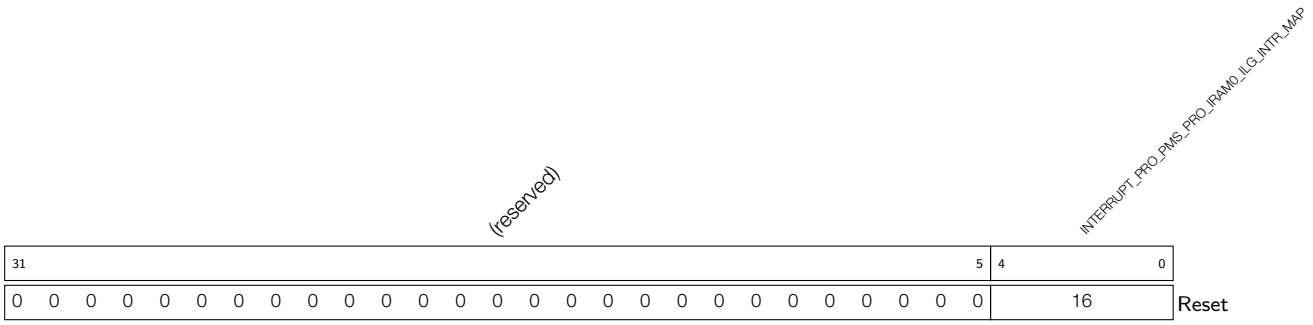
INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP This register is used to map SYSTIMER_TARGET2_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.75: INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP_REG (0x0128)



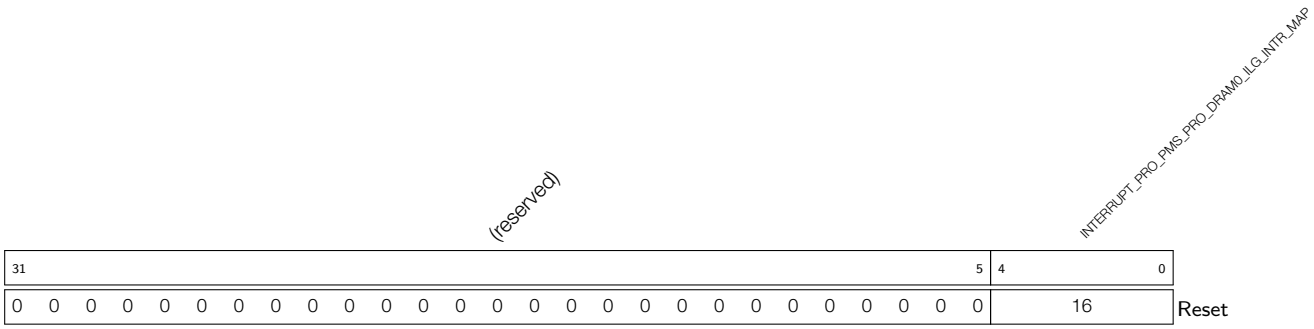
INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP This register is used to map ASSIST_DEBUG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.76: INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP_REG (0x012C)



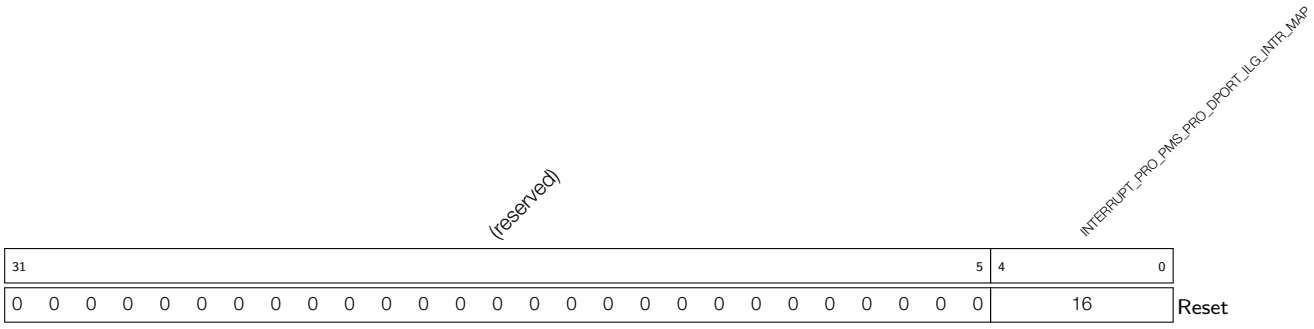
INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP This register is used to map PMS_PRO_IRAM0_ILG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.77: INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP_REG (0x0130)



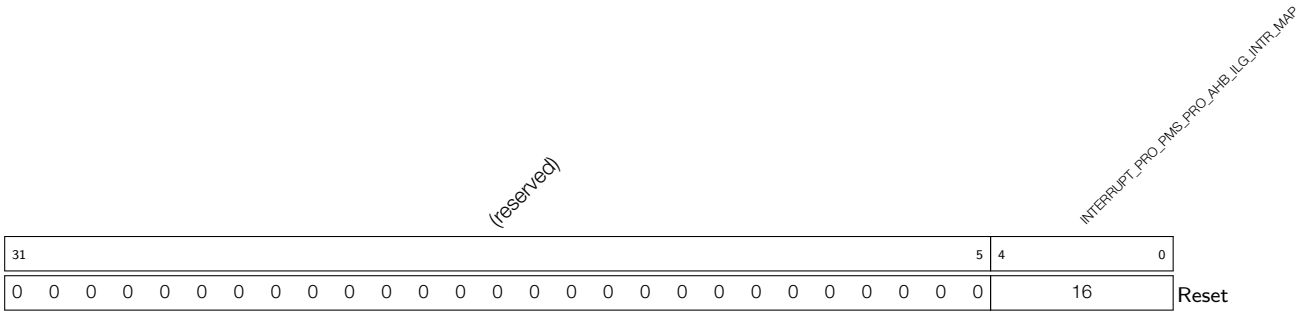
INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP This register is used to map PMS_PRO_DRAM0_ILG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.78: INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP_REG (0x0134)



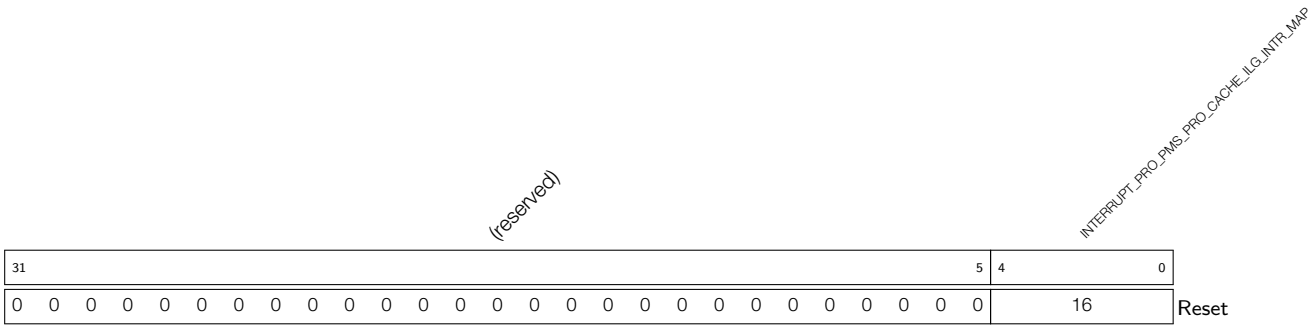
INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP This register is used to map PMS_PRO_DPORT_ILG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.79: INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP_REG (0x0138)



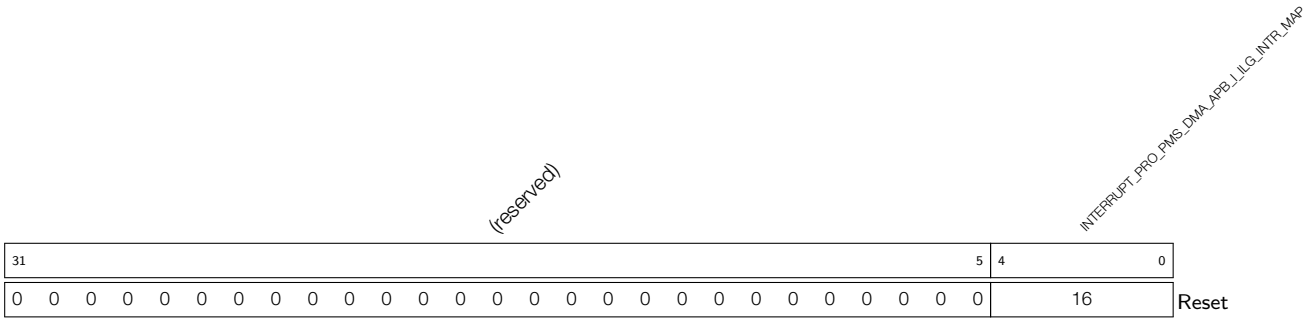
INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP This register is used to map PMS_PRO_AHB_ILG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.80: INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP_REG (0x013C)



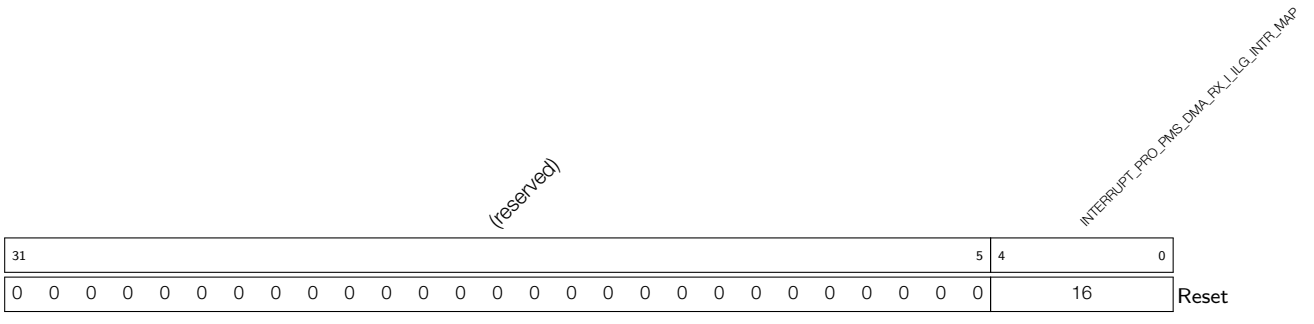
INTERRUPT_PRO_PMS_PRO_CACHE_ILG_INTR_MAP This register is used to map PMS_PRO_CACHE_ILG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.81: INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP_REG (0x0140)



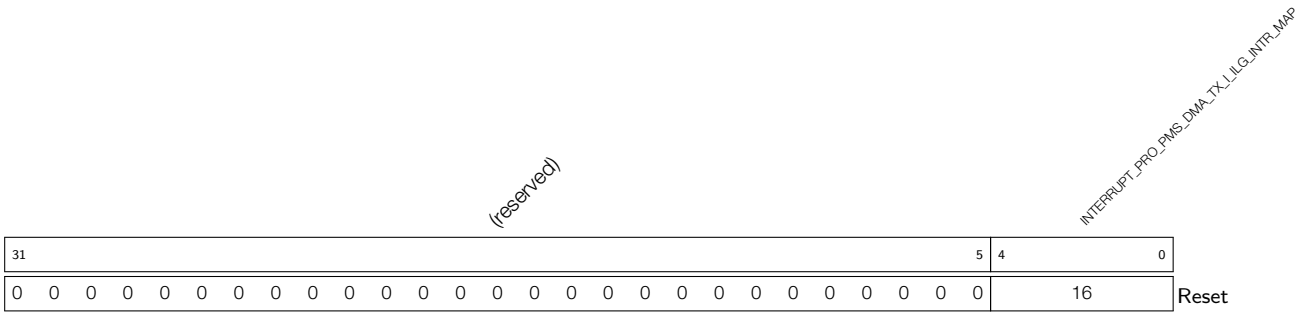
INTERRUPT_PRO_PMS_DMA_APB_I_ILG_INTR_MAP This register is used to map PMS_DMA_APB_I_ILG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.82: INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP_REG (0x0144)



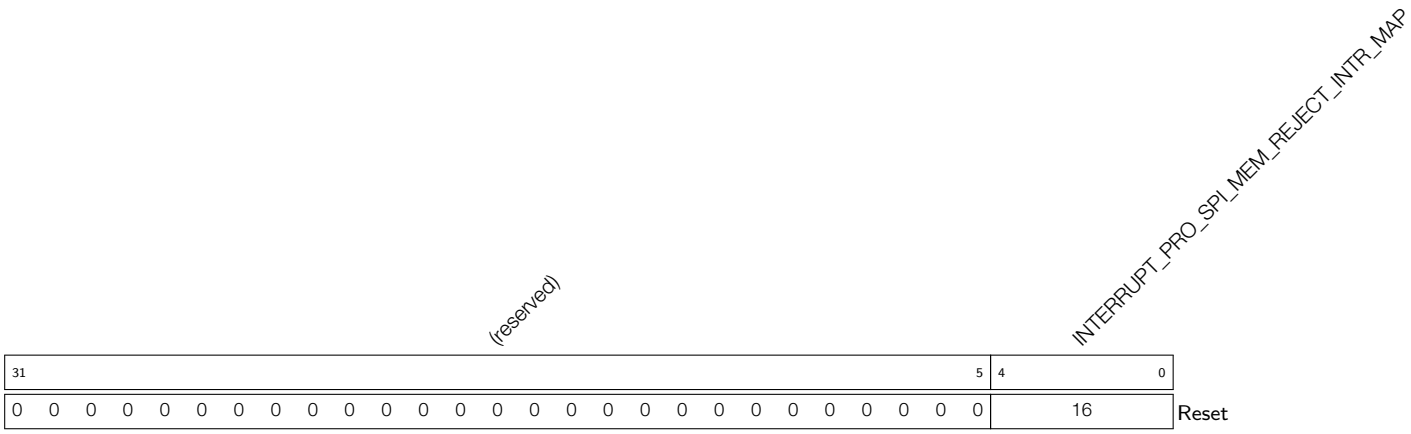
INTERRUPT_PRO_PMS_DMA_RX_I_ILG_INTR_MAP This register is used to map PMS_DMA_RX_I_ILG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.83: INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP_REG (0x0148)



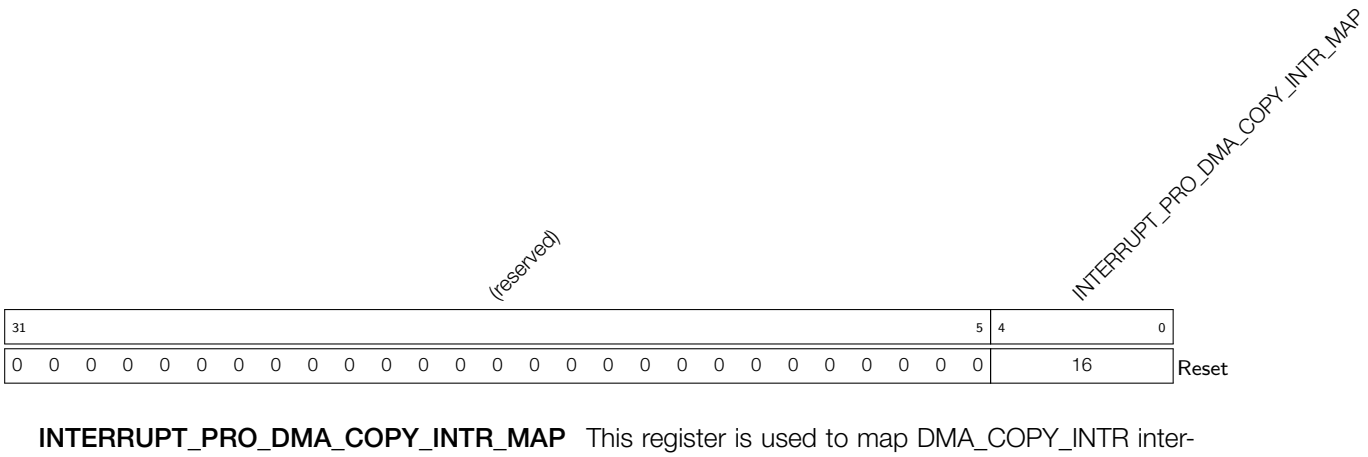
INTERRUPT_PRO_PMS_DMA_TX_I_ILG_INTR_MAP This register is used to map PMS_DMA_TX_I_ILG_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.84: INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP_REG (0x014C)



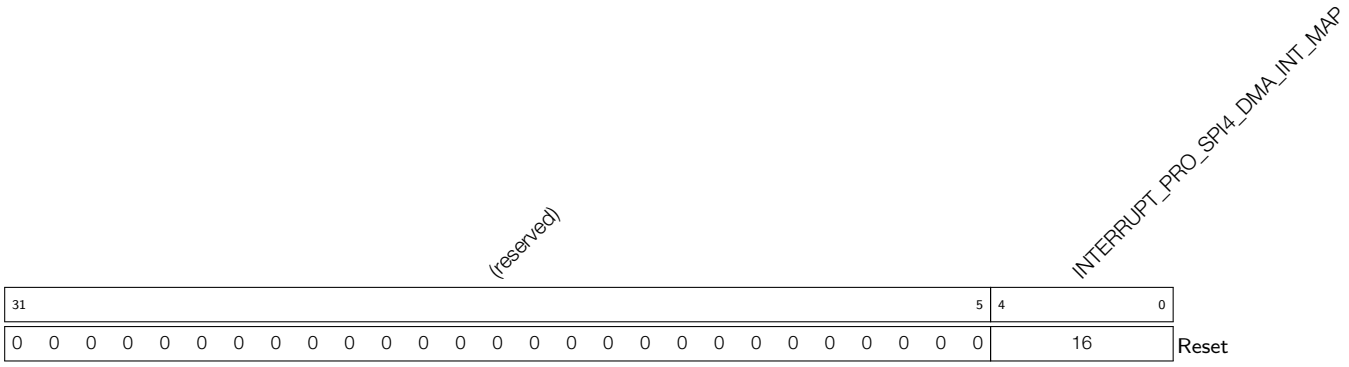
INTERRUPT_PRO_SPI_MEM_REJECT_INTR_MAP This register is used to map SPI_MEM_REJECT_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.85: INTERRUPT_PRO_DMA_COPY_INTR_MAP_REG (0x0150)



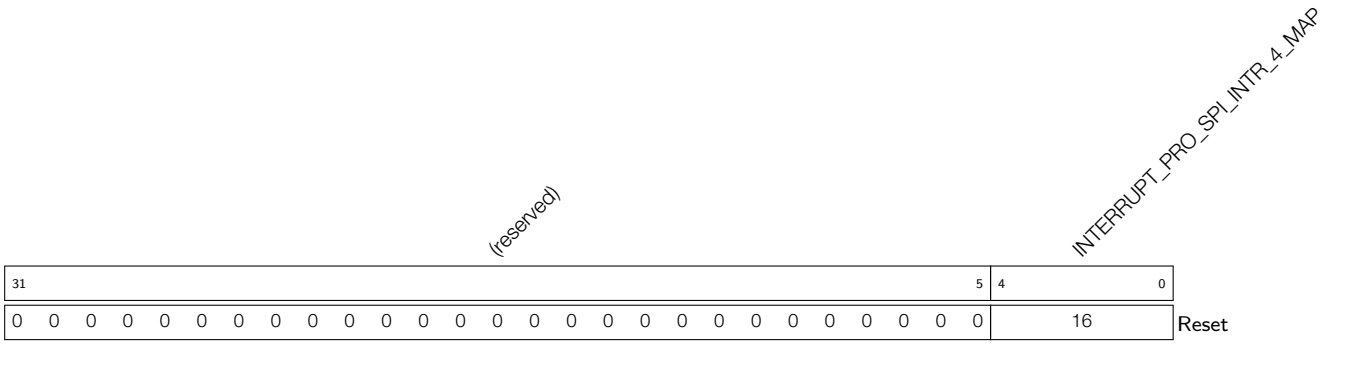
INTERRUPT_PRO_DMA_COPY_INTR_MAP This register is used to map DMA_COPY_INTR interrupt signal to one of the CPU interrupts. (R/W)

Register 4.86: INTERRUPT_PRO_SPI4_DMA_INT_MAP_REG (0x0154)



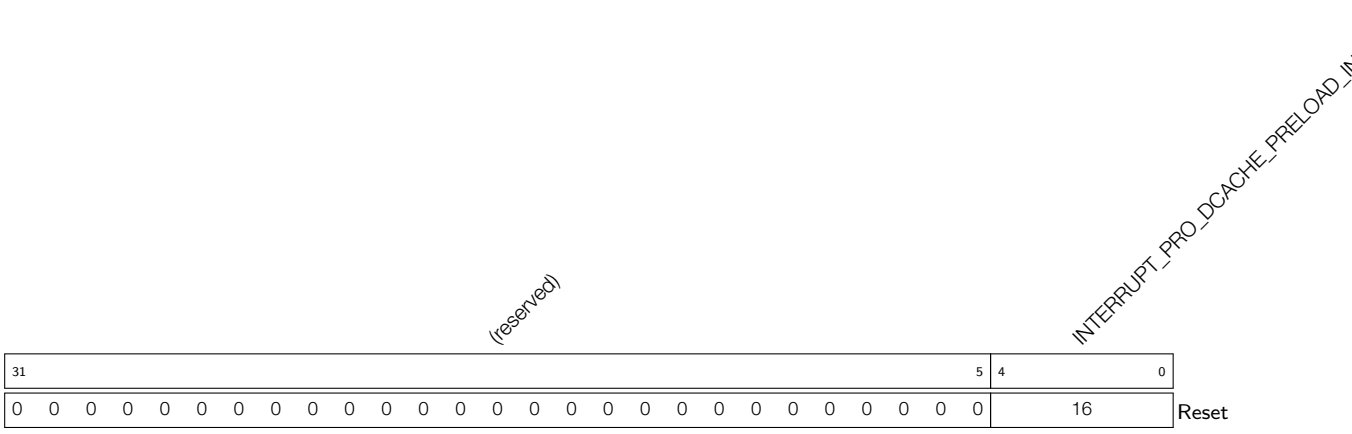
INTERRUPT_PRO_SPI4_DMA_INT_MAP This register is used to map SPI4_DMA_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.87: INTERRUPT_PRO_SPI_INTR_4_MAP_REG (0x0158)



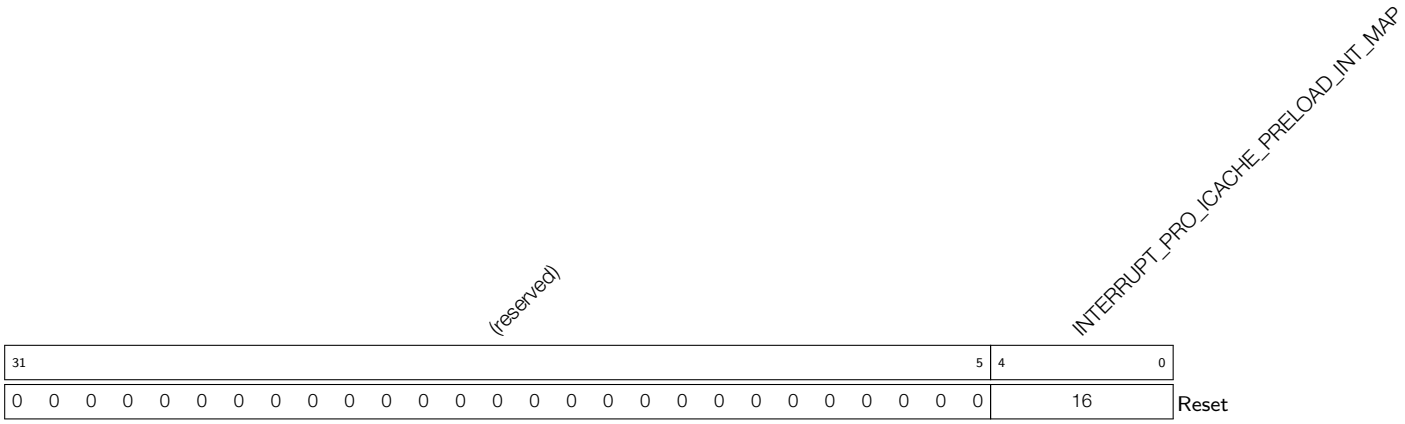
INTERRUPT_PRO_SPI_INTR_4_MAP This register is used to map SPI_INTR_4 interrupt signal to one of the CPU interrupts. (R/W)

Register 4.88: INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP_REG (0x015C)



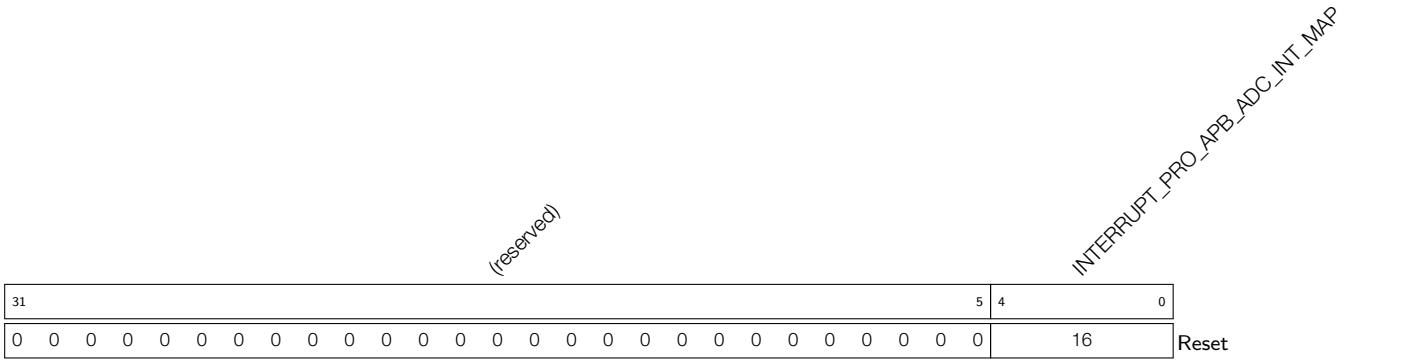
INTERRUPT_PRO_DCACHE_PRELOAD_INT_MAP This register is used to map DCACHE_PRELOAD_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.89: INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP_REG (0x0160)



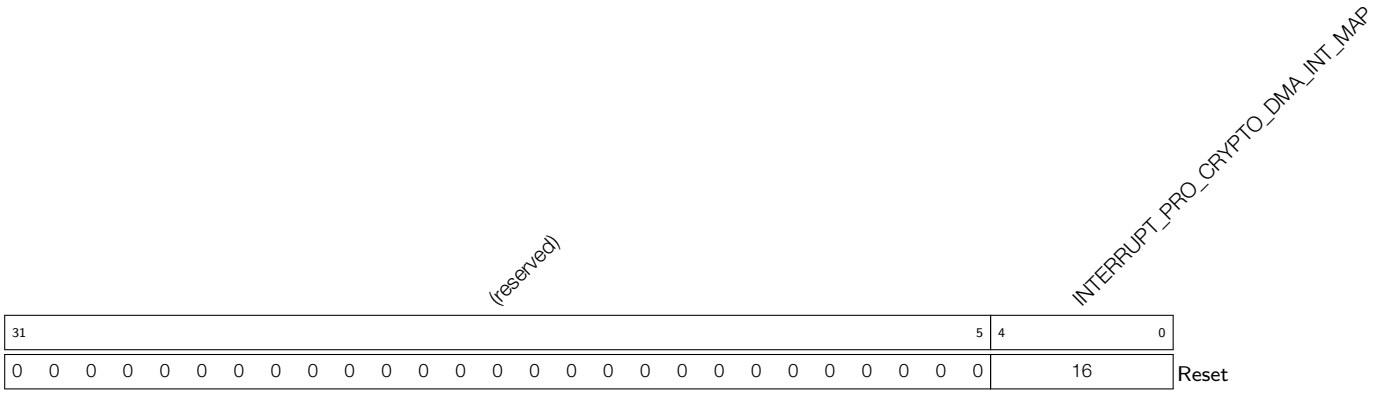
INTERRUPT_PRO_ICACHE_PRELOAD_INT_MAP This register is used to map ICACHE_PRELOAD_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.90: INTERRUPT_PRO_APB_ADC_INT_MAP_REG (0x0164)



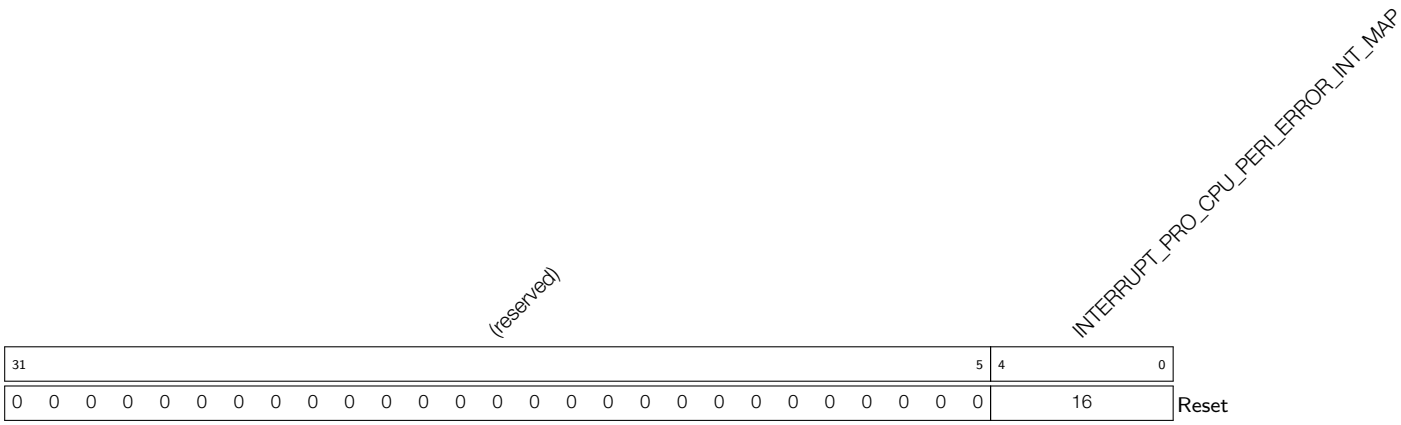
INTERRUPT_PRO_APB_ADC_INT_MAP This register is used to map APB_ADC_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.91: INTERRUPT_PRO_CRYPTO_DMA_INT_MAP_REG (0x0168)



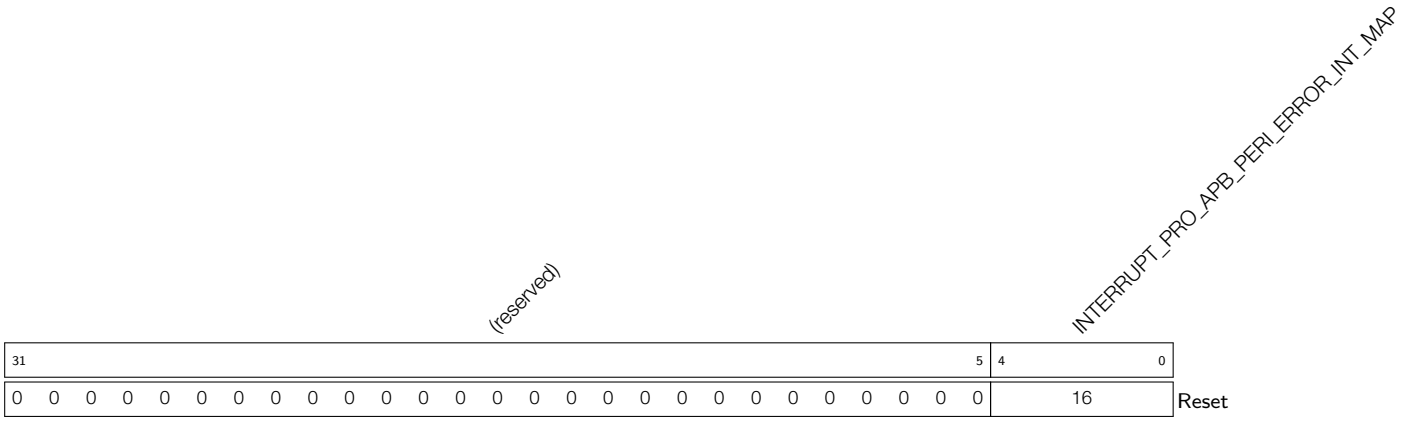
INTERRUPT_PRO_CRYPTO_DMA_INT_MAP This register is used to map CRYPTO_DMA_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.92: INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP_REG (0x016C)



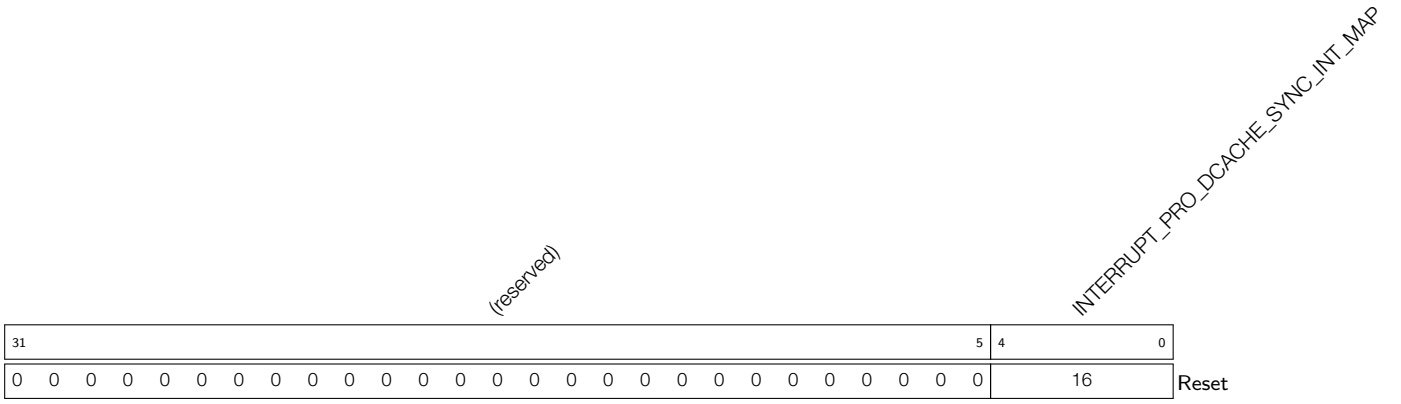
INTERRUPT_PRO_CPU_PERI_ERROR_INT_MAP This register is used to map CPU_PERI_ERROR_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.93: INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP_REG (0x0170)



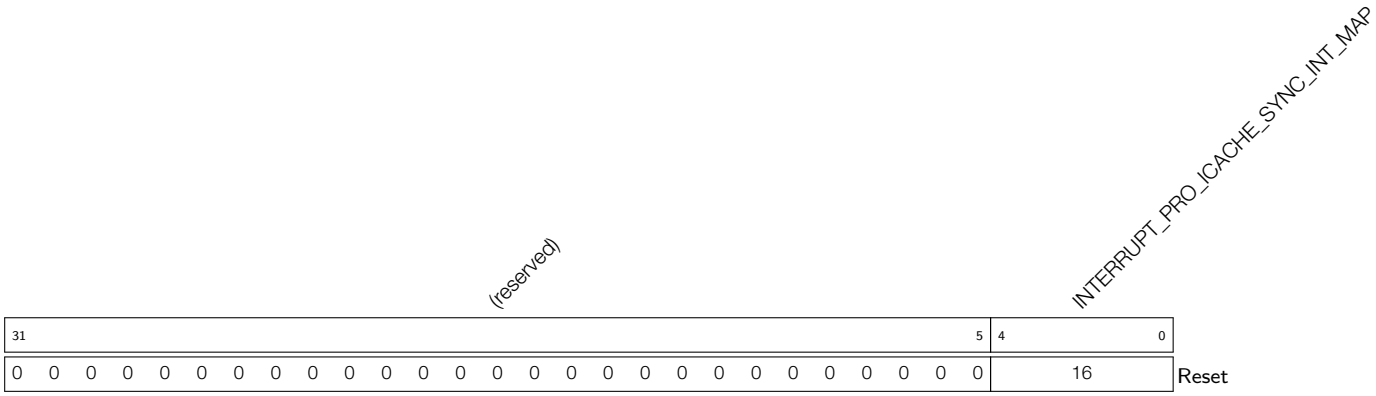
INTERRUPT_PRO_APB_PERI_ERROR_INT_MAP This register is used to map APB_PERI_ERROR_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.94: INTERRUPT_PRO_DCACHE_SYNC_INT_MAP_REG (0x0174)



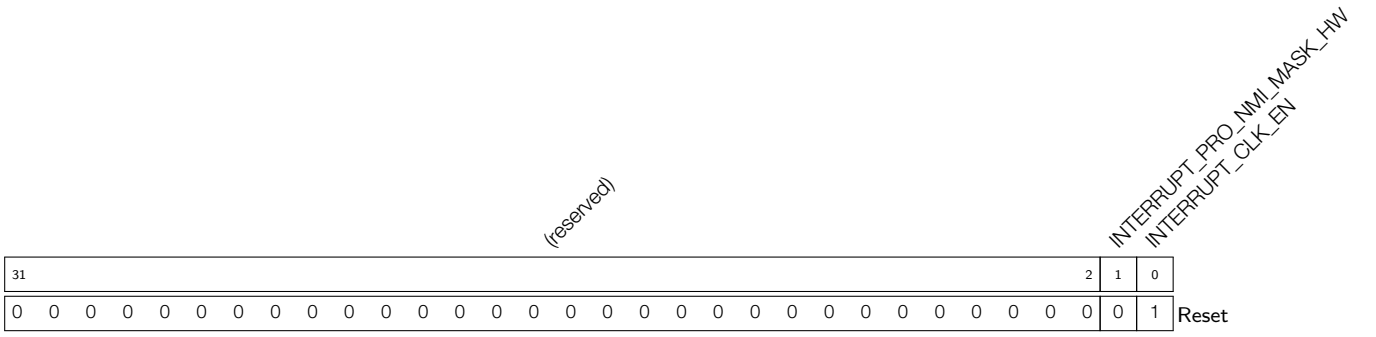
INTERRUPT_PRO_DCACHE_SYNC_INT_MAP This register is used to map DCACHE_SYNC_INT interrupt signal to one of the CPU interrupts. (R/W)

Register 4.95: INTERRUPT_PRO_ICACHE_SYNC_INT_MAP_REG (0x0178)



INTERRUPT_PRO_ICACHE_SYNC_INT_MAP This register is used to map ICACHE_SYNC_INT interrupt signal to one of the CPU interrupts. (R/W)

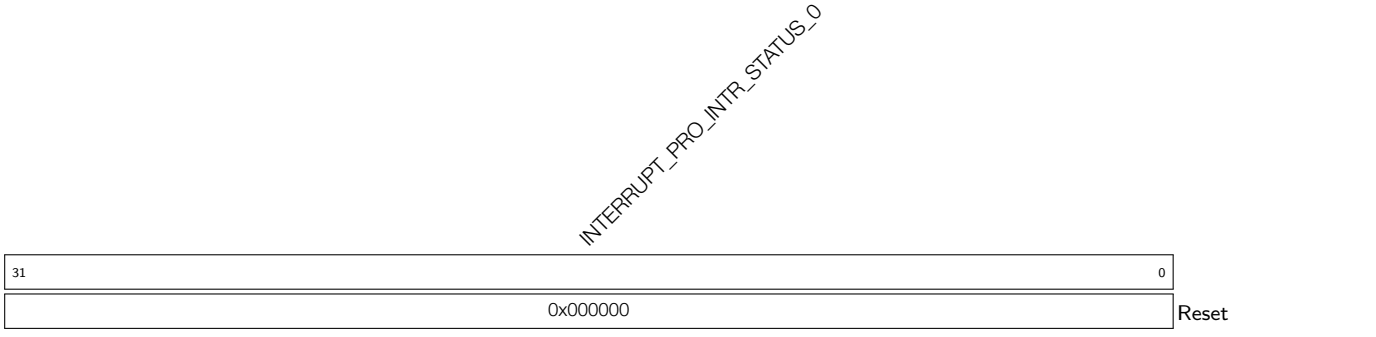
Register 4.96: INTERRUPT_CLOCK_GATE_REG (0x0188)



INTERRUPT_CLK_EN This bit is used to enable or disable the clock of interrupt matrix. 1: enable the clock; 2: disable the clock. (R/W)

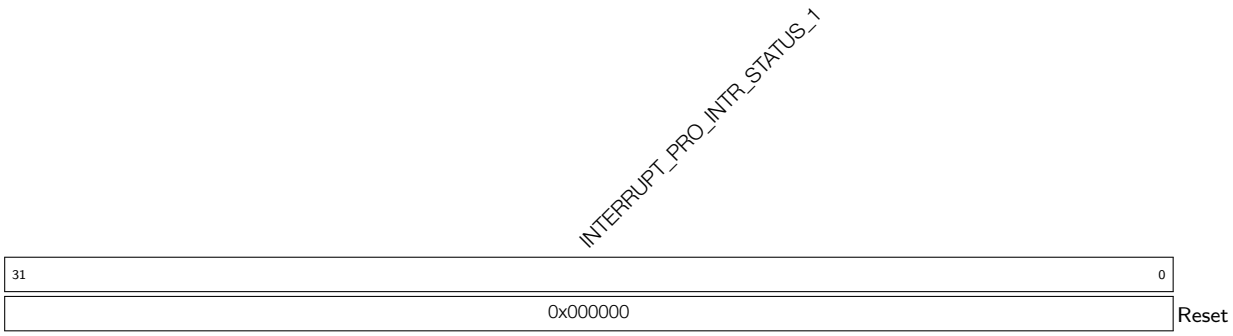
INTERRUPT_PRO_NMI_MASK_HW This bit is used to disable all NMI interrupt signals to CPU. (R/W)

Register 4.97: INTERRUPT_PRO_INTR_STATUS_REG_0_REG (0x017C)



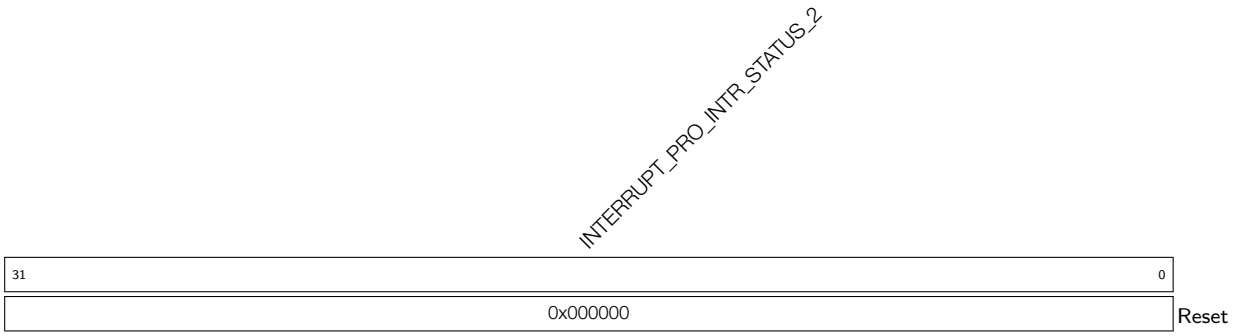
INTERRUPT_PRO_INTR_STATUS_0 This register stores the status of the first 32 input interrupt sources. (RO)

Register 4.98: INTERRUPT_PRO_INTR_STATUS_REG_1_REG (0x0180)



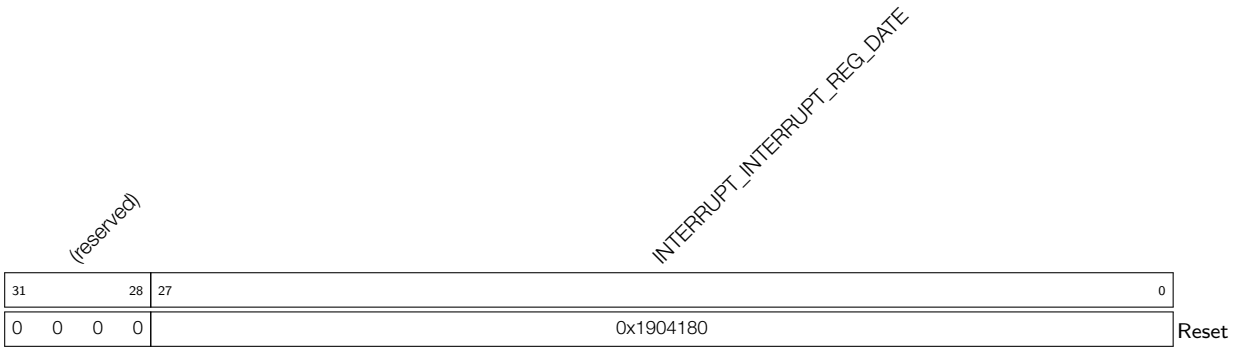
INTERRUPT_PRO_INTR_STATUS_1 This register stores the status of the second 32 input interrupt sources. (RO)

Register 4.99: INTERRUPT_PRO_INTR_STATUS_REG_2_REG (0x0184)



INTERRUPT_PRO_INTR_STATUS_2 This register stores the status of the last 31 input interrupt sources. (RO)

Register 4.100: INTERRUPT_REG_DATE_REG (0x0FFC)



INTERRUPT_DATE Version control register. (R/W)

5. System Registers

5.1 Overview

The ESP32-S2 integrates a large number of peripherals, and enables the control of individual peripherals to achieve optimal characteristics in performance-vs-power-consumption scenarios. Specifically, ESP32-S2 has a various of system configuration registers that can be used for the chip's clock management (clock gating), power management, and the configuration of peripherals and core-system modules. This chapter lists all these system registers and their functions.

5.2 Features

ESP32-S2 system registers can be used to control the following peripheral blocks and core modules:

- System and memory
- Reset and clock
- Interrupt matrix
- eFuse controller
- Low-power management
- Peripheral clock gating and reset

5.3 Function Description

5.3.1 System and Memory Registers

The following registers are used for system and memory configuration, such as cache configuration and memory remapping. For additional information, please refer to Chapter 1 *System and Memory*.

- SYSTEM_ROM_CTRL_0_REG
- SYSTEM_ROM_CTRL_1_REG
- SYSTEM_SRAM_CTRL_0_REG
- SYSTEM_SRAM_CTRL_1_REG
- SYSTEM_SRAM_CTRL_2_REG
- SYSTEM_RSA_PD_CTRL_REG
- SYSTEM_MEM_PD_MASK_REG
- SYSTEM_CACHE_CONTROL_REG
- SYSTEM_BUSTOEXTMEM_ENA_REG
- SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG

ROM Power Consumption Control

Registers SYSTEM_ROM_CTRL_0_REG and SYSTEM_ROM_CTRL_1_REG can be used to control the power consumption of ESP32-S2's ROM. Specifically:

- Setting different bits of the SYSTEM_ROM_FO field in register SYSTEM_ROM_CTRL_0_REG forces on the clock gates of different blocks of ROM.
- Setting different bits of the SYSTEM_ROM_FORCE_PD field in register SYSTEM_ROM_CTRL_1_REG powers down different blocks of internal ROM.
- Setting different bits of the SYSTEM_ROM_FORCE_PU field in register SYSTEM_ROM_CTRL_1_REG powers up different blocks of internal ROM.

For detailed information about the controlling bits of different blocks, please see Table 5-1 below.

Table 5-1. ROM Controlling Bit

ROM	Lowest Address1	Highest Address1	Lowest Address2	Highest Address2	Controlling Bit
Block0	0x4000_0000	0x4000_FFFF	-	-	Bit0
Block1	0x4001_2000	0x4001_FFFF	0x3FFA_0000	0x3FFA_FFFF	Bit1

SRAM Power Consumption Control

Registers SYSTEM_SRAM_CTRL_0_REG, SYSTEM_SRAM_CTRL_1_REG, and SYSTEM_SRAM_CTRL_2_REG can be used to control the power consumption of ESP32-S2's internal SRAM. Specifically,

- Setting different bits of the SYSTEM_SRAM_FO field in register SYSTEM_SRAM_CTRL_0_REG forces on the clock gates of different blocks of internal SRAM.
- Setting different bits of the SYSTEM_SRAM_FORCE_PD field in register SYSTEM_SRAM_CTRL_1_REG powers down different blocks of internal SRAM.
- Setting different bits of the SYSTEM_SRAM_FORCE_PU field in register SYSTEM_SRAM_CTRL_2_REG powers up different blocks of internal SRAM.

For detailed information about the controlling bits of different blocks, please see Table 5-2 below.

Table 5-2. SRAM Controlling Bit

SRAM	Lowest Address1	Highest Address1	Lowest Address2	Highest Address2	Controlling Bit
Block0	0x4002_0000	0x4002_1FFF	0x3FFB_0000	0x3FFB_1FFF	Bit0
Block1	0x4002_2000	0x4002_3FFF	0x3FFB_2000	0x3FFB_3FFF	Bit1
Block2	0x4002_4000	0x4002_5FFF	0x3FFB_4000	0x3FFB_5FFF	Bit2
Block3	0x4002_6000	0x4002_7FFF	0x3FFB_6000	0x3FFB_7FFF	Bit3
Block4	0x4002_8000	0x4002_BFFF	0x3FFB_8000	0x3FFB_BFFF	Bit4
Block5	0x4002_C000	0x4002_FFFF	0x3FFB_C000	0x3FFB_FFFF	Bit5
Block6	0x4003_0000	0x4003_3FFF	0x3FFC_0000	0x3FFC_3FFF	Bit6
Block7	0x4003_4000	0x4003_7FFF	0x3FFC_4000	0x3FFC_7FFF	Bit7
Block8	0x4003_8000	0x4003_BFFF	0x3FFC_8000	0x3FFC_BFFF	Bit8
Block9	0x4003_C000	0x4003_FFFF	0x3FFC_C000	0x3FFC_FFFF	Bit9

Block10	0x4004_0000	0x4004_3FFF	0x3FFD_0000	0x3FFD_3FFF	Bit10
Block11	0x4004_4000	0x4004_7FFF	0x3FFD_4000	0x3FFD_7FFF	Bit11
Block12	0x4004_8000	0x4004_BFFF	0x3FFD_8000	0x3FFD_BFFF	Bit12
Block13	0x4004_C000	0x4004_FFFF	0x3FFD_C000	0x3FFD_FFFF	Bit13
Block14	0x4005_0000	0x4005_3FFF	0x3FFE_0000	0x3FFE_3FFF	Bit14
Block15	0x4005_4000	0x4005_7FFF	0x3FFE_4000	0x3FFE_7FFF	Bit15
Block16	0x4005_8000	0x4005_BFFF	0x3FFE_8000	0x3FFE_BFFF	Bit16
Block17	0x4005_C000	0x4005_FFFF	0x3FFE_C000	0x3FFE_FFFF	Bit17
Block18	0x4006_0000	0x4006_3FFF	0x3FFF_0000	0x3FFF_3FFF	Bit18
Block19	0x4006_4000	0x4006_7FFF	0x3FFF_4000	0x3FFF_7FFF	Bit19
Block20	0x4006_8000	0x4006_BFFF	0x3FFF_8000	0x3FFF_BFFF	Bit20
Block21	0x4006_C000	0x4006_FFFF	0x3FFF_C000	0x3FFF_FFFF	Bit21

5.3.2 Reset and Clock Registers

The following registers are used for reset and clock. For additional information, please refer to Chapter 2 *Reset and Clock*.

- SYSTEM_CPU_PER_CONF_REG
- SYSTEM_SYSCLK_CONF_REG
- SYSTEM_BT_LPCK_DIV_FRAC_REG

5.3.3 Interrupt Matrix Registers

The following registers are used for generating the CPU interrupt signals for the interrupt matrix. For additional information, please refer to Chapter 4 *Interrupt Matrix*

- SYSTEM_CPU_INTR_FROM_CPU_0_REG
- SYSTEM_CPU_INTR_FROM_CPU_1_REG
- SYSTEM_CPU_INTR_FROM_CPU_2_REG
- SYSTEM_CPU_INTR_FROM_CPU_3_REG

5.3.4 JTAG Software Enable Registers

The following registers are used for revoking the temporary disable of eFuse to JTAG.

- SYSTEM_JTAG_CTRL_0_REG
- SYSTEM_JTAG_CTRL_1_REG
- SYSTEM_JTAG_CTRL_2_REG
- SYSTEM_JTAG_CTRL_3_REG
- SYSTEM_JTAG_CTRL_4_REG
- SYSTEM_JTAG_CTRL_5_REG
- SYSTEM_JTAG_CTRL_6_REG

- SYSTEM_JTAG_CTRL_7_REG

5.3.5 Low-power Management Registers

The following registers are used for low-power management.

- SYSTEM_RTC_FASTMEM_CONFIG_REG
- SYSTEM_RTC_FASTMEM_CRC_REG

5.3.6 Peripheral Clock Gating and Reset Registers

The following registers are used for controlling the clock gating and reset of different peripherals. Details can be seen in Table 5-3.

- SYSTEM_CPU_PERI_CLK_EN_REG
- SYSTEM_CPU_PERI_RST_EN_REG
- SYSTEM_PERIP_CLK_EN0_REG
- SYSTEM_PERIP_RST_EN0_REG
- SYSTEM_PERIP_CLK_EN1_REG
- SYSTEM_PERIP_RST_EN1_REG

Table 5-3. Peripheral Clock Gating and Reset Bits

Peripheral	Clock Enabling Bit[1]	Reset Controlling Bit[2][3]
CPU Peripherals	**SYSTEM_CPU_PERI_CLK_EN_REG**	**SYSTEM_CPU_PERI_RST_EN_REG**
DEDICATED GPIO	SYSTEM_CLK_EN_DEDICATED_GPIO	SYSTEM_RST_EN_DEDICATED_GPIO
Peripherals	**SYSTEM_PERIP_CLK_EN0_REG**	**SYSTEM_PERIP_RST_EN0_REG**
Timers	SYSTEM_TIMERS_CLK_EN	SYSTEM_TIMERS_RST
Timer Group0	SYSTEM_TIMERGROUP_CLK_EN	SYSTEM_TIMERGROUP_RST
Timer Group1	SYSTEM_TIMERGROUP1_CLK_EN	SYSTEM_TIMERGROUP1_RST
System Timer	SYSTEM_SYSTIMER_CLK_EN	SYSTEM_SYSTIMER_RST
UART0	SYSTEM_UART_CLK_EN	SYSTEM_UART_RST
UART1	SYSTEM_UART1_CLK_EN	SYSTEM_UART1_RST
UART MEM	SYSTEM_UART_MEM_CLK_EN [4]	SYSTEM_UART_MEM_RST
SPI0, SPI1	SYSTEM_SPI01_CLK_EN	SYSTEM_SPI01_RST
SPI2	SYSTEM_SPI2_CLK_EN	SYSTEM_SPI2_RST
SPI3	SYSTEM_SPI3_DMA_CLK_EN	SYSTEM_SPI3_RST
SPI4	SYSTEM_SPI4_CLK_EN	SYSTEM_SPI4_RST
SPI2 DMA	SYSTEM_SPI2_DMA_CLK_EN	SYSTEM_SPI2_DMA_RST
SPI3 DMA	SYSTEM_SPI3_DMA_CLK_EN	SYSTEM_SPI3_DMA_RST
I2C0	SYSTEM_I2C_EXT0_CLK_EN	SYSTEM_I2C_EXT0_RST
I2C1	SYSTEM_I2C_EXT1_CLK_EN	SYSTEM_I2C_EXT1_RST
I2S0	SYSTEM_I2S0_CLK_EN	SYSTEM_I2S0_RST
I2S1	SYSTEM_I2S1_CLK_EN	SYSTEM_SPI2_DMA_RST
TWAI Controller	SYSTEM_CAN_CLK_EN	SYSTEM_CAN_RST

UHCI0	SYSTEM_UHCI0_CLK_EN	SYSTEM_UHCI0_RST
UHCI1	SYSTEM_UHCI1_CLK_EN	SYSTEM_UHCI1_RST
USB	SYSTEM_USB_CLK_EN	SYSTEM_USB_RST
RMT	SYSTEM_RMT_CLK_EN	SYSTEM_RMT_RST
PCNT	SYSTEM_PCNT_CLK_EN	SYSTEM_PCNT_RST
PWM0	SYSTEM_PWM0_CLK_EN	SYSTEM_PWM0_RST
PWM1	SYSTEM_PWM1_CLK_EN	SYSTEM_PWM1_RST
PWM2	SYSTEM_PWM2_CLK_EN	SYSTEM_PWM2_RST
PWM3	SYSTEM_PWM3_CLK_EN	SYSTEM_PWM3_RST
LED_PWM Controller	SYSTEM_LEDC_CLK_EN	SYSTEM_LEDC_RST
eFuse	SYSTEM_EFUSE_CLK_EN	SYSTEM_EFUSE_RST
APB SARADC	SYSTEM_APB_SARADC_CLK_EN	SYSTEM_APB_SARADC_RST
ADC2 ARB	SYSTEM_ADC2_ARB_CLK_EN	SYSTEM_ADC2_ARB_RST
WDG	SYSTEM_WDG_CLK_EN	SYSTEM_WDG_RST
Accelerators	**SYSTEM_PERIP_CLK_EN1_REG**	**SYSTEM_PERIP_RST_EN1_REG**
DMA	SYSTEM_CRYPTO_DMA_CLK_EN	SYSTEM_CRYPTO_DMA_RST [5]
HMAC	SYSTEM_CRYPTO_HMAC_CLK_EN	SYSTEM_CRYPTO_HMAC_RST [6]
Digital Signature	SYSTEM_CRYPTO_DS_CLK_EN	SYSTEM_CRYPTO_DS_RST [7]
RSA Accelerator	SYSTEM_CRYPTO_RSA_CLK_EN	SYSTEM_CRYPTO_RSA_RST
SHA Accelerator	SYSTEM_CRYPTO_SHA_CLK_EN	SYSTEM_CRYPTO_SHA_RST
AES Accelerator	SYSTEM_CRYPTO_AES_CLK_EN	SYSTEM_CRYPTO_AES_RST

Note:

1. Set the clock enable register to 1 to enable the clock, and to 0 to disable the clock;
2. Set the reset enabling register to 1 to reset a peripheral, and to 0 to disable the reset.
3. Reset registers are not cleared by hardware.
4. UART memory is shared by all UART peripherals, meaning having any active UART peripherals will prevent the UART memory from entering the clock-gated state.
5. Crypto DMA is shared by AES and SHA accelerators.
6. Resetting this bit also resets the SHA accelerator.
7. Resetting this bit also resets the AES, SHA, and RSA accelerators.

5.4 Base Address

Users can access the system registers with base address, which can be seen in the following table. For more information about accessing system registers, please see Chapter 1 *System and Memory*.

Table 5-4. System Register Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F4C0000

5.5 Register Summary

The addresses in the following table are relative to the system registers base addresses provided in Section 5.4.

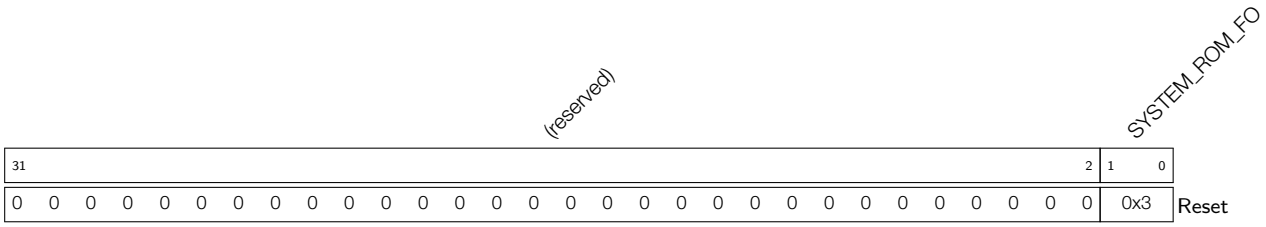
Name	Description	Address	Access
System and Memory Registers			
SYSTEM_ROM_CTRL_0_REG	System ROM configuration register 0	0x0000	R/W
SYSTEM_ROM_CTRL_1_REG	System ROM configuration register 1	0x0004	R/W
SYSTEM_SRAM_CTRL_0_REG	System SRAM configuration register 0	0x0008	R/W
SYSTEM_SRAM_CTRL_1_REG	System SRAM configuration register 1	0x000C	R/W
SYSTEM_SRAM_CTRL_2_REG	System SRAM configuration register 2	0x0088	R/W
SYSTEM_MEM_PD_MASK_REG	Memory power-related controlling register (under low-sleep)	0x003C	R/W
SYSTEM_RSA_PD_CTRL_REG	RSA memory remapping register	0x0068	R/W
SYSTEM_BUSTOEXTMEM_ENA_REG	EDMA enable register	0x006C	R/W
SYSTEM_CACHE_CONTROL_REG	Cache control register	0x0070	R/W
SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG	External memory encrypt and decrypt controlling register	0x0074	R/W
Reset and Clock Registers			
SYSTEM_CPU_PER_CONF_REG	CPU peripheral clock configuration register	0x0018	R/W
SYSTEM_BT_LPCK_DIV_FRAC_REG	Divider fraction configuration register for low-power clock	0x0054	R/W
SYSTEM_SYSCLK_CONF_REG	SoC clock configuration register	0x008C	Varies
Interrupt Matrix Registers			
SYSTEM_CPU_INTR_FROM_CPU_0_REG	CPU interrupt controlling register 0	0x0058	R/W
SYSTEM_CPU_INTR_FROM_CPU_1_REG	CPU interrupt controlling register 1	0x005C	R/W
SYSTEM_CPU_INTR_FROM_CPU_2_REG	CPU interrupt controlling register 2	0x0060	R/W
SYSTEM_CPU_INTR_FROM_CPU_3_REG	CPU interrupt controlling register 3	0x0064	R/W
JTAG Software Enable Registers			
SYSTEM_JTAG_CTRL_0_REG	JTAG configuration register 0	0x001C	WO
SYSTEM_JTAG_CTRL_1_REG	JTAG configuration register 1	0x0020	WO
SYSTEM_JTAG_CTRL_2_REG	JTAG configuration register 2	0x0024	WO
SYSTEM_JTAG_CTRL_3_REG	JTAG configuration register 3	0x0028	WO
SYSTEM_JTAG_CTRL_4_REG	JTAG configuration register 4	0x002C	WO
SYSTEM_JTAG_CTRL_5_REG	JTAG configuration register 5	0x0030	WO

Name	Description	Address	Access
SYSTEM_JTAG_CTRL_6_REG	JTAG configuration register 6	0x0034	WO
SYSTEM_JTAG_CTRL_7_REG	JTAG configuration register 7	0x0038	WO
Low-Power Management Registers			
SYSTEM_RTC_FASTMEM_CONFIG_REG	RTC fast memory configuration register	0x0078	Varies
SYSTEM_RTC_FASTMEM_CRC_REG	RTC fast memory CRC controlling register	0x007C	RO
Peripheral Clock Gating and Reset Registers			
SYSTEM_CPU_PERI_CLK_EN_REG	CPU peripheral clock enable register	0x0010	R/W
SYSTEM_CPU_PERI_RST_EN_REG	CPU peripheral reset register	0x0014	R/W
SYSTEM_PERIP_CLK_EN0_REG	System peripheral clock (for hardware accelerators) enable register 0	0x0040	R/W
SYSTEM_PERIP_CLK_EN1_REG	System peripheral clock (for hardware accelerators) enable register 1	0x0044	R/W
SYSTEM_PERIP_RST_EN0_REG	System peripheral (hardware accelerators) reset register 0	0x0048	R/W
SYSTEM_PERIP_RST_EN1_REG	System peripheral (hardware accelerators) reset register 1	0x004C	R/W
Version Register			
SYSTEM_REG_DATE_REG	Version control register	0x0FFC	R/W

5.6 Registers

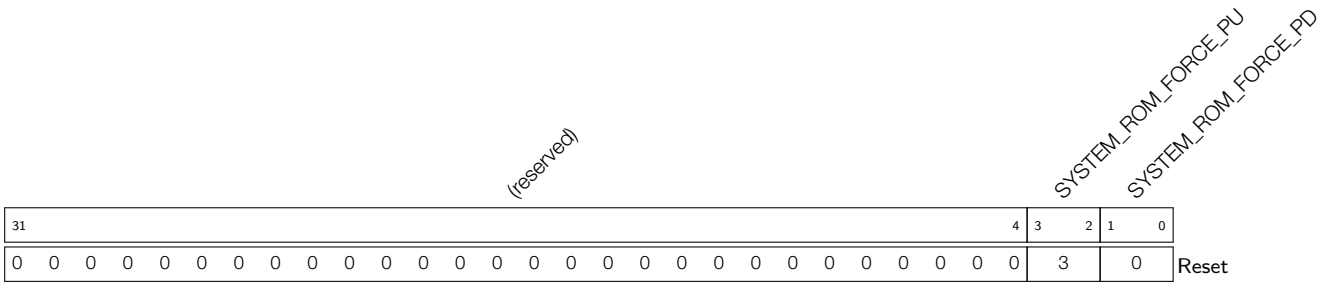
The addresses below are relative to the system registers base addresses provided in Section 5.4.

Register 5.1: SYSTEM_ROM_CTRL_0_REG (0x0000)



SYSTEM_ROM_FO This field is used to force on clock gate of internal ROM. For details, please refer to Table 5-1. (R/W)

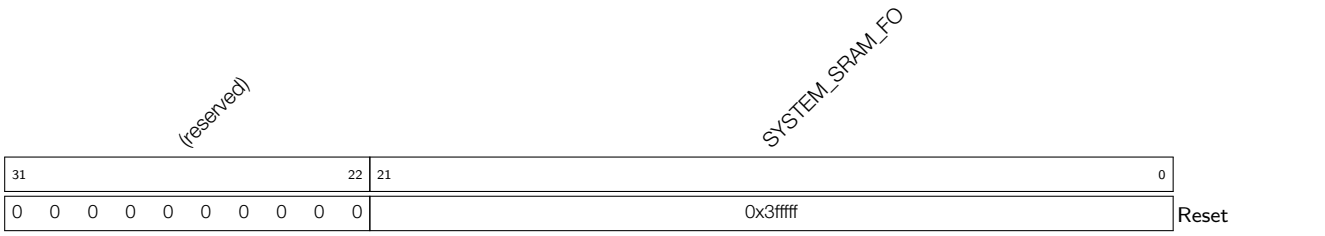
Register 5.2: SYSTEM_ROM_CTRL_1_REG (0x0004)



SYSTEM_ROM_FORCE_PD This field is used to power down internal ROM. For details, please refer to Table 5-1. (R/W)

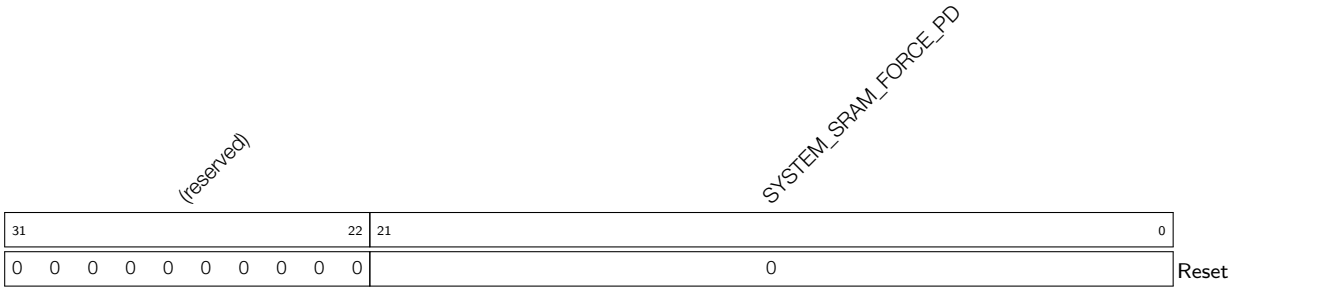
SYSTEM_ROM_FORCE_PU This field is used to power up internal ROM. For details, please refer to Table 5-1. (R/W)

Register 5.3: SYSTEM_SRAM_CTRL_0_REG (0x0008)



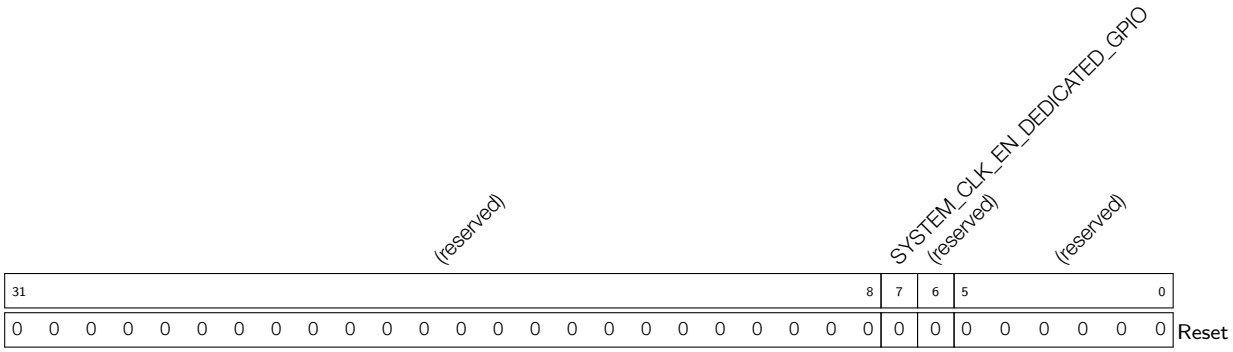
SYSTEM_SRAM_FO This field is used to force on clock gate of internal SRAM. For details, please refer to Table 5-2. (R/W)

Register 5.4: SYSTEM_SRAM_CTRL_1_REG (0x000C)



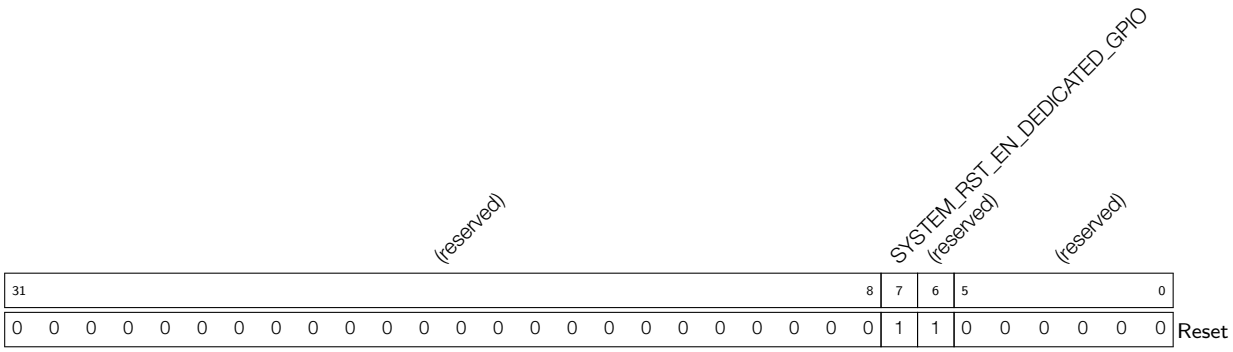
SYSTEM_SRAM_FORCE_PD This field is used to power down internal SRAM. For details, please refer to Table 5-2. (R/W)

Register 5.5: SYSTEM_CPU_PERI_CLK_EN_REG (0x0010)



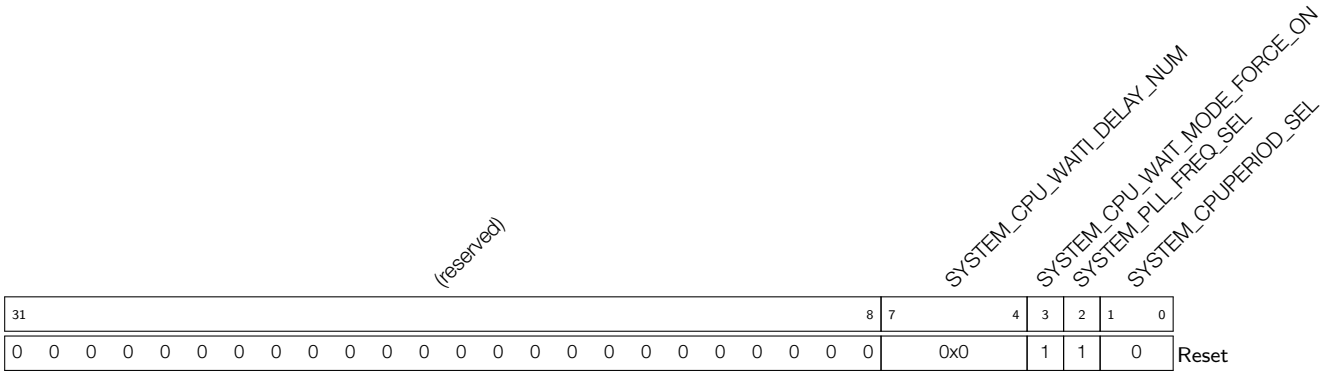
SYSTEM_CLK_EN_DEDICATED_GPIO Set this bit to enable clock of DEDICATED GPIO module.

Register 5.6: SYSTEM_CPU_PERI_RST_EN_REG (0x0014)



SYSTEM_RST_EN_DEDICATED_GPIO Set this bit to reset DEDICATED GPIO module. (R/W)

Register 5.7: SYSTEM_CPU_PER_CONF_REG (0x0018)



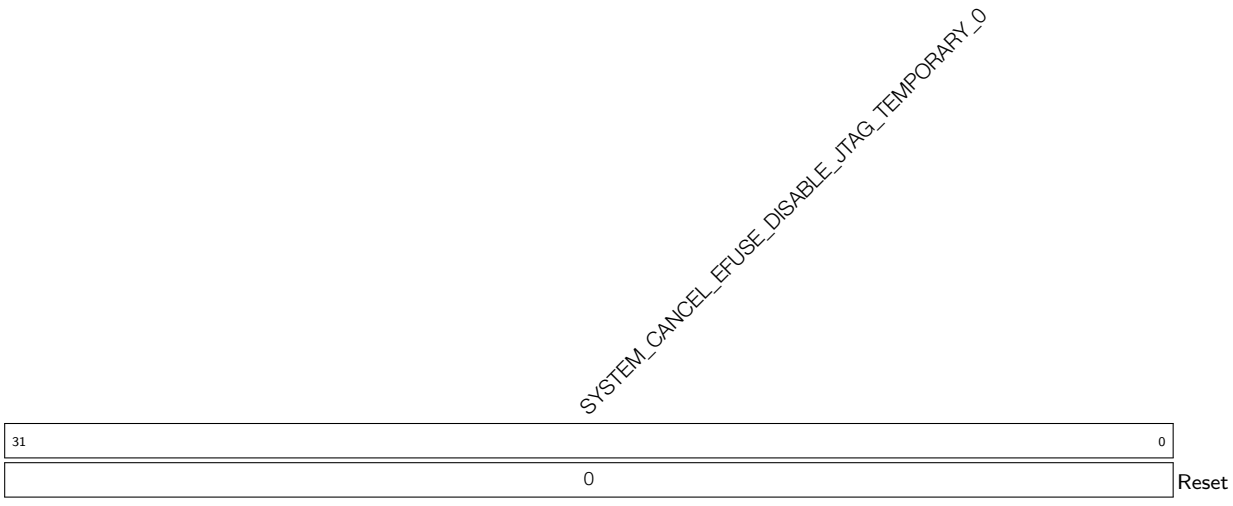
SYSTEM_CPUPERIOD_SEL This field is used to select the clock frequency of CPU or CPU period. For details, please refer to Table 2-2 in Chapter 2 *Reset and Clock*. (R/W)

SYSTEM_PLL_FREQ_SEL This field is used to select the PLL clock frequency based on CPU period. For details, please refer to Table 2-2 in Chapter 2 *Reset and Clock*. (R/W)

SYSTEM_CPU_WAIT_MODE_FORCE_ON Set this bit to force on CPU wait mode. In this mode, the clock gate of CPU is turned off until any interrupts happen. This mode could also be force on via WAITI instruction. (R/W)

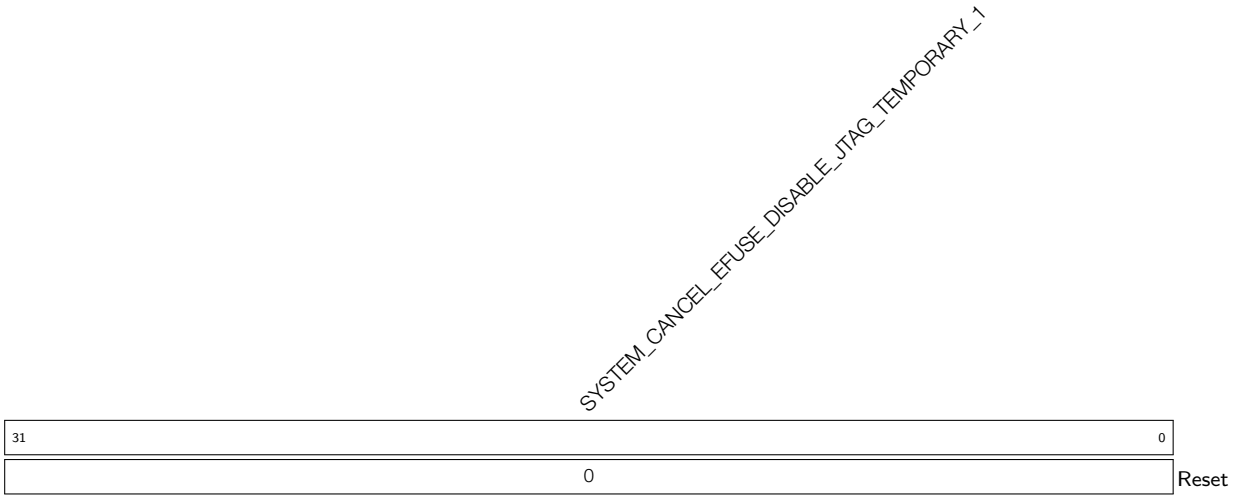
SYSTEM_CPU_WAITI_DELAY_NUM Sets the number of delay cycles to enter CPU wait mode after a WAITI instruction. (R/W)

Register 5.8: SYSTEM_JTAG_CTRL_0_REG (0x001C)



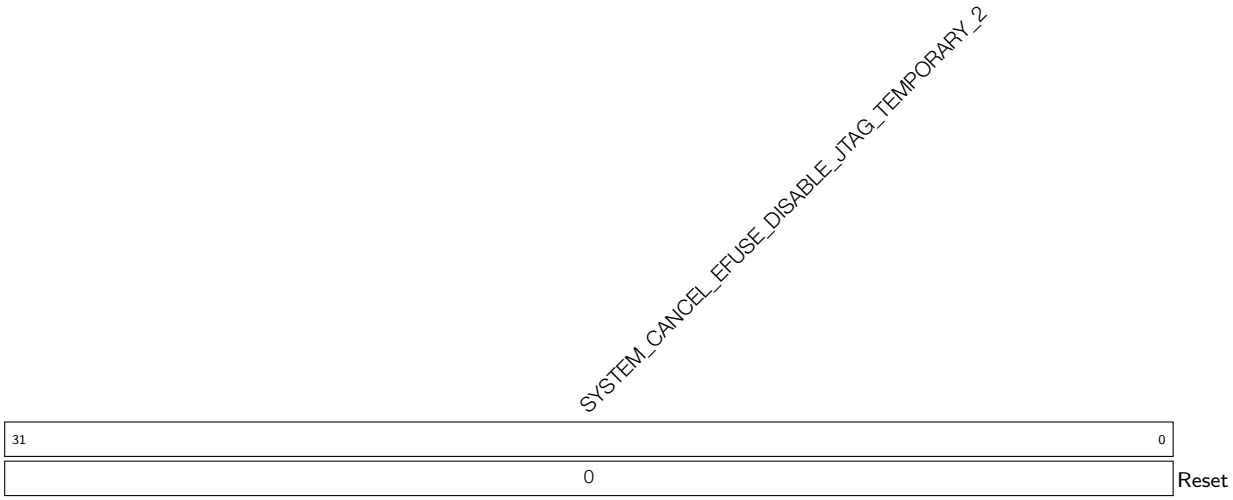
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_0 Stores the 0 to 31 bits of the 256 bits register used to cancel the temporary disable of eFuse to JTAG.

Register 5.9: SYSTEM_JTAG_CTRL_1_REG (0x0020)



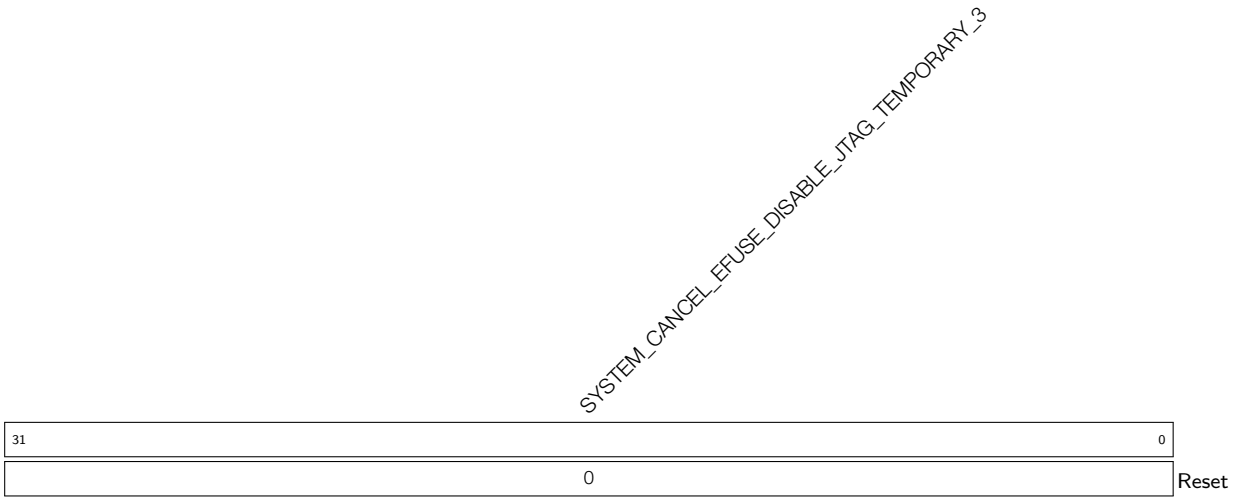
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_1 Stores the 32 to 63 bits of the 256 bits register used to cancel the temporary disable of eFuse to JTAG.

Register 5.10: SYSTEM_JTAG_CTRL_2_REG (0x0024)



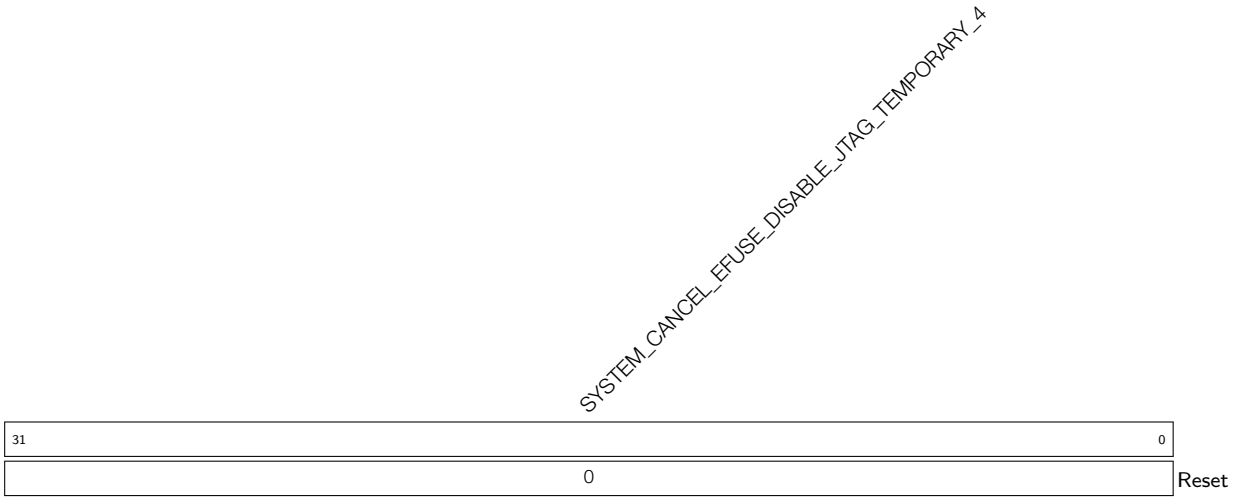
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_2 Stores the 64 to 95 bits of the 256 bits register used to cancel the temporary disable of eFuse to JTAG.

Register 5.11: SYSTEM_JTAG_CTRL_3_REG (0x0028)



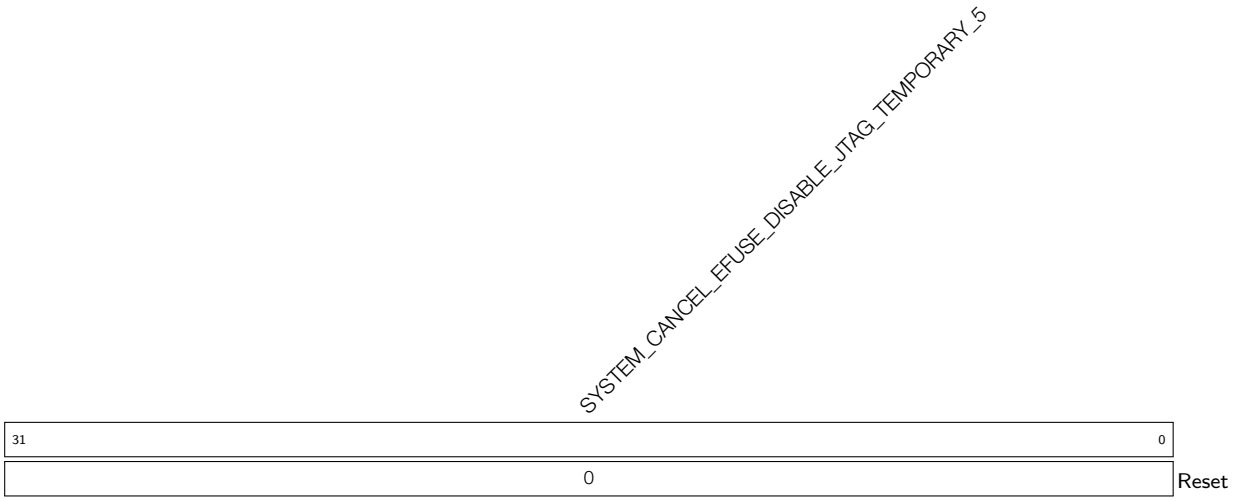
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_3 Stores the 96 to 127 bits of the 256 bits register used to cancel the temporary disable of eFuse to JTAG.

Register 5.12: SYSTEM_JTAG_CTRL_4_REG (0x002C)



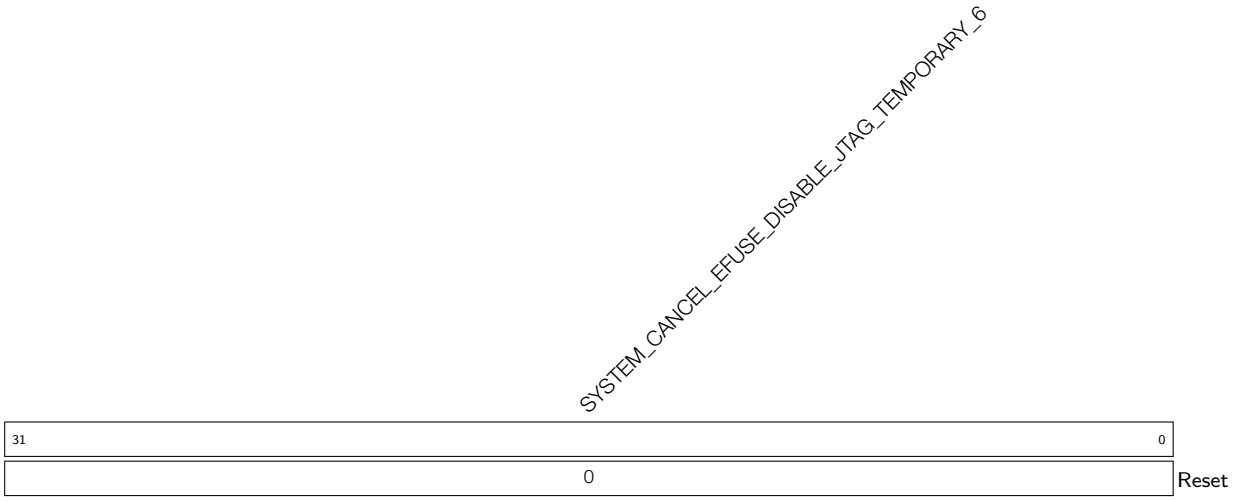
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_4 Stores the 128 to 159 bits of the 256 bits register used to cancel the temporary disable of eFuse to JTAG.

Register 5.13: SYSTEM_JTAG_CTRL_5_REG (0x0030)



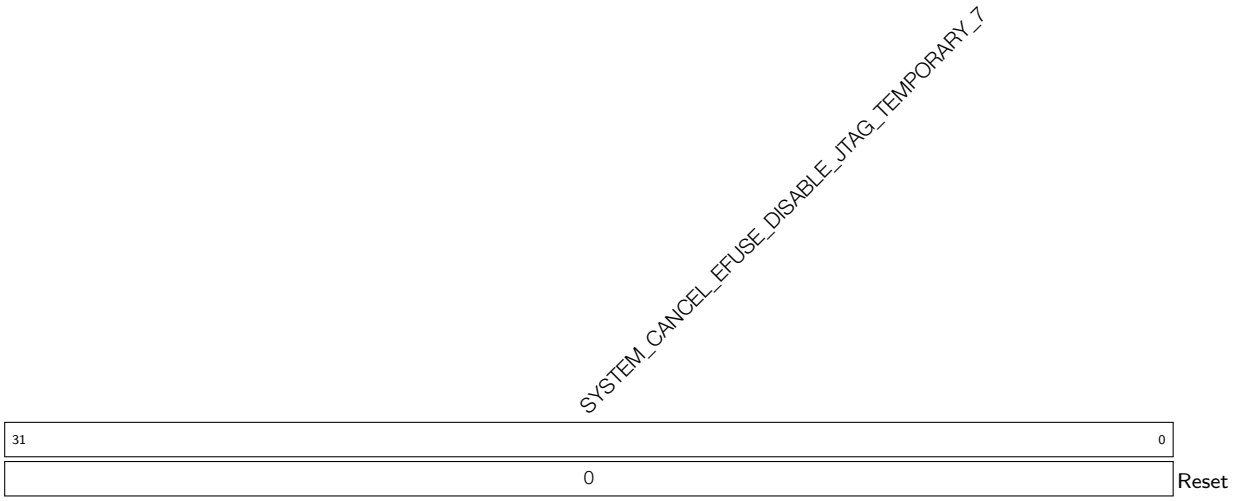
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_5 Stores the 160 to 191 bits of the 256 bits register used to cancel the temporary disable of eFuse to JTAG.

Register 5.14: SYSTEM_JTAG_CTRL_6_REG (0x0034)



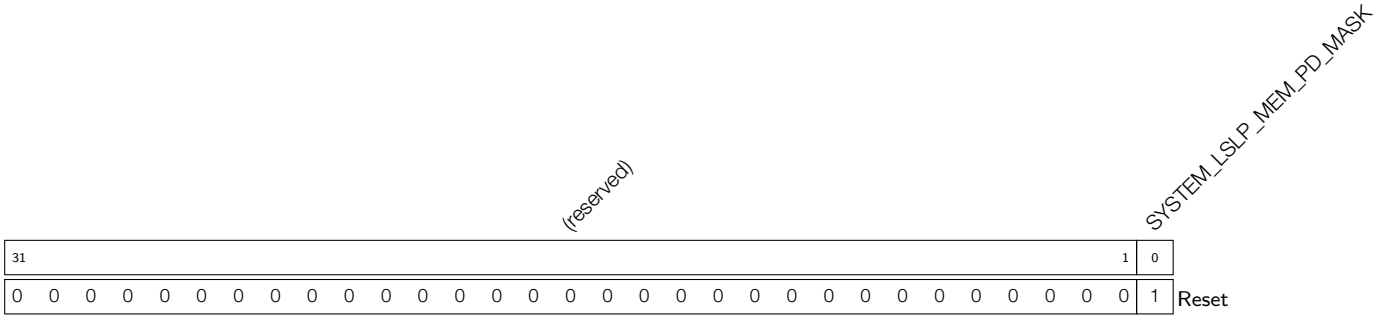
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_6 Stores the 192 to 223 bits of the 256 bits register used to cancel the temporary disable of eFuse to JTAG.

Register 5.15: SYSTEM_JTAG_CTRL_7_REG (0x0038)



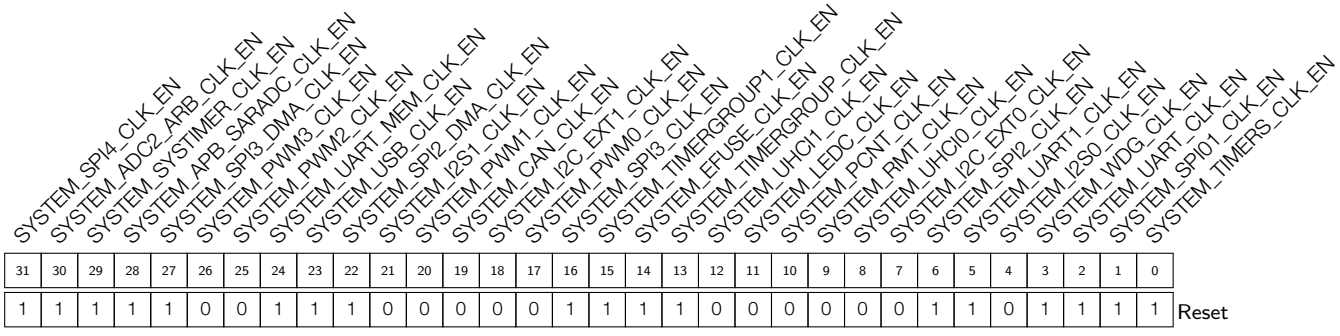
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_7 Stores the 224 to 255 bits of the 256 bits register used to cancel the temporary disable of eFuse to JTAG.

Register 5.16: SYSTEM_MEM_PD_MASK_REG (0x003C)



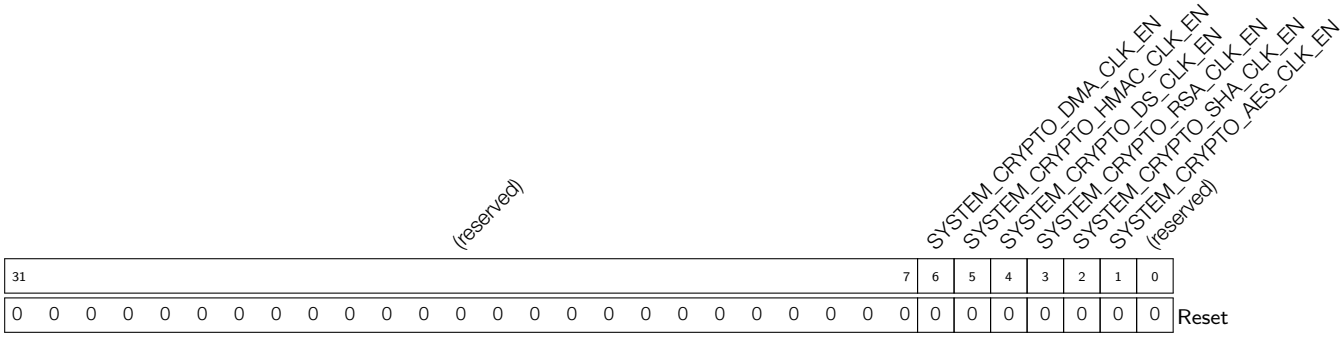
SYSTEM_LSLP_MEM_PD_MASK Set this bit to allow the memory to work as usual when the chip enters the light-sleep state. (R/W)

Register 5.17: SYSTEM_PERIP_CLK_EN0_REG (0x0040)



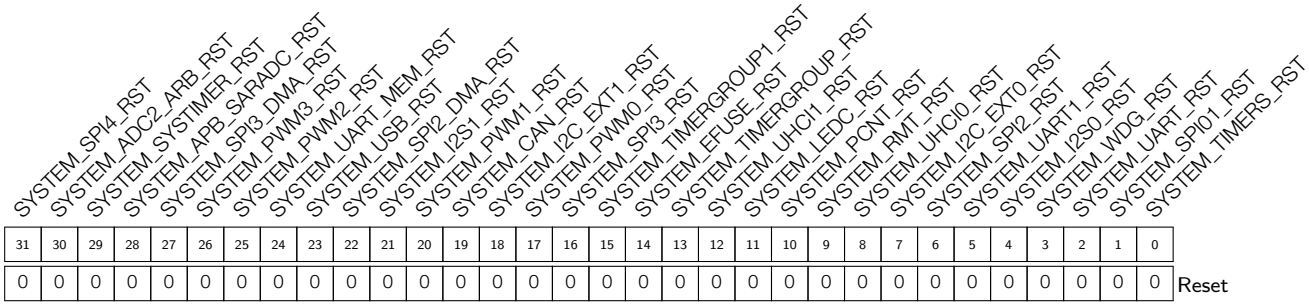
SYSTEM_CPU_PERI_CLK_EN0_REG Configures this register to enable different peripheral clocks. For details, please refer to Table 5-3.

Register 5.18: SYSTEM_PERIP_CLK_EN1_REG (0x0044)



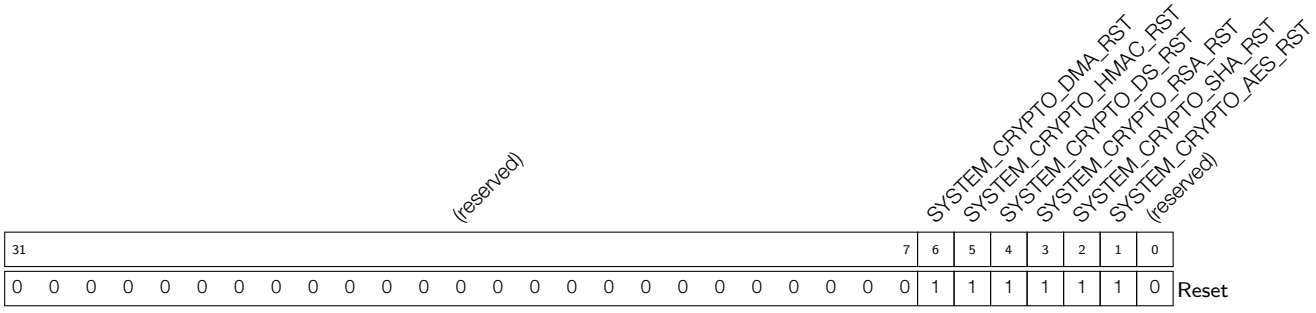
SYSTEM_PERIP_CLK_EN1_REG Configures this register to enable different accelerator clocks. For details, please refer to Table 5-3.

Register 5.19: SYSTEM_PERIP_RST_EN0_REG (0x0048)



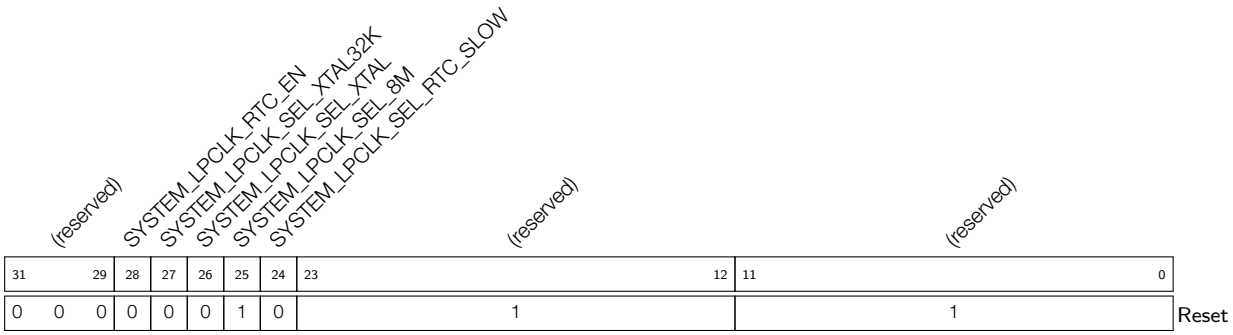
SYSTEM_PERIP_RST_EN0_REG Configures this register to reset different peripherals. For details, please refer to Table 5-3.

Register 5.20: SYSTEM_PERIP_RST_EN1_REG (0x004C)



SYSTEM_PERIP_RST_EN1_REG Configures this register to reset different accelerators. For details, please refer to Table 5-3.

Register 5.21: SYSTEM_BT_LPCK_DIV_FRAC_REG (0x0054)



SYSTEM_LPCLK_SEL_RTC_SLOW Set this bit to select RTC slow clock as the low power clock. (R/W)

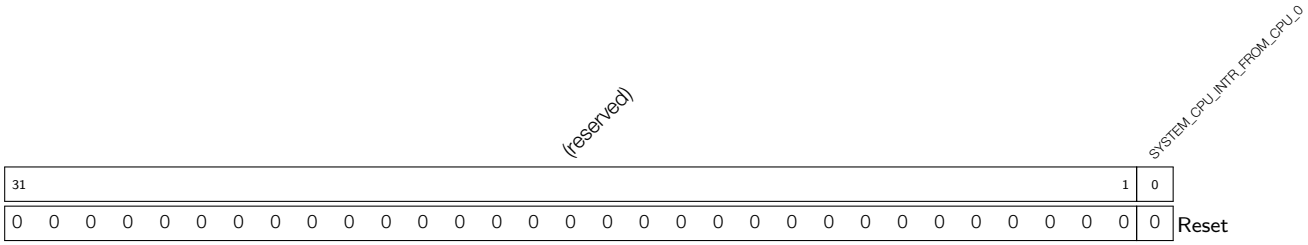
SYSTEM_LPCLK_SEL_8M Set this bit to select 8m clock as the low power clock. (R/W)

SYSTEM_LPCLK_SEL_XTAL Set this bit to select xtal clock as the low power clock. (R/W)

SYSTEM_LPCLK_SEL_XTAL32K Set this bit to select xtal32k clock as the low power clock. (R/W)

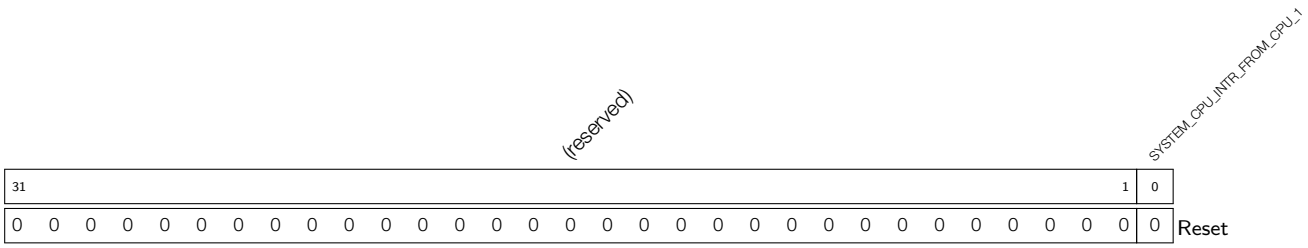
SYSTEM_LPCLK_RTC_EN Set this bit to enable the RTC low power clock. (R/W)

Register 5.22: SYSTEM_CPU_INTR_FROM_CPU_0_REG (0x0058)



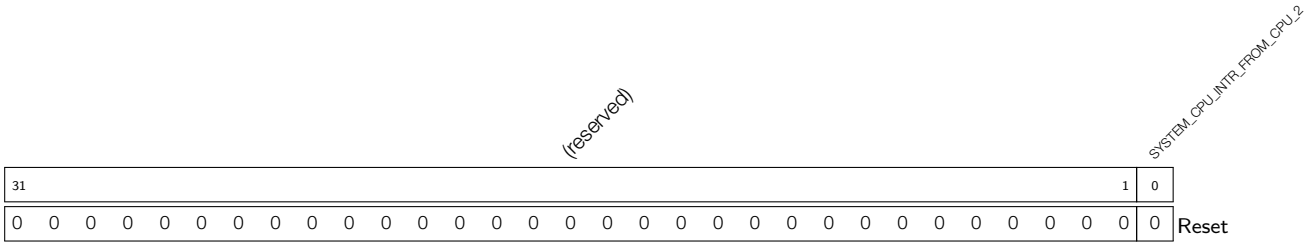
SYSTEM_CPU_INTR_FROM_CPU_0 Set this bit to generate CPU interrupt 0. This bit needs to be reset by software in the ISR process. (R/W)

Register 5.23: SYSTEM_CPU_INTR_FROM_CPU_1_REG (0x005C)



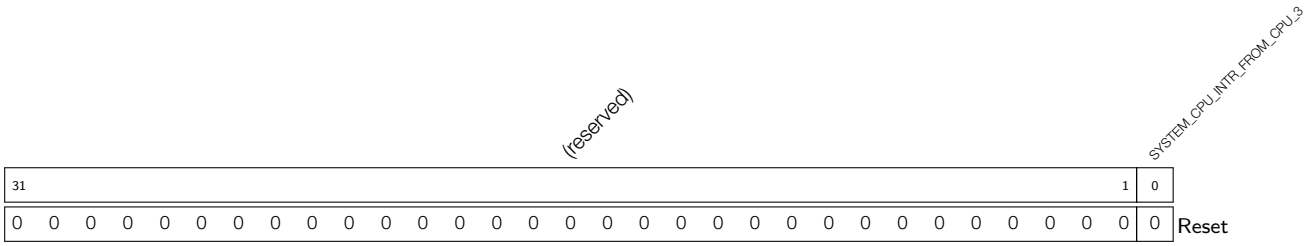
SYSTEM_CPU_INTR_FROM_CPU_1 Set this bit to generate CPU interrupt 1. This bit needs to be reset by software in the ISR process. (R/W)

Register 5.24: SYSTEM_CPU_INTR_FROM_CPU_2_REG (0x0060)



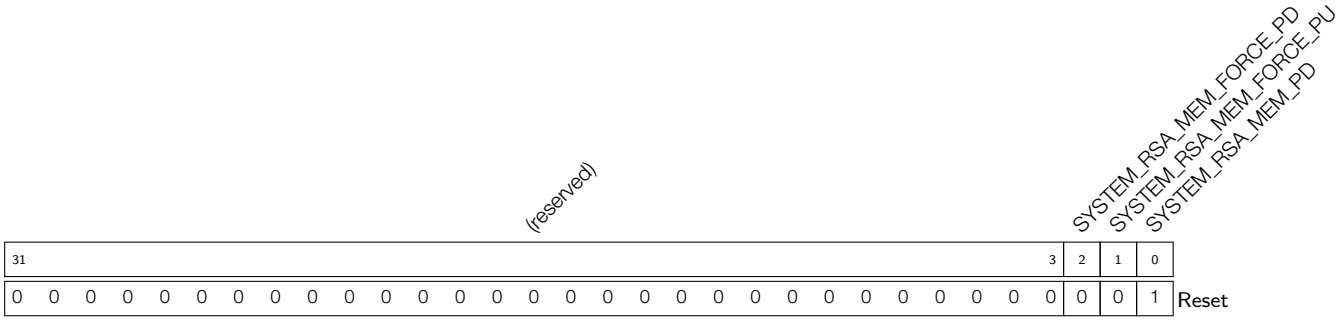
SYSTEM_CPU_INTR_FROM_CPU_2 Set this bit to generate CPU interrupt 2. This bit needs to be reset by software in the ISR process. (R/W)

Register 5.25: SYSTEM_CPU_INTR_FROM_CPU_3_REG (0x0064)



SYSTEM_CPU_INTR_FROM_CPU_3 Set this bit to generate CPU interrupt 3. This bit needs to be reset by software in the ISR process. (R/W)

Register 5.26: SYSTEM_RSA_PD_CTRL_REG (0x0068)

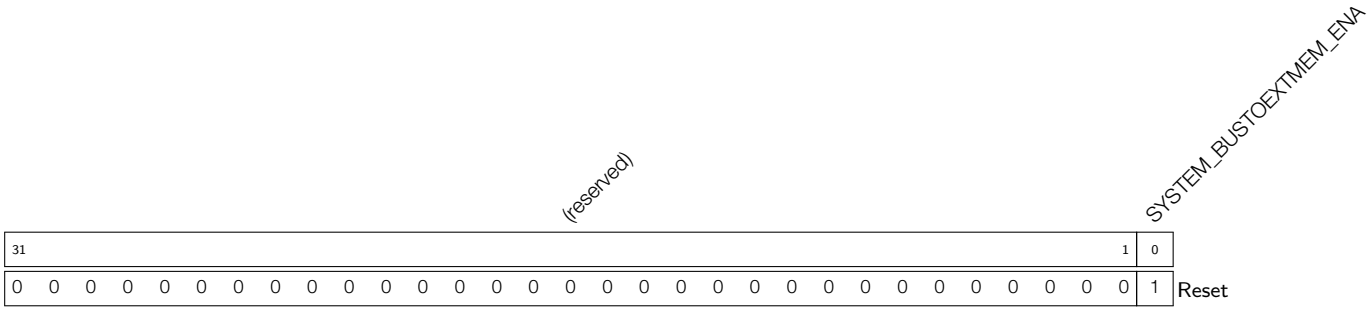


SYSTEM_RSA_MEM_PD Set this bit to power down RSA memory. This bit has the lowest priority. When Digital Signature occupies the RSA, this bit is invalid. (R/W)

SYSTEM_RSA_MEM_FORCE_PU Set this bit to force power up RSA memory. This bit has the second highest priority. (R/W)

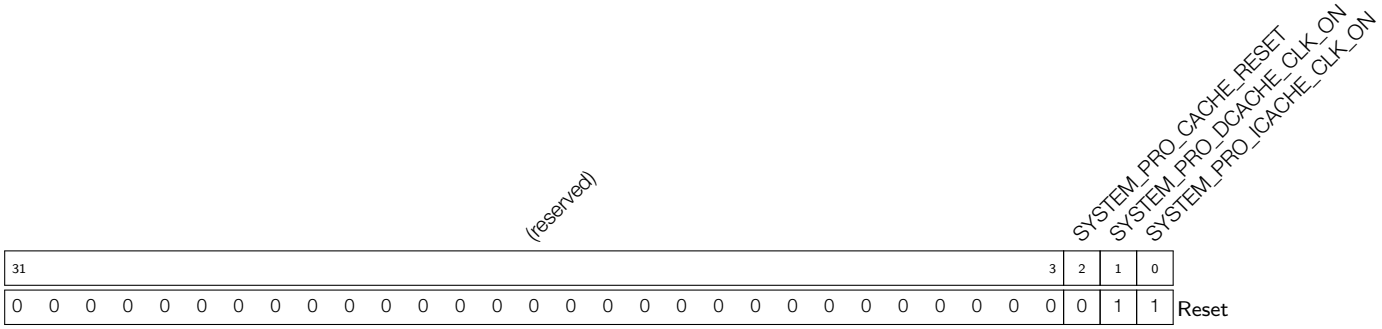
SYSTEM_RSA_MEM_FORCE_PD Set this bit to force power down RSA memory. This bit has the highest priority. (R/W)

Register 5.27: SYSTEM_BUSTOEXTMEM_ENA_REG (0x006C)



SYSTEM_BUSTOEXTMEM_ENA Set this bit to enable bus to EDMA. (R/W)

Register 5.28: SYSTEM_CACHE_CONTROL_REG (0x0070)

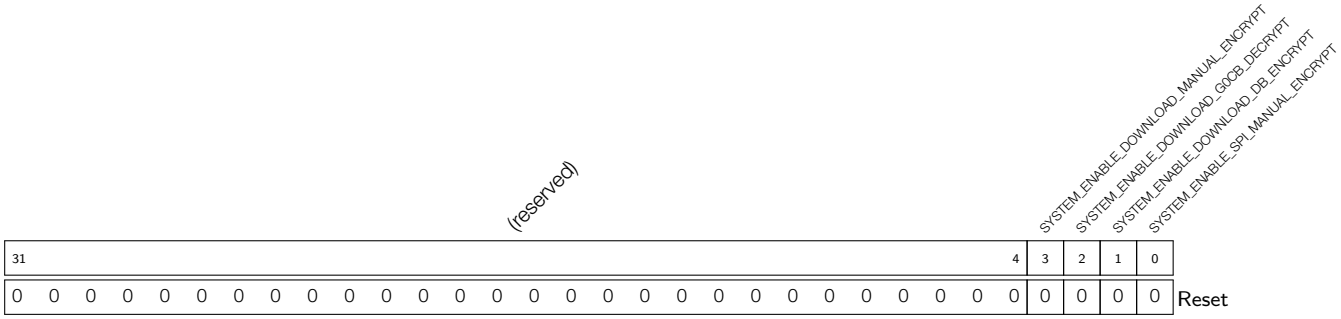


SYSTEM_PRO_ICACHE_CLK_ON Set this bit to enable clock of i-cache. (R/W)

SYSTEM_PRO_DCACHE_CLK_ON Set this bit to enable clock of d-cache. (R/W)

SYSTEM_PRO_CACHE_RESET Set this bit to reset cache. (R/W)

Register 5.29: SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG (0x0074)



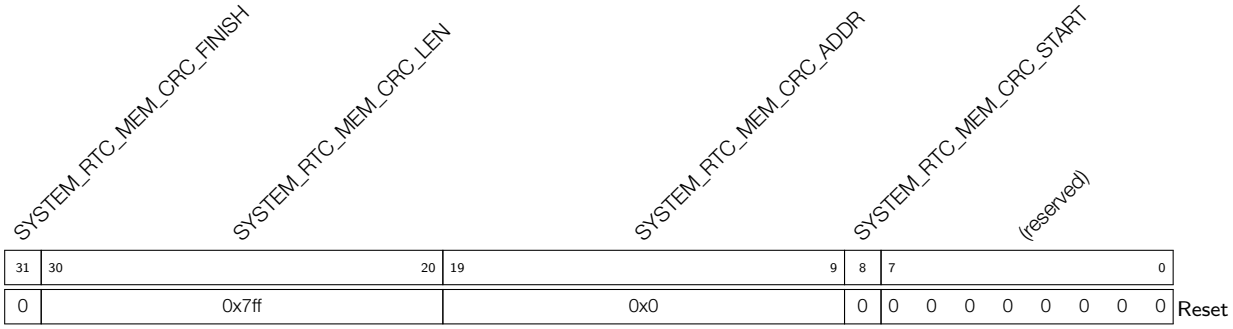
SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT Set this bit to enable Manual Encryption under SPI Boot mode. (R/W)

SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT Set this bit to enable Auto Encryption under Download Boot mode. (R/W)

SYSTEM_ENABLE_DOWNLOAD_G0CB_DECRYPT Set this bit to enable Auto Decryption under Download Boot mode. (R/W)

SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT Set this bit to enable Manual Encryption under Download Boot mode. (R/W)

Register 5.30: SYSTEM_RTC_FASTMEM_CONFIG_REG (0x0078)



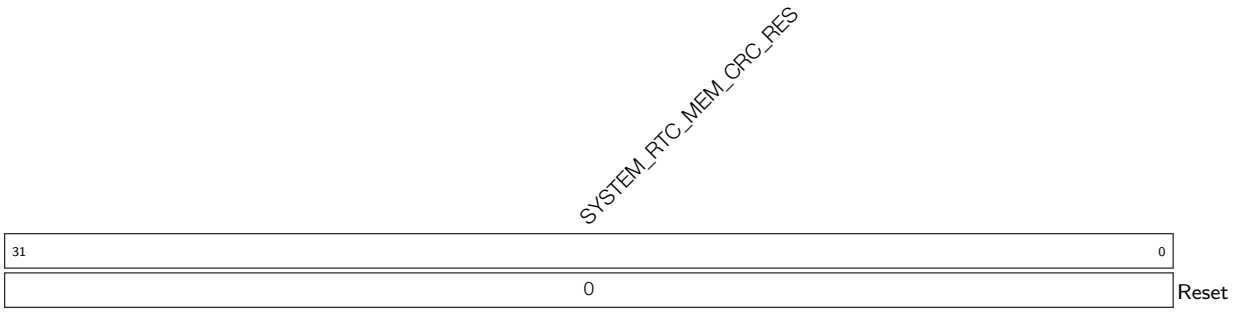
SYSTEM_RTC_MEM_CRC_START Set this bit to start the CRC of RTC memory. (R/W)

SYSTEM_RTC_MEM_CRC_ADDR This field is used to set address of RTC memory for CRC. (R/W)

SYSTEM_RTC_MEM_CRC_LEN This field is used to set length of RTC memory for CRC based on start address. (R/W)

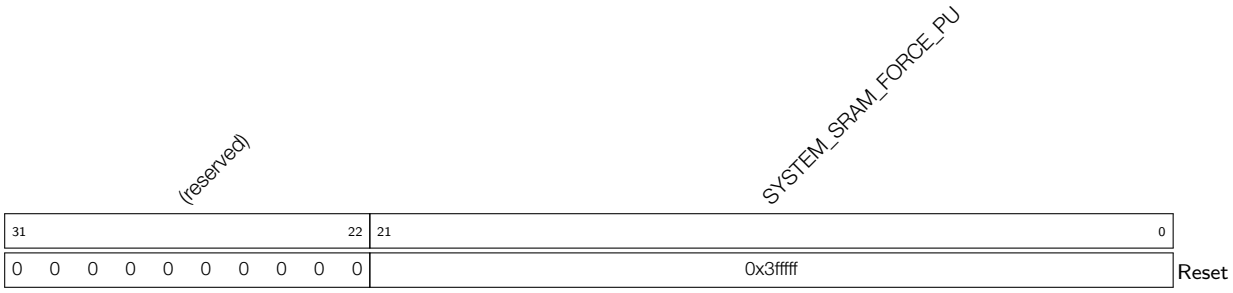
SYSTEM_RTC_MEM_CRC_FINISH This bit stores the status of RTC memory CRC. High level means finished while low level means not finished. (RO)

Register 5.31: SYSTEM_RTC_FASTMEM_CRC_REG (0x007C)



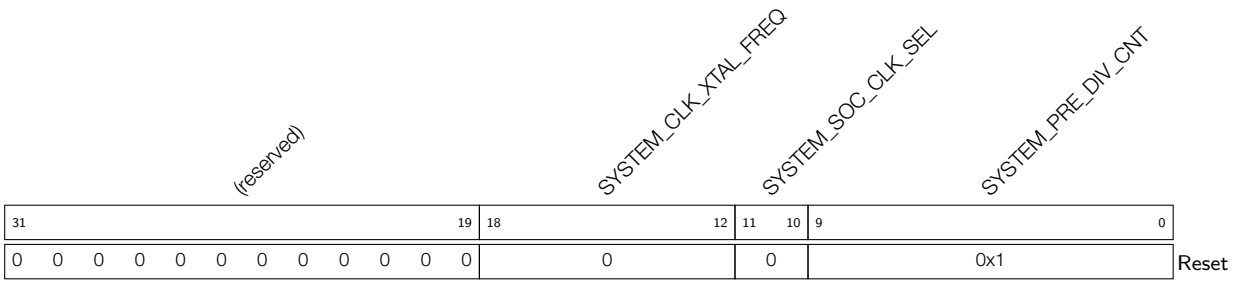
SYSTEM_RTC_MEM_CRC_RES This field stores the CRC result of RTC memory. (RO)

Register 5.32: SYSTEM_SRAM_CTRL_2_REG (0x0088)



SYSTEM_SRAM_FORCE_PU This field is used to power up internal SRAM. For details, please refer to Table 5-2. (R/W)

Register 5.33: SYSTEM_SYSCLK_CONF_REG (0x008C)

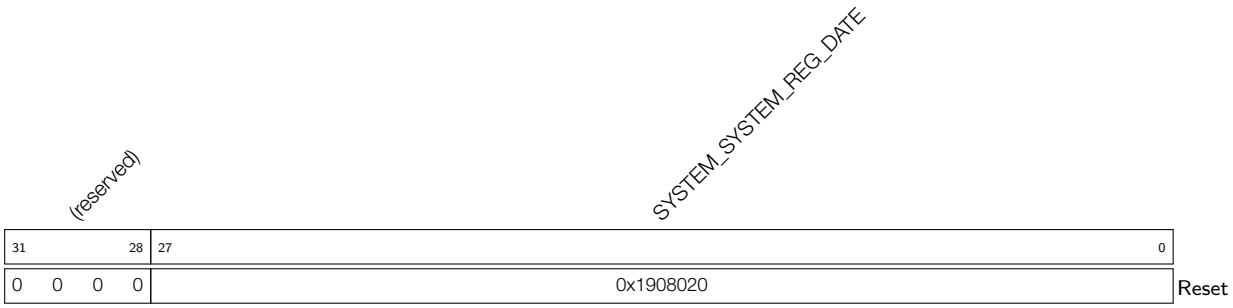


SYSTEM_PRE_DIV_CNT This field is used to set the count of prescaler of XTAL_CLK. For details, please refer to Table 2-4 in Chapter 2 *Reset and Clock*. (R/W)

SYSTEM_SOC_CLK_SEL This field is used to select SOC clock. For details, please refer to Table 2-2 in Chapter 2 *Reset and Clock*. (R/W)

SYSTEM_CLK_XTAL_FREQ This field is used to read XTAL frequency in MHz. (RO)

Register 5.34: SYSTEM_REG_DATE_REG (0x0FFC)



SYSTEM_DATE Version control register. (R/W)

6. LED PWM Controller

6.1 Overview

The LED PWM controller is primarily designed to control LED devices, as well as generate PWM signals. It has 14-bit timers and waveform generators.

6.2 Features

The LED PWM controller has the following features:

- Four independent timers that support division by fractions
- Eight independent waveform generators able to produce eight PWM signals
- Fading duty cycle of PWM signals without interference from any processors. An interrupt can be generated after the fade has completed
- Adjustable phase of PWM signal output
- PWM signal output in low-power mode

For the convenience of description, in the following sections the eight waveform generators are collectively referred to as PWM*n*, and the four timers are collectively referred to as Timer*x*.

6.3 Functional Description

6.3.1 Architecture

Figure 6-1 shows the architecture of the LED PWM controller. Figure 6-2 illustrates a PWM generator with its selected timer and a counter.

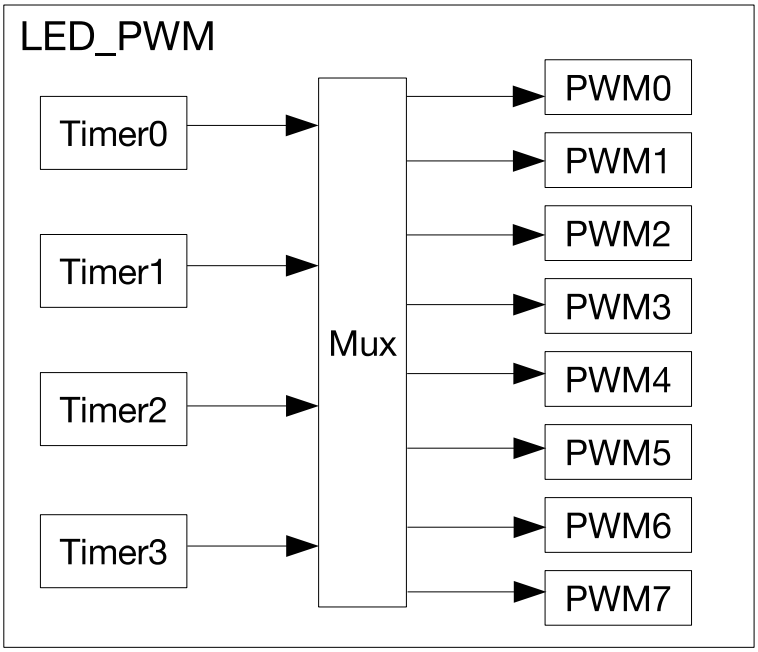


Figure 6-1. LED_PWM Architecture

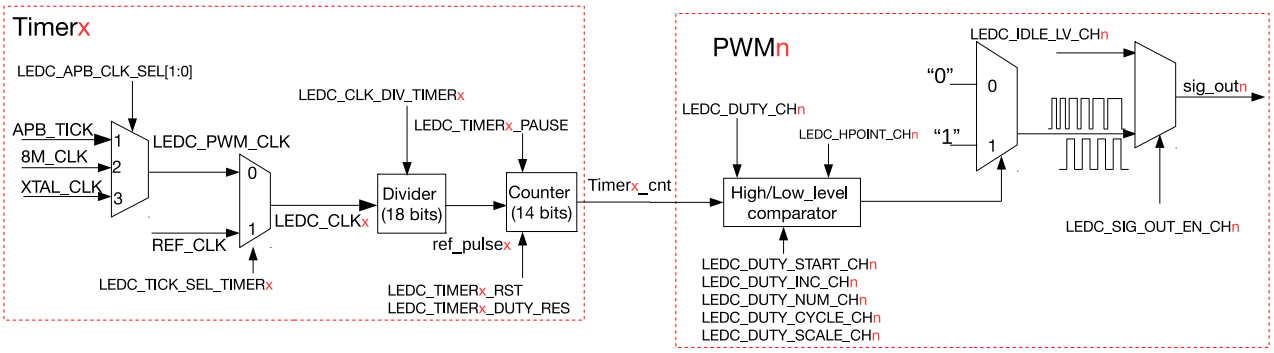


Figure 6-2. LED_PWM generator Diagram

6.3.2 Timers

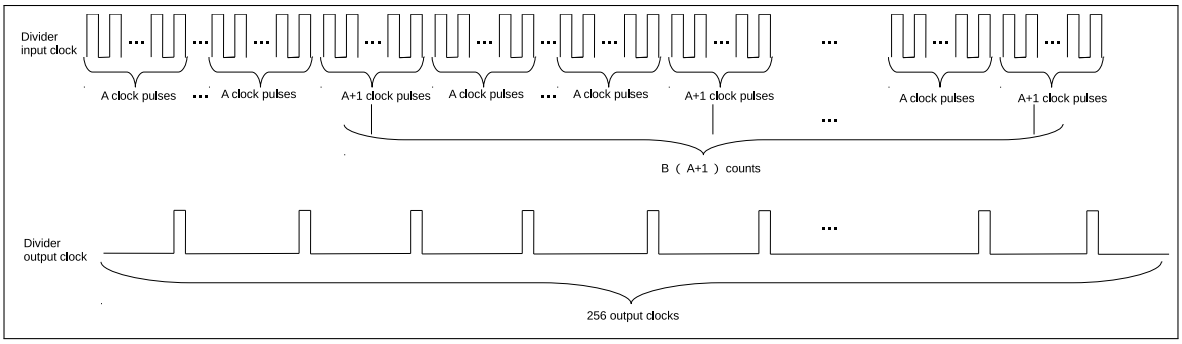


Figure 6-3. LED_PWM Divider

The clock of the LED PWM controller, LEDC_PWM_CLK, has three clock sources: APB_CLK, RTC8M_CLK and XTAL_CLK, selected by configuring LEDC_APB_CLK_SEL[1:0]. The clock of each LED PWM timer, LEDC_CLK*x*, has two clock sources: LEDC_PWM_CLK or REF_TICK. When REF_TICK is used as the clock source of a timer, LEDC_APB_CLK_SEL[1:0] should be set 1 and the cycle of REF_TICK should be an integral multiple of APB_CLK cycles. Otherwise this clock will be not accurate. For more information on the clock sources, please see Chapter Reset and Clock.

The output clock derived from LEDC_CLK*x* is used as the base clock for the counter. The divider's divisor is configured by LEDC_CLK_DIV_TIMER*x*. It is a fixed-point number: the highest 10 bits is the integer part represented as A, while the lowest eight bits is the fractional part represented as B. This divisor LEDC_CLK_DIV*x* is calculated as:

$$LEDC\_CLK\_DIVx = A + \frac{B}{256}$$

When the fractional part B is not 0, the input and output clock of the divider is shown as in figure 6-3. Among the 256 output clocks, B of them are divided by (A+1), whereas the remaining (256-B) are divided by A. Output clocks divided by (A+1) are evenly distributed in the total 256 output clocks.

The LED PWM controller has a 14-bit counter that counts up to $2^{LEDC\_TIMERx\_DUTY\_RES} - 1$. If the counting value reaches $2^{LEDC\_TIMERx\_DUTY\_RES} - 1$, the counter will overflow and restart counting from 0. The counting value can be reset, suspended or read by software. A LEDC_TIMER*x*_OVF_INT interrupt can be generated every time when the counter overflows or when it overflows for LEDC_OVF_NUM_CH*n* times. The interrupt configuration is as follows:

1. Set LEDC_OVF_CNT_EN_CH*n*

2. Configure times of overflow LEDC_OVF_NUM_CH*n*
3. Set LEDC_OVF_CNT_CH*n*_INT_ENA
4. Set LEDC_TIMER*x*_DUTY_RES to enable the timer and wait for a LEDC_OVF_CNT_CH*n*_INT interrupt

The frequency of a PWM generator output signal, sig_out*n*, depends on both the divisor of the divider, as well as the range of the counter:

$$f_{\text{sig\_out}n} = \frac{f_{\text{LEDC\_CLK}x}}{\text{LEDC\_CLK\_DIV}x \cdot 2^{\text{LEDC\_TIMER}x\text{\_DUTY\_RES}}}$$

To change the divisor and times of overflow, you should configure LEDC_CLK_DIV_TIMER*x* and LEDC_TIMER*x*_DUTY_RES respectively, and then set LEDC_TIMER*x*_PARA_UP; otherwise this change is not valid. The newly configured value is updated upon next overflow of the counter.

6.3.3 PWM Generators

As shown in figure 6-2, each PWM generator has a high/low level comparator and two selectors. A PWM generator takes the 14-bit counting value of the selected timer, compares it to values of the comparator Hpoint*n* and Lpoint*n*, and therefore control the level of PWM signals.

- If Timer*x*_cnt == Hpoint*n*, sig_out*n* is 1.
- If Timer*x*_cnt == Lpoint*n*, sig_out*n* is 0.

Hpoint*n* is updated by LEDC_HPOINT_CH*n* when the counter overflows. The initial value of Lpoint*n* is the sum of LEDC_DUTY_CH*n*[18..4] and LEDC_HPOINT_CH*n* when the counter overflows. By configuring these two fields, the relative phase and the duty cycle of the PWM output can be set.

Figure 6-4 illustrates PWM's waveform when the duty cycle is fixed.

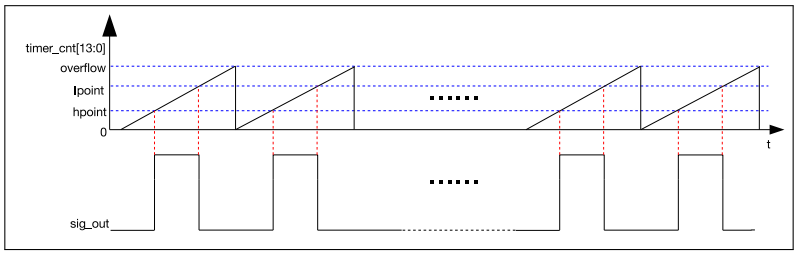


Figure 6-4. LED_PWM Output Signal Diagram

LEDC_DUTY_CH*n* is a fixed-point register with four fractional bits. LEDC_DUTY_CH*n*[18..4] is the integral part used directly for PWM calculation. LEDC_DUTY_CH*n*[3..0] is the fractional part used to dither the output. If LEDC_DUTY_CH*n*[3..0] is non-zero, then among every 16 cycles of sig_out*n*, LEDC_DUTY_CH*n*[3..0] have PWM pulses with width one timer cycle longer than that of (16-LEDC_DUTY_CH*n*[3..0]). This feature effectively increases the resolution of the PWM generator to 18 bits.

6.3.4 Duty Cycle Fading

The PWM generators is able to fade the duty cycle, that is to gradually change the duty cycle from one value to another. This is achieved by configuring LEDC_DUTY_CH*n*, LEDC_DUTY_START_CH*n*, LEDC_DUTY_INC_CH*n*, LEDC_DUTY_NUM_CH*n* and LEDC_DUTY_SCALE_CH*n*.

LEDC_DUTY_START_CH*n* is used to update the value of Lpoint*n*. When this bit is set and the counter overflows, Lpoint*n* increments or decrements automatically, depending on whether the bit LEDC_DUTY_INC_CH*n* is set or cleared.

The duty cycle of sig_out*n* changes every LEDC_DUTY_CYCLE_CH*n* PWM pulse cycles by adding or subtracting the value of LEDC_DUTY_SCALE_CH*n*.

Figure 6-5 is a diagram of fading duty cycle. Upon reaching LEDC_DUTY_NUM_CH*n*, the fade stops and a LEDC_DUTY_CHNG_END_CH*n*_INT interrupt is generated. When configured like this, the duty cycle of sig_out*n* increases by LEDC_DUTY_SCALE_CH*n* every LEDC_DUTY_CYCLE_CH*n* PWM pulse cycles.

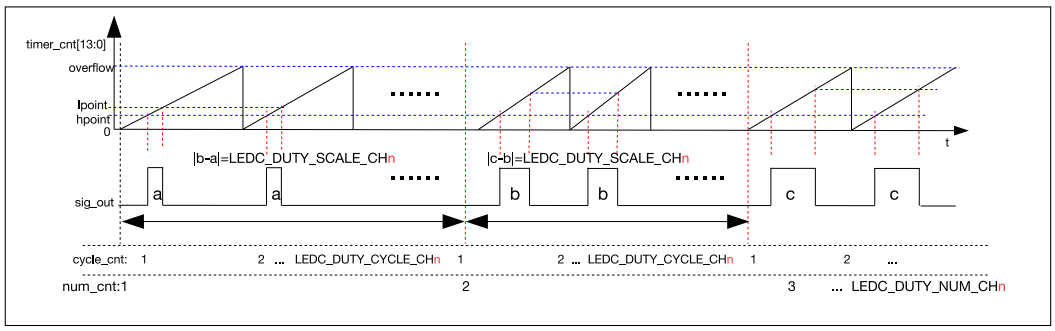


Figure 6-5. Output Signal Diagram of Fading Duty Cycle

LEDC_SIG_OUT_EN_CH*n* used to enable PWM waveform output. When LEDC_SIG_OUT_EN_CH*n* is 0, the level of sig_out*n* is constant as specified in LEDC_IDLE_LV_CH*n*.

If LEDC_HPOINT_CH*n*, LEDC_DUTY_START_CH*n*, LEDC_SIG_OUT_EN_CH*n*, LEDC_TIMER_SEL_CH*n*, LEDC_DUTY_NUM_CH*n*, LEDC_DUTY_CYCLE_CH*n*, LEDC_DUTY_SCALE_CH*n*, LEDC_DUTY_INC_CH*n* and LEDC_OVF_CNT_EN_CH*n* are reconfigured, LEDC_PARA_UP_CH*n* should be set to apply the new configuration.

6.3.5 Interrupts

- LEDC_OVF_CNT_CH*n*_INT: Triggered when the timer counter overflows for LEDC_OVF_NUM_CH*n* times and register LEDC_OVF_CNT_EN_CH*n* is set to 1.
- LEDC_DUTY_CHNG_END_CH*n*_INT: Triggered when a fade on a LED PWM generator has finished.
- LEDC_TIMER*x*_OVF_INT: Triggered when a LED PWM timer has reached its maximum counter value.

6.4 Base Address

Users can access the LED PWM controller with two base addresses, which can be seen in the following table. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 6-1. LED_PWM Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F419000
PeriBUS2	0x60019000

6.5 Register Summary

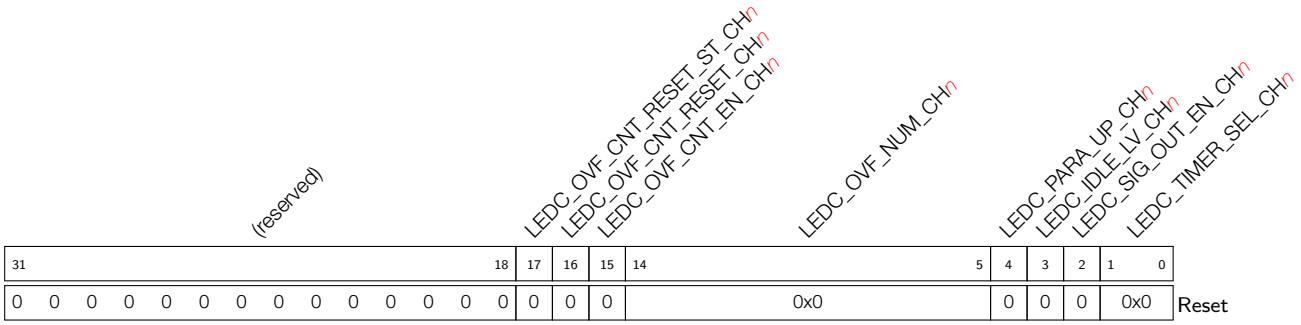
The addresses in the following table are relative to the LED PWM controller base addresses provided in Section 6.4.

Name	Description	Address	Access
Configuration Register			
LEDC_CH0_CONF0_REG	Configuration register 0 for channel 0	0x0000	varies
LEDC_CH0_CONF1_REG	Configuration register 1 for channel 0	0x000C	R/W
LEDC_CH1_CONF0_REG	Configuration register 0 for channel 1	0x0014	varies
LEDC_CH1_CONF1_REG	Configuration register 1 for channel 1	0x0020	R/W
LEDC_CH2_CONF0_REG	Configuration register 0 for channel 2	0x0028	varies
LEDC_CH2_CONF1_REG	Configuration register 1 for channel 2	0x0034	R/W
LEDC_CH3_CONF0_REG	Configuration register 0 for channel 3	0x003C	varies
LEDC_CH3_CONF1_REG	Configuration register 1 for channel 3	0x0048	R/W
LEDC_CH4_CONF0_REG	Configuration register 0 for channel 4	0x0050	varies
LEDC_CH4_CONF1_REG	Configuration register 1 for channel 4	0x005C	R/W
LEDC_CH5_CONF0_REG	Configuration register 0 for channel 5	0x0064	varies
LEDC_CH5_CONF1_REG	Configuration register 1 for channel 5	0x0070	R/W
LEDC_CH6_CONF0_REG	Configuration register 0 for channel 6	0x0078	varies
LEDC_CH6_CONF1_REG	Configuration register 1 for channel 6	0x0084	R/W
LEDC_CH7_CONF0_REG	Configuration register 0 for channel 7	0x008C	varies
LEDC_CH7_CONF1_REG	Configuration register 1 for channel 7	0x0098	R/W
LEDC_CONF_REG	Global ledc configuration register	0x00D0	R/W
Hpoint Register			
LEDC_CH0_HPOINT_REG	High point register for channel 0	0x0004	R/W
LEDC_CH1_HPOINT_REG	High point register for channel 1	0x0018	R/W
LEDC_CH2_HPOINT_REG	High point register for channel 2	0x002C	R/W
LEDC_CH3_HPOINT_REG	High point register for channel 3	0x0040	R/W
LEDC_CH4_HPOINT_REG	High point register for channel 4	0x0054	R/W
LEDC_CH5_HPOINT_REG	High point register for channel 5	0x0068	R/W
LEDC_CH6_HPOINT_REG	High point register for channel 6	0x007C	R/W
LEDC_CH7_HPOINT_REG	High point register for channel 7	0x0090	R/W
Duty Cycle Register			
LEDC_CH0_DUTY_REG	Initial duty cycle for channel 0	0x0008	R/W
LEDC_CH0_DUTY_R_REG	Current duty cycle for channel 0	0x0010	RO
LEDC_CH1_DUTY_REG	Initial duty cycle for channel 1	0x001C	R/W
LEDC_CH1_DUTY_R_REG	Current duty cycle for channel 1	0x0024	RO
LEDC_CH2_DUTY_REG	Initial duty cycle for channel 2	0x0030	R/W

Name	Description	Address	Access
LEDC_CH2_DUTY_R_REG	Current duty cycle for channel 2	0x0038	RO
LEDC_CH3_DUTY_REG	Initial duty cycle for channel 3	0x0044	R/W
LEDC_CH3_DUTY_R_REG	Current duty cycle for channel 3	0x004C	RO
LEDC_CH4_DUTY_REG	Initial duty cycle for channel 4	0x0058	R/W
LEDC_CH4_DUTY_R_REG	Current duty cycle for channel 4	0x0060	RO
LEDC_CH5_DUTY_REG	Initial duty cycle for channel 5	0x006C	R/W
LEDC_CH5_DUTY_R_REG	Current duty cycle for channel 5	0x0074	RO
LEDC_CH6_DUTY_REG	Initial duty cycle for channel 6	0x0080	R/W
LEDC_CH6_DUTY_R_REG	Current duty cycle for channel 6	0x0088	RO
LEDC_CH7_DUTY_REG	Initial duty cycle for channel 7	0x0094	R/W
LEDC_CH7_DUTY_R_REG	Current duty cycle for channel 7	0x009C	RO
Timer Register			
LEDC_TIMER0_CONF_REG	Timer 0 configuration	0x00A0	varies
LEDC_TIMER0_VALUE_REG	Timer 0 current counter value	0x00A4	RO
LEDC_TIMER1_CONF_REG	Timer 1 configuration	0x00A8	varies
LEDC_TIMER1_VALUE_REG	Timer 1 current counter value	0x00AC	RO
LEDC_TIMER2_CONF_REG	Timer 2 configuration	0x00B0	varies
LEDC_TIMER2_VALUE_REG	Timer 2 current counter value	0x00B4	RO
LEDC_TIMER3_CONF_REG	Timer 3 configuration	0x00B8	varies
LEDC_TIMER3_VALUE_REG	Timer 3 current counter value	0x00BC	RO
Interrupt Register			
LEDC_INT_RAW_REG	Raw interrupt status	0x00C0	RO
LEDC_INT_ST_REG	Masked interrupt status	0x00C4	RO
LEDC_INT_ENA_REG	Interrupt enable bits	0x00C8	R/W
LEDC_INT_CLR_REG	Interrupt clear bits	0x00CC	WO
Version Register			
LEDC_DATE_REG	Version control register	0x00FC	R/W

6.6 Registers

Register 6.1: LEDC_CH*n*_CONF0_REG (*n*: 0-7) (0x0000+20n*)**



LEDC_TIMER_SEL_CH*n* This field is used to select one of timers for channel *n*. 0: select timer0 1: select timer1 2: select timer2 3: select timer3 (R/W)

LEDC_SIG_OUT_EN_CH*n* Set this bit to enable signal output on channel *n*. (R/W)

LEDC_IDLE_LV_CH*n* This bit is used to control the output value when channel *n* is inactive. (R/W)

LEDC_PARA_UP_CH*n* This bit is used to update register LEDC_CH*n*_HPOINT and LEDC_CH*n*_DUTY for channel *n*. (WO)

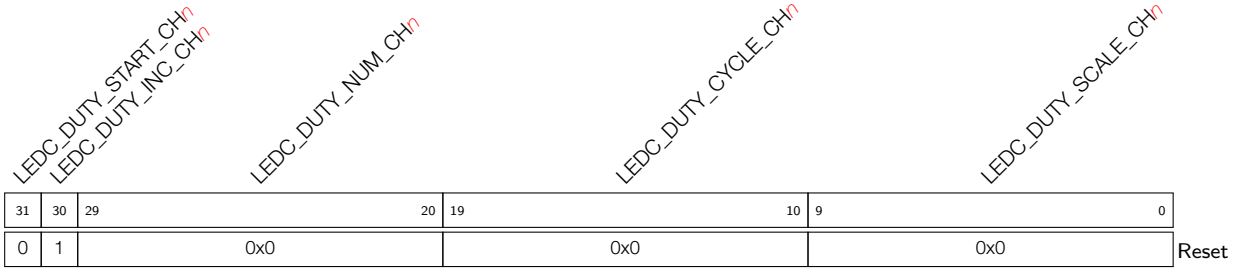
LEDC_OVF_NUM_CH*n* This register is used to configure the maximum value of ovf_cnt. The LEDC_OVF_CNT_CH*n*_INT interrupt will be triggered when the ovf_cnt of channel *n* has reached the value specified by this register. (R/W)

LEDC_OVF_CNT_EN_CH*n* This bit is used to enable the ovf_cnt of channel *n*. (R/W)

LEDC_OVF_CNT_RESET_CH*n* Set this bit to reset the ovf_cnt of channel *n*. (WO)

LEDC_OVF_CNT_RESET_ST_CH*n* This is the status bit of LEDC_OVF_CNT_RESET_CH*n*. (RO)

Register 6.2: LEDC_CH*n*_CONF1_REG (*n*: 0-7) (0x000C+20n*)**



LEDC_DUTY_SCALE_CH*n* This register is used to configure the changing step scale of duty on channel *n*. (R/W)

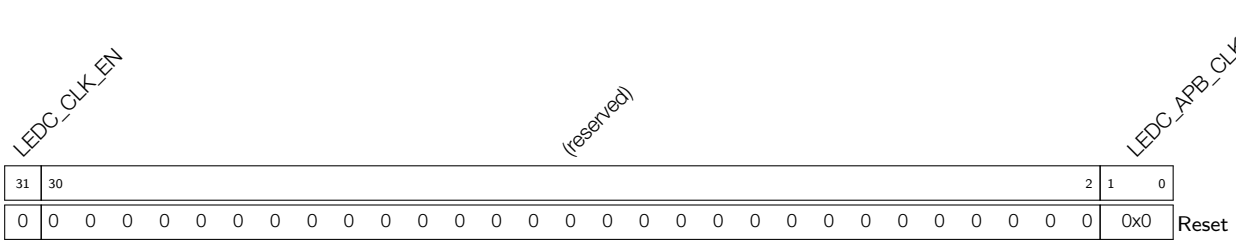
LEDC_DUTY_CYCLE_CH*n* The duty will change every LEDC_DUTY_CYCLE_CH*n* on channel *n*. (R/W)

LEDC_DUTY_NUM_CH*n* This register is used to control the number of times the duty cycle will be changed. (R/W)

LEDC_DUTY_INC_CH*n* This register is used to increase or decrease the duty of output signal on channel *n*. 1: Increase; 0: Decrease. (R/W)

LEDC_DUTY_START_CH*n* Other configured fields in LEDC_CH*n*_CONF1_REG will start to take effect when this bit is set to 1. (R/W)

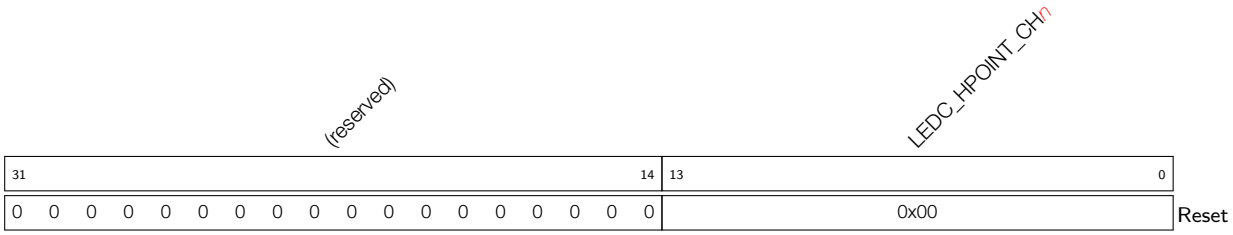
Register 6.3: LEDC_CONF_REG (0x00D0)



LEDC_APB_CLK_SEL This bit is used to set the frequency of SLOW_CLK for all the 4 timers . 2'd1: APB_CLK 2'd2: RTC8M_CLK 2'd3: XTAL_CLK (R/W)

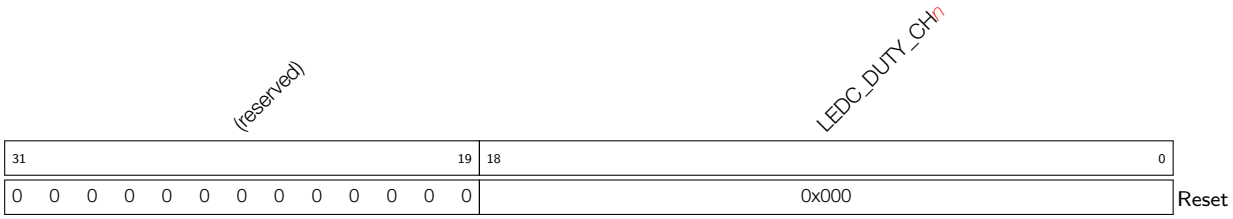
LEDC_CLK_EN This bit is used to control clock. 1'b1: Force clock on for register. 1'h0: Support clock only when application writes registers. (R/W)

Register 6.4: LEDC_CH*n*_HPOINT_REG (*n*: 0-7) (0x0004+20n*)**



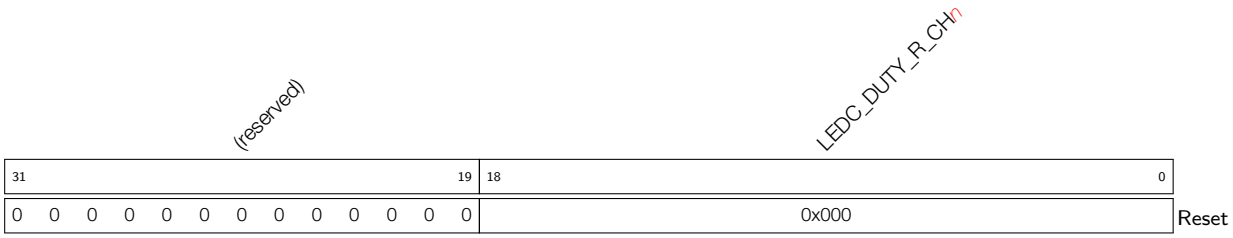
LEDC_HPOINT_CH*n* The output value changes to high when the selected timers has reached the value specified by this register. (R/W)

Register 6.5: LEDC_CH*n*_DUTY_REG (*n*: 0-7) (0x0008+20n*)**



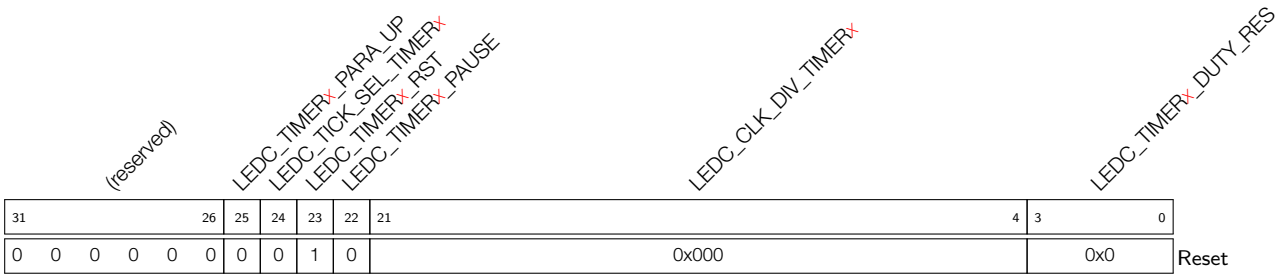
LEDC_DUTY_CH*n* This register is used to change the output duty by controlling the Lpoint. The output value turns to low when the selected timers has reached the Lpoint. (R/W)

Register 6.6: LEDC_CH*n*_DUTY_R_REG (*n*: 0-7) (0x0010+20n*)**



LEDC_DUTY_R_CH*n* This register stores the current duty of output signal on channel *n*. (RO)

Register 6.7: LEDC_TIMERx_CONF_REG (x: 0-3) (0x00A0+8*x)



LEDC_TIMERx_DUTY_RES This register is used to control the range of the counter in timer *x*. (R/W)

LEDC_CLK_DIV_TIMERx This register is used to configure the divisor for the divider in timer *x*. The least significant eight bits represent the fractional part. (R/W)

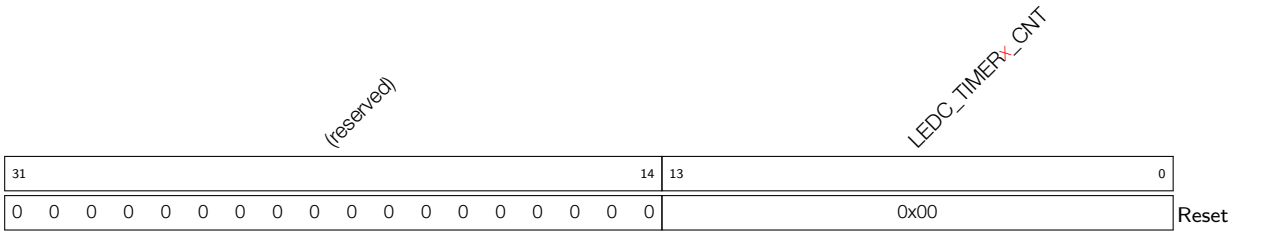
LEDC_TIMERx_PAUSE This bit is used to suspend the counter in timer *x*. (R/W)

LEDC_TIMERx_RST This bit is used to reset timer *x*. The counter will show 0 after reset. (R/W)

LEDC_TICK_SEL_TIMERx This bit is used to select clock for timer *x*. When this bit is set to 1 LEDC_APB_CLK_SEL[1:0] should be 1, otherwise the timer clock may be not accurate. 1'h0: SLOW_CLK 1'h1: REF_TICK (R/W)

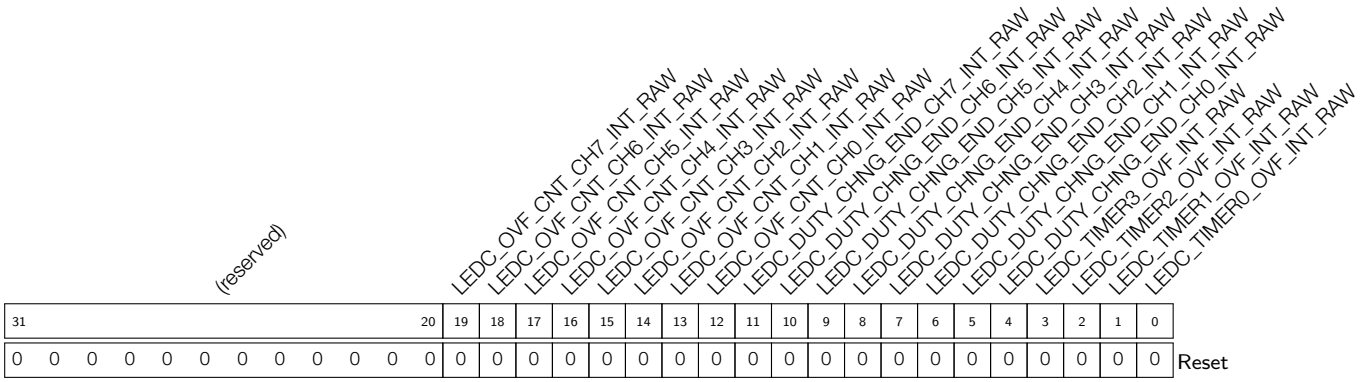
LEDC_TIMERx_PARA_UP Set this bit to update LEDC_CLK_DIV_TIMERx and LEDC_TIMERx_DUTY_RES. (WO)

Register 6.8: LEDC_TIMERx_VALUE_REG (x: 0-3) (0x00A4+8*x)



LEDC_TIMERx_CNT This register stores the current counter value of timer *x*. (RO)

Register 6.9: LEDC_INT_RAW_REG (0x00C0)

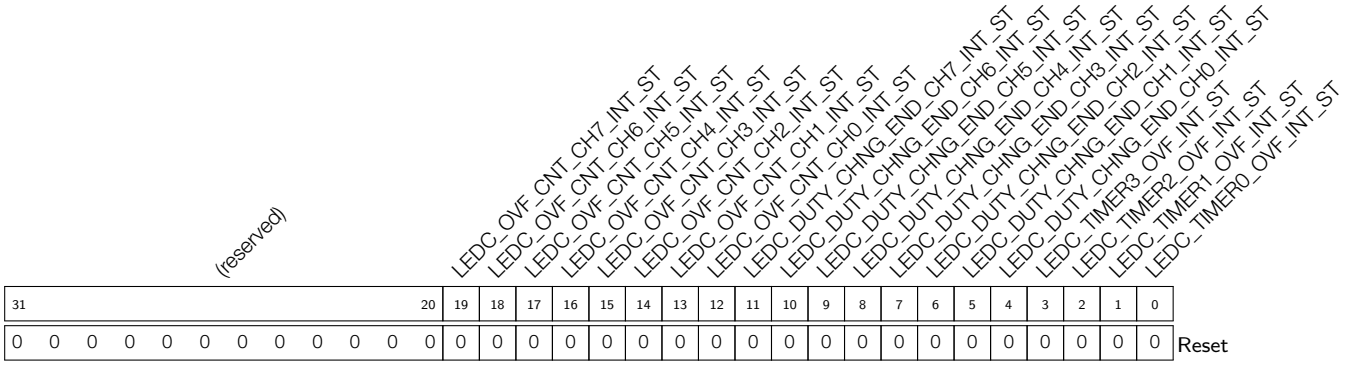


LEDC_TIMER*x*_OVF_INT_RAW Triggered when the timer*x* has reached its maximum counter value. (RO)

LEDC_DUTY_CHNG_END_CH*n*_INT_RAW Interrupt raw bit for channel *n*. Triggered when the gradual change of duty has finished. (RO)

LEDC_OVF_CNT_CH*n*_INT_RAW Interrupt raw bit for channel *n*. Triggered when the ovf_cnt has reached the value specified by LEDC_OVF_NUM_CH*n*. (RO)

Register 6.10: LEDC_INT_ST_REG (0x00C4)

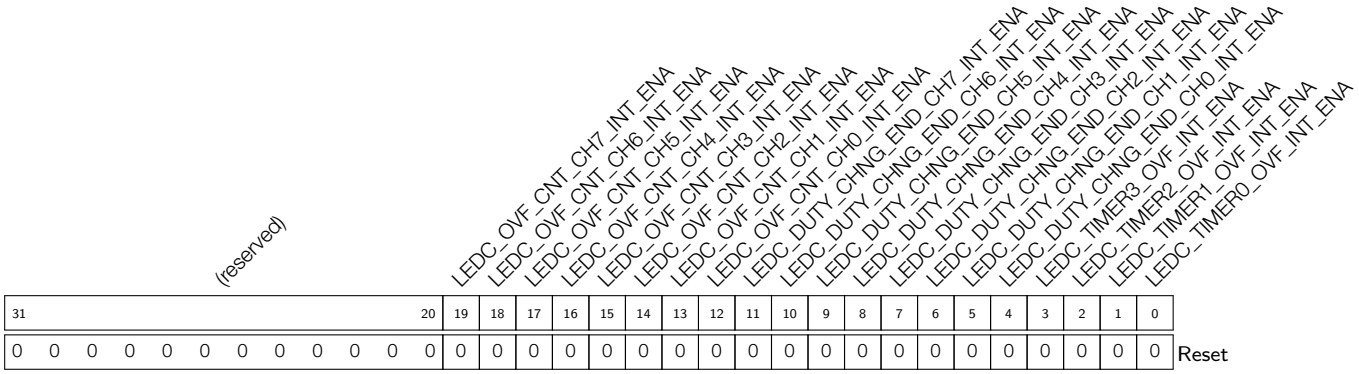


LEDC_TIMER*x*_OVF_INT_ST This is the masked interrupt status bit for the LEDC_TIMER*x*_OVF_INT interrupt when LEDC_TIMER*x*_OVF_INT_ENA is set to 1. (RO)

LEDC_DUTY_CHNG_END_CH*n*_INT_ST This is the masked interrupt status bit for the LEDC_DUTY_CHNG_END_CH*n*_INT interrupt when LEDC_DUTY_CHNG_END_CH*n*_INT_ENAis set to 1. (RO)

LEDC_OVF_CNT_CH*n*_INT_ST This is the masked interrupt status bit for the LEDC_OVF_CNT_CH*n*_INT interrupt when LEDC_OVF_CNT_CH*n*_INT_ENA is set to 1. (RO)

Register 6.11: LEDC_INT_ENA_REG (0x00C8)

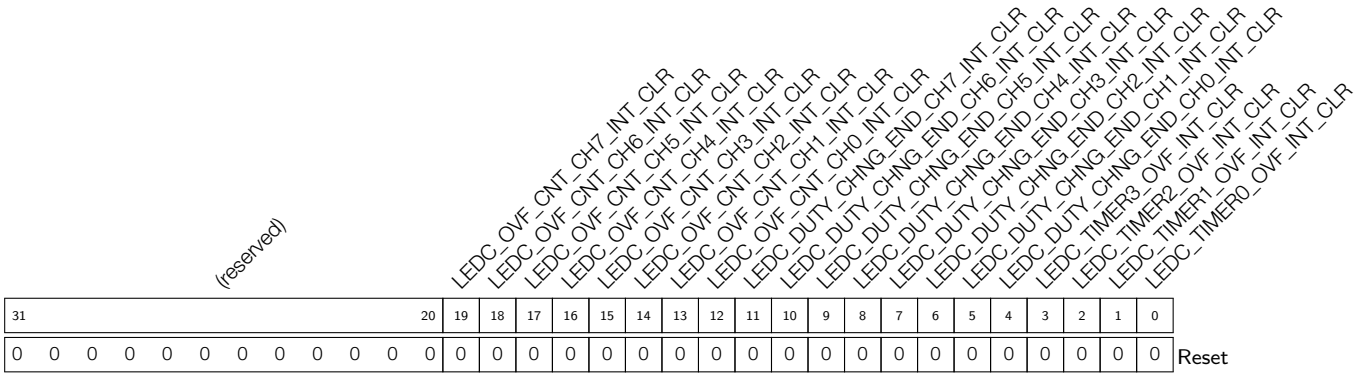


LEDC_TIMER*x*_OVF_INT_ENA The interrupt enable bit for the LEDC_TIMER*x*_OVF_INT interrupt. (R/W)

LEDC_DUTY_CHNG_END_CH*n*_INT_ENA The interrupt enable bit for the LEDC_DUTY_CHNG_END_CH*n*_INT interrupt. (R/W)

LEDC_OVF_CNT_CH*n*_INT_ENA The interrupt enable bit for the LEDC_OVF_CNT_CH*n*_INT interrupt. (R/W)

Register 6.12: LEDC_INT_CLR_REG (0x00CC)

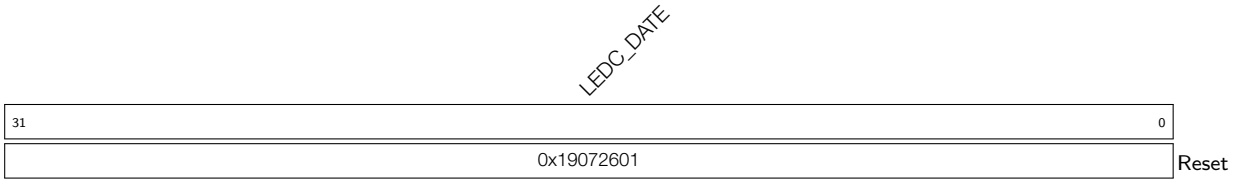


LEDC_TIMER*x*_OVF_INT_CLR Set this bit to clear the LEDC_TIMER*x*_OVF_INT interrupt. (WO)

LEDC_DUTY_CHNG_END_CH*n*_INT_CLR Set this bit to clear the LEDC_DUTY_CHNG_END_CH*n*_INT interrupt. (WO)

LEDC_OVF_CNT_CH*n*_INT_CLR Set this bit to clear the LEDC_OVF_CNT_CH*n*_INT interrupt. (WO)

Register 6.13: LEDC_DATE_REG (0x00FC)



LEDC_DATE This is the version control register. (R/W)

7. Remote Control Peripheral

7.1 Introduction

The RMT (Remote Control) module is designed to send/receive infrared remote control signals that support for a variety of remote control protocols. The RMT module converts pulse codes stored in the module's built-in RAM into output signals, or converts input signals into pulse codes and stores them back in RAM. In addition, the RMT module optionally modulates its output signals with a carrier wave, or optionally filters its input signals.

The RMT module has four channels, numbered from zero to three. Each channel has the same functionality controlled by dedicated set of registers and is able to independently transmit or receive data. Registers in each channel are indicated by *n* which is used as a placeholder for the channel number.

7.2 Functional Description

7.2.1 RMT Architecture

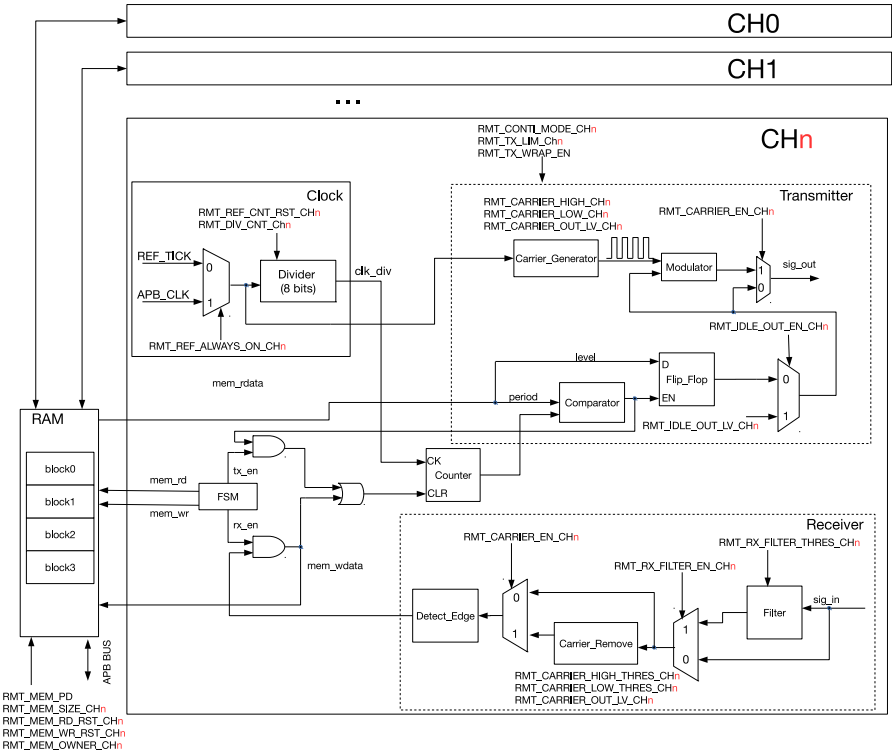


Figure 7-1. RMT Architecture

The RMT module contains four independent channels. Each channel has a clock divider, a counter, a transmitter and a receiver. As for the transmitter and the receiver of a single channel, only one of them can be active. The four channels share a 256 x 32-bit RAM.

7.2.2 RMT RAM

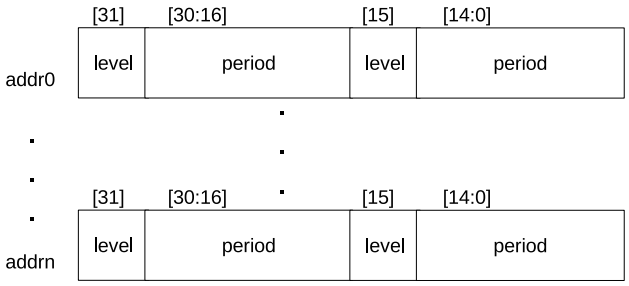


Figure 7-2. Format of Pulse Code in RAM

The format of pulse code in RAM is shown in Figure 7-2. Each pulse code contains a 16-bit entry with two fields, level and period. Level (0 or 1) indicates a high-/low-level value was received or is going to be sent, while "period" points out the clock cycles (see Figure 7-1 clk_div) for which the level lasts. A zero period is interpreted as a transmission end-marker.

The RAM is divided into four 64 x 32-bit blocks. By default, each channel uses one block (block zero for channel zero, block one for channel one, and so on). Usually, only one block of 64 x 32-bit worth of data can be sent or received in channel *n*. If the data size is larger than this block size, users can configure the channel to enable wrap mode or to use more blocks by setting RMT_MEM_SIZE_CH*n*. Setting RMT_MEM_SIZE_CH*n* > 1 will prompt channel *n* to use the memory of subsequent channels, block (*n*) ~ block (*n* + RMT_MEM_SIZE_CH*n* -1). If so, the subsequent channels *n* +1 ~ *n* + RMT_MEM_SIZE_CH*n* - 1 cannot be used once their RAM blocks are occupied. Note that the RAM used by each channel is mapped from low address to high address. In such mode, channel 0 is able to use the RAM blocks for channels 1, 2 and 3 by setting RMT_MEM_SIZE_CH*n*, but channel 3 cannot use the blocks for channels 0, 1, or 2.

The RMT RAM can be accessed via APB bus, or read by the transmitter and written by the receiver. To protect a receiver from overwriting the blocks a transmitter is about to transmit, RMT_MEM_OWNER_CH*n* can be configured to designate the block's owner, be it a transmitter or receiver. This way, if this ownership is violated, an RMT_MEM_OWNER_ERR_CH*n* flag will be generated.

When the RMT module is inactive, the RAM can be put into low-power mode by setting RMT_MEM_FORCE_PD.

7.2.3 Clock

The drive clock of a divider is generated by taking either the APB_CLK or REF_TICK according to the state of RMT_REF_ALWAYS_ON_CH*n*. (For more information on clock sources, please see Chapter Reset and Clock). Divider value is normally equal to the value of RMT_DIV_CNT_CH*n*, except value 0 that represents divider 256. The clock divider can be reset to zero by clearing RMT_REF_CNT_RST_CH*n*. The clock generated from the divider can be used by the clock counter (see Figure 7-2).

7.2.4 Transmitter

When RMT_TX_START_CH*n* is set to 1, the transmitter of channel *n* will start reading and sending pulse codes from the starting address of its RAM block. The codes are sent starting from low-address entry. The transmitter will stop the transmission, return to idle state and generate an RMT_CH*n*_TX_END_INT interrupt, when an end-marker (a zero period) is encountered. Also, setting RMT_TX_STOP_CH*n* to 1 stops the transmission and

immediately sets the transmitter back to idle. The output level of a transmitter in idle state is determined by the "level" field of the end-marker or by the content of RMT_IDLE_OUT_LV_CH*n*, depending on the configuration of RMT_IDLE_OUT_EN_CH*n*.

To transmit more pulse codes than can be fitted in the channel's RAM, users can enable wrap mode by configuring RMT_MEM_TX_WRAP_EN. In this mode, when the transmitter has reached the end-marker in the channel's memory, it will loop back to the first byte. For example, if RMT_MEM_SIZE_CH*n* is set to 1, the transmitter will start sending data from the address 64 * *n*, and then the data from higher RAM address. Once the transmitter finishes sending the data from (64 * (*n* +1) - 1), it will continue sending data from 64 * *n* till encounters an end-marker. Wrap mode is also applicable for RMT_MEM_SIZE_CH*n* > 1.

An RMT_CH*n*_TX_THR_EVENT_INT interrupt will be generated whenever the size of transmitted pulse codes is larger than or equal to the value set by RMT_TX_LIM_CH*n*. In wrap mode, RMT_TX_LIM_CH*n* can be set to a half or a fraction of the size of the channel's RAM block. When an RMT_CH*n*_TX_THR_EVENT_INT interrupt is detected, the already used RAM region should be updated by subsequent user defined events. Therefore, when the wrap mode happens the transmitter will seamlessly continue sending the new events.

The output of the transmitter can be modulated using a carrier wave by setting RMT_CARRIER_EN_CH*n*. The carrier waveform is configurable. In a carrier cycle, the high level lasts for (RMT_CARRIER_HIGH_CH*n* +1) clock cycles of APB_CLK or REF_TICK, while the low level lasts for (RMT_CARRIER_LOW_CH*n* +1) clock cycles of APB_CLK or REF_TICK. When RMT_CARRIER_OUT_LV_CH*n* is set to 1, carrier wave will be added on high-level output signals; while RMT_CARRIER_OUT_LV_CH*n* is set to 0, carrier wave will be added on low-level output signals. Carrier wave can be added on output signals during modulation, or just added on valid pulse codes (the data stored in RAM), which can be set by configuring RMT_CARRIER_EFF_EN_CH*n*.

The continuous transmission of the transmitter can be enabled by setting RMT_TX_CONTI_MODE_CH*n*. When this register is set, the transmitter will send the pulse codes from RAM in loops. If RMT_TX_LOOP_CNT_EN_CH*n* is set to 1, the transmitter will start counting loop times. Once the counting reaches the value of register RMT_TX_LOOP_NUM_CH*n*, an RMT_CH*n*_TX_LOOP_INT interrupt will be generated.

Setting RMT_TX_SIM_EN to 1 will enable multiple channels to start sending data simultaneously. RMT_TX_SIM_CH*n* will choose which multiple channels are enabled to send data simultaneously.

7.2.5 Receiver

When RMT_RX_EN_CH*n* is set to 1, the receiver in channel *n* becomes active, detecting signal levels and measuring clock cycles the signals lasts for. These data will be written in RAM in the form of pulse codes. Receiving ends, when the receiver detects no change in a signal level for a number of clock cycles more than the value set by RMT_IDLE_THRES_CH*n*. The receiver will return to idle state and generate an RMT_CH*n*_RX_END_INT interrupt.

The receiver has an input signal filter which can be enabled by configuring RMT_RX_FILTER_EN_CH*n*. The filter samples input signals continuously, and will detect the signals which remain unchanged for a continuous RMT_RX_FILTER_THRES_CH*n* APB clock cycles as valid, otherwise, the signals will be detected as invalid. Only the valid signals can pass through this filter. The filter will remove pulses with a length of less than RMT_RX_FILTER_THRES_CH*n* APB clock cycles.

7.2.6 Interrupts

- RMT_CH*n*_ERR_INT: Triggered when channel *n* does not read or write data correctly. For example, if the transmitter still tries to read data from RAM when the RAM is empty, or the receiver still tries to write data into RAM when the RAM is full, this interrupt will be triggered.
- RMT_CH*n*_TX_THR_EVENT_INT: Triggered when the amount of data the transmitter has sent matches the value of RMT_TX_LIM_CH*n*.
- RMT_CH*n*_TX_END_INT: Triggered when the transmitter has finished transmitting signals.
- RMT_CH*n*_RX_END_INT: Triggered when the receiver has finished receiving signals.
- RMT_CH*n*_TX_LOOP_INT: Triggered when the loop counting reaches the value set by RMT_TX_LOOP_NUM_CH*n*.

7.3 Base Address

Users can access RMT with two base addresses, which can be seen in the following table. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 7-1. RMT Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F416000
PeriBUS2	0x60016000

7.4 Register Summary

The addresses in the following table are relative to RMT base addresses provided in Section 7.3.

Name	Description	Address	Access
Configuration registers			
RMT_CH0CONF0_REG	Channel 0 configuration register 0	0x0010	R/W
RMT_CH0CONF1_REG	Channel 0 configuration register 1	0x0014	varies
RMT_CH1CONF0_REG	Channel 1 configuration register 0	0x0018	R/W
RMT_CH1CONF1_REG	Channel 1 configuration register 1	0x001C	varies
RMT_CH2CONF0_REG	Channel 2 configuration register 0	0x0020	R/W
RMT_CH2CONF1_REG	Channel 2 configuration register 1	0x0024	varies
RMT_CH3CONF0_REG	Channel 3 configuration register 0	0x0028	R/W
RMT_CH3CONF1_REG	Channel 3 configuration register 1	0x002C	varies
RMT_APB_CONF_REG	RMT APB configuration register	0x0080	R/W
RMT_REF_CNT_RST_REG	RMT clock divider reset register	0x0088	R/W
RMT_CH0_RX_CARRIER_RM_REG	Channel 0 carrier remove register	0x008C	R/W
RMT_CH1_RX_CARRIER_RM_REG	Channel 1 carrier remove register	0x0090	R/W
RMT_CH2_RX_CARRIER_RM_REG	Channel 2 carrier remove register	0x0094	R/W
RMT_CH3_RX_CARRIER_RM_REG	Channel 3 carrier remove register	0x0098	R/W
Carrier wave duty cycle registers			

Name	Description	Address	Access
RMT_CH0CARRIER_DUTY_REG	Channel 0 duty cycle configuration register	0x0060	R/W
RMT_CH1CARRIER_DUTY_REG	Channel 1 duty cycle configuration register	0x0064	R/W
RMT_CH2CARRIER_DUTY_REG	Channel 2 duty cycle configuration register	0x0068	R/W
RMT_CH3CARRIER_DUTY_REG	Channel 3 duty cycle configuration register	0x006C	R/W
Tx event configuration registers			
RMT_CH0_TX_LIM_REG	Channel 0 Tx event configuration register	0x0070	varies
RMT_CH1_TX_LIM_REG	Channel 1 Tx event configuration register	0x0074	varies
RMT_CH2_TX_LIM_REG	Channel 2 Tx event configuration register	0x0078	varies
RMT_CH3_TX_LIM_REG	Channel 3 Tx event configuration register	0x007C	varies
RMT_TX_SIM_REG	Enable RMT simultaneous transmission	0x0084	R/W
Status registers			
RMT_CH0STATUS_REG	Channel 0 status register	0x0030	RO
RMT_CH1STATUS_REG	Channel 1 status register	0x0034	RO
RMT_CH2STATUS_REG	Channel 2 status register	0x0038	RO
RMT_CH3STATUS_REG	Channel 3 status register	0x003C	RO
RMT_CH0ADDR_REG	Channel 0 address register	0x0040	RO
RMT_CH1ADDR_REG	Channel 1 address register	0x0044	RO
RMT_CH2ADDR_REG	Channel 2 address register	0x0048	RO
RMT_CH3ADDR_REG	Channel 3 address register	0x004C	RO
Version register			
RMT_DATE_REG	Version control register	0x00FC	R/W
FIFO R/W registers			
RMT_CH0DATA_REG	Read and write data for channel 0 via APB FIFO	0x0000	RO
RMT_CH1DATA_REG	Read and write data for channel 1 via APB FIFO	0x0004	RO
RMT_CH2DATA_REG	Read and write data for channel 2 via APB FIFO	0x0008	RO
RMT_CH3DATA_REG	Read and write data for channel 3 via APB FIFO	0x000C	RO
Interrupt registers			
RMT_INT_RAW_REG	Raw interrupt status register	0x0050	RO
RMT_INT_ST_REG	Masked interrupt status register	0x0054	RO
RMT_INT_ENA_REG	Interrupt enable register	0x0058	R/W
RMT_INT_CLR_REG	Interrupt clear register	0x005C	WO

7.5 Registers

Register 7.1: RMT_CH*n*CONF0_REG (*n*: 0-3) (0x0010+8n*)**

RMT_DIV_CNT_CH*n* This field is used to configure clock divider for channel *n*. (R/W)

RMT_IDLE_THRES_CH*n* Receiving ends when no edge is detected on input signals for continuous clock cycles larger than this register value. (R/W)

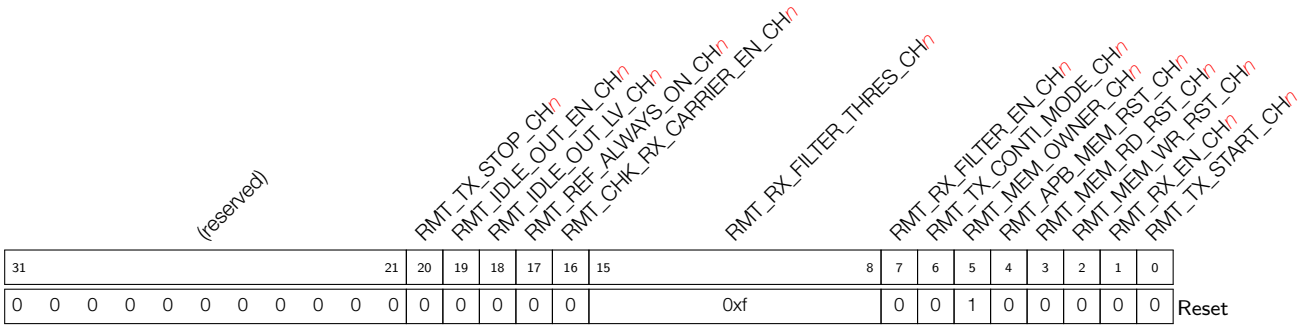
RMT_MEM_SIZE_CH*n* This field is used to configure the maximum blocks allocated to channel *n*. The valid range is from 1 ~ 4-*n*. (R/W)

RMT_CARRIER_EFF_EN_CH*n* 1: Add carrier modulation on output signals only at data sending state for channel *n*. 0: Add carrier modulation on signals at all states for channel *n*. States here include idle state(ST_IDLE), reading data from RAM (ST_RD_MEM), and sending data stored in RAM (ST_SEND). Only valid when RMT_CARRIER_EN_CH*n* is set to 1. (R/W)

RMT_CARRIER_EN_CH*n* This bit is used to enable carrier modulation for channel *n*. 1: Add carrier modulation on output signals. 0: No carrier modulation is added on output signals. (R/W)

RMT_CARRIER_OUT_LV_CH*n* This bit is used to configure the position of carrier wave for channel *n*. 1'h0: Add carrier wave on low-level output signals. 1'h1: Add carrier wave on high-level output signals. (R/W)

Register 7.2: RMT_CH*n*CONF1_REG (*n*: 0-3) (0x0014+8n*)**



RMT_TX_START_CH*n* Set this bit to start sending data on channel *n*. (R/W)

RMT_RX_EN_CH*n* Set this bit to enable receiver to receive data on channel *n*. (R/W)

RMT_MEM_WR_RST_CH*n* Set this bit to reset RAM write address accessed by the receiver for channel *n*. (WO)

RMT_MEM_RD_RST_CH*n* Set this bit to reset RAM read address accessed by the transmitter for channel *n*. (WO)

RMT_MEM_OWNER_CH*n* This bit marks the ownership of channel *n*'s RAM block. 1'h1: Receiver is using the RAM. 1'h0: Transmitter is using the RAM. (R/W)

RMT_TX_CONTI_MODE_CH*n* Set this bit to restart transmission in continuous mode from the first data in channel *n*. (R/W)

RMT_RX_FILTER_EN_CH*n* Set this bit to enable the receiver's filter for channel *n*. (R/W)

RMT_RX_FILTER_THRES_CH*n* Set this field to ignore the input pulse when its width is less than RMT_RX_FILTER_THRES_CH*n* APB clock cycles in receive mode. (R/W)

RMT_CHK_RX_CARRIER_EN_CH*n* Set this bit to enable memory loop read mode when carrier modulation is enabled for channel *n*. (R/W)

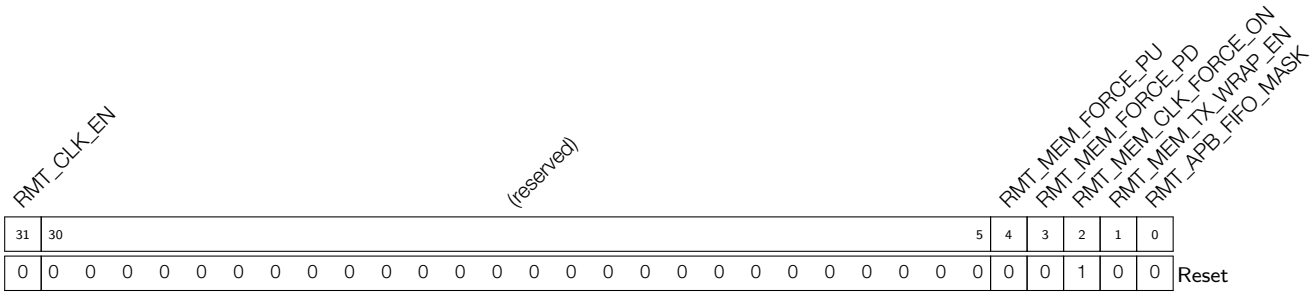
RMT_REF_ALWAYS_ON_CH*n* Set this bit to select a base clock for channel *n*. 1'h1: APB_CLK; 1'h0: REF_TICK (R/W)

RMT_IDLE_OUT_LV_CH*n* This bit configures the level of output signals in channel *n* when the transmitter is in idle state. (R/W)

RMT_IDLE_OUT_EN_CH*n* This is the output enable bit for channel *n* in idle state. (R/W)

RMT_TX_STOP_CH*n* Set this bit to stop the transmitter of channel *n* sending data out. (R/W)

Register 7.3: RMT_APB_CONF_REG (0x0080)



RMT_APB_FIFO_MASK 1'h1: Access memory directly; 1'h0: Access memory via APB FIFO. (R/W)

RMT_MEM_TX_WRAP_EN Set this bit to enable wrap mode. (R/W)

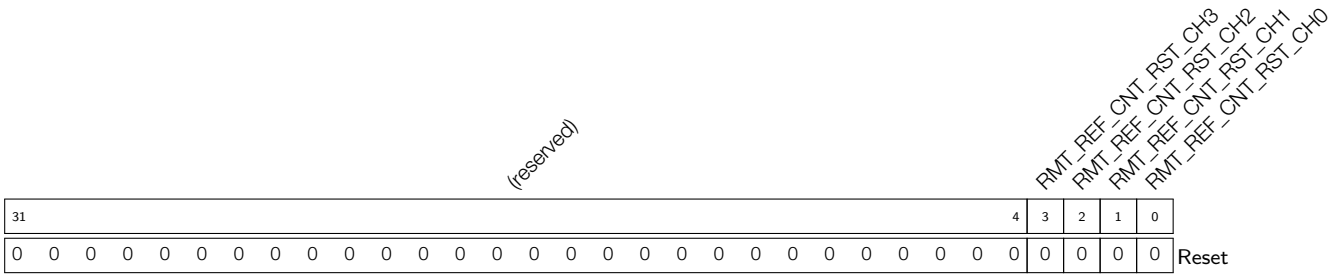
RMT_MEM_CLK_FORCE_ON Set this bit to enable the clock for RAM when RMT module starts working; disable this clock when RMT stops working, to achieve low-power scheme. (R/W)

RMT_MEM_FORCE_PD Set this bit to power down RMT memory. (R/W)

RMT_MEM_FORCE_PU 1: Disable RAM's Light-sleep power down function. 0: power down RMT RAM when RMT is in Light-sleep mode. (R/W)

RMT_CLK_EN Clock gating enable bit for RMT registers to achieve low-power scheme. 1: Power up drive clock for RMT registers. 0: Power down drive clock for RMT registers. (R/W)

Register 7.4: RMT_REF_CNT_RST_REG (0x0088)



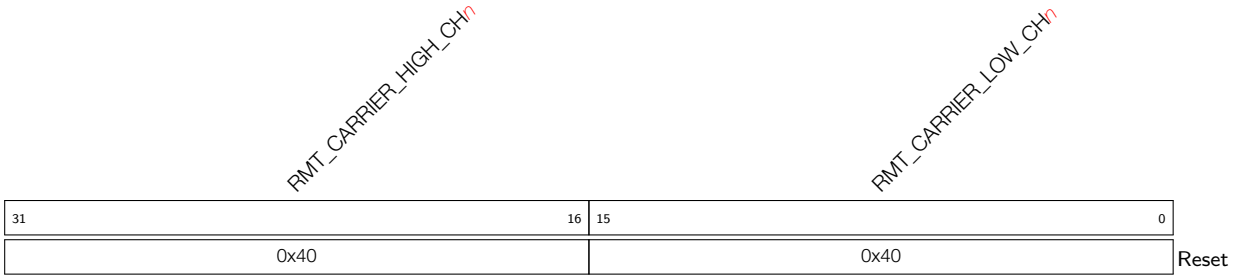
RMT_REF_CNT_RST_CH*n* This bit is used to reset the clock divider of channel *n*. (R/W)

Register 7.5: RMT_CH*n*_RX_CARRIER_RM_REG (*n*: 0-3) (0x008C+4n*)**

RMT_CARRIER_LOW_THRES_CH*n* The low level period in carrier modulation mode is (RMT_CARRIER_LOW_THRES_CH*n* + 1) clock cycles for channel *n*. (R/W)

RMT_CARRIER_HIGH_THRES_CH*n* The high level period in carrier modulation mode is (RMT_CARRIER_HIGH_THRES_CH*n* + 1) clock cycles for channel *n*. (R/W)

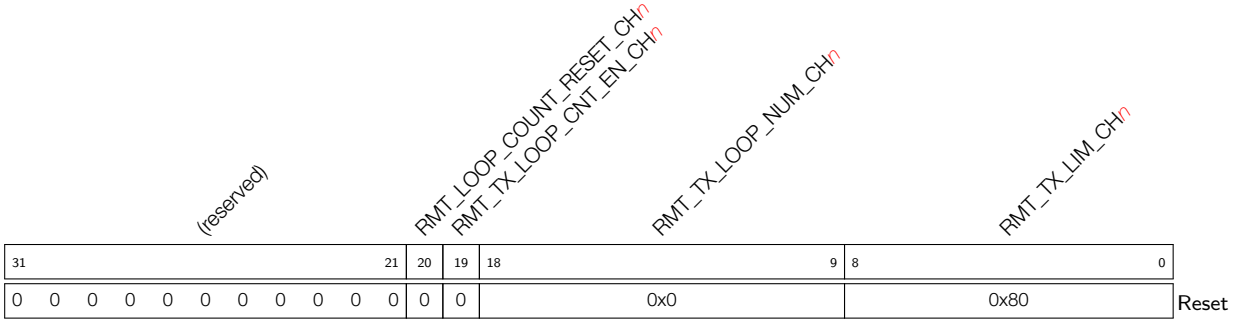
Register 7.6: RMT_CH*n*CARRIER_DUTY_REG (*n*: 0-3) (0x0060+4n*)**



RMT_CARRIER_LOW_CH*n* This field is used to configure the clock cycles of carrier wave at low level for channel *n*. (R/W)

RMT_CARRIER_HIGH_CH*n* This field is used to configure the clock cycles of carrier wave at high level for channel *n*. (R/W)

Register 7.7: RMT_CH*n*_TX_LIM_REG (*n*: 0-3) (0x0070+4n*)**



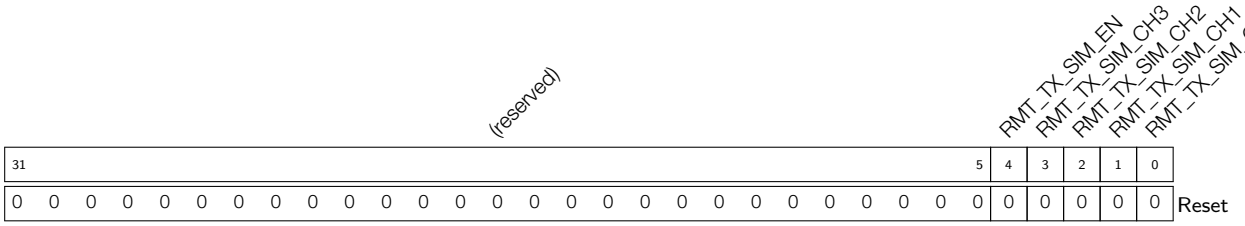
RMT_TX_LIM_CH*n* This field is used to configure the maximum entries channel *n* can send out. When RMT_MEM_SIZE_CH*n* = 1, this field can be set to any value among 0 ~ 128 (64 * 32/16 = 128); when RMT_MEM_SIZE_CH*n* > 1, this field can be set to any value among (0 ~ 128) * RMT_MEM_SIZE_CH*n*. (R/W)

RMT_TX_LOOP_NUM_CH*n* This field is used to configure the maximum loop times when continuous transmission mode is enabled. (R/W)

RMT_TX_LOOP_CNT_EN_CH*n* This bit is used to enable loop counting. (R/W)

RMT_LOOP_COUNT_RESET_CH*n* This bit is used to reset loop counting when continuous transmission mode is valid. (WO)

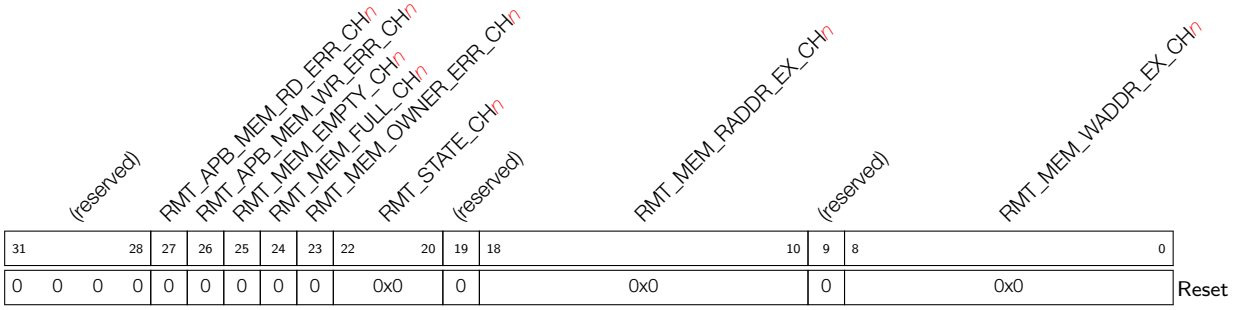
Register 7.8: RMT_TX_SIM_REG (0x0084)



RMT_TX_SIM_CH*n* Set this bit to enable channel *n* to start sending data simultaneously with other enabled channels. (R/W)

RMT_TX_SIM_EN This bit is used to enable multiple channels to start sending data simultaneously. (R/W)

Register 7.9: RMT_CH*n*STATUS_REG (*n*: 0-3) (0x0030+4n*)**



RMT_MEM_WADDR_EX_CH*n* This field records the memory address offset when receiver of channel *n* is using the RAM. (RO)

RMT_MEM_RADDR_EX_CH*n* This field records the memory address offset when transmitter of channel *n* is using the RAM. (RO)

RMT_STATE_CH*n* This field records the FSM status of channel *n*. (RO)

RMT_MEM_OWNER_ERR_CH*n* This status bit will be set when the ownership of memory block is violated. (RO)

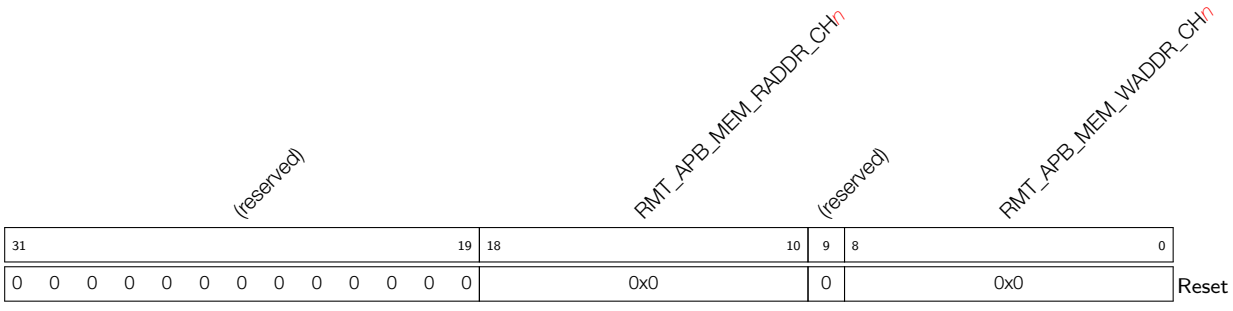
RMT_MEM_FULL_CH*n* This status bit will be set if the receiver receives more data than the memory allows. (RO)

RMT_MEM_EMPTY_CH*n* This status bit will be set when the data to be sent is more than the memory allows and the wrap mode is disabled. (RO)

RMT_APB_MEM_WR_ERR_CH*n* This status bit will be set if the offset address is out of memory size when channel *n* writes RAM via APB bus. (RO)

RMT_APB_MEM_RD_ERR_CH*n* This status bit will be set if the offset address is out of memory size when channel *n* reads RAM via APB bus. (RO)

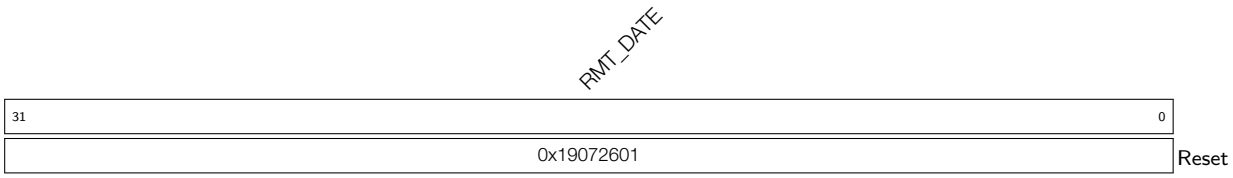
Register 7.10: RMT_CH*n*ADDR_REG (*n*: 0-3) (0x0040+4n*)**



RMT_APB_MEM_WADDR_CH*n* This field records the memory address offset when channel *n* writes RAM via APB bus. (RO)

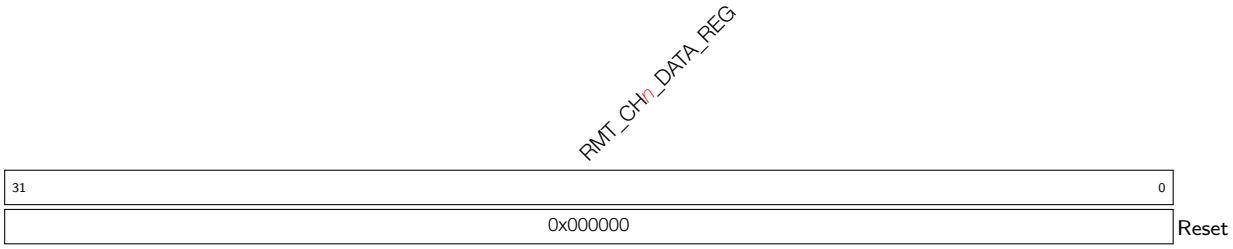
RMT_APB_MEM_RADDR_CH*n* This field records the memory address offset when channel *n* reads RAM via APB bus. (RO)

Register 7.11: RMT_DATE_REG (0x00FC)



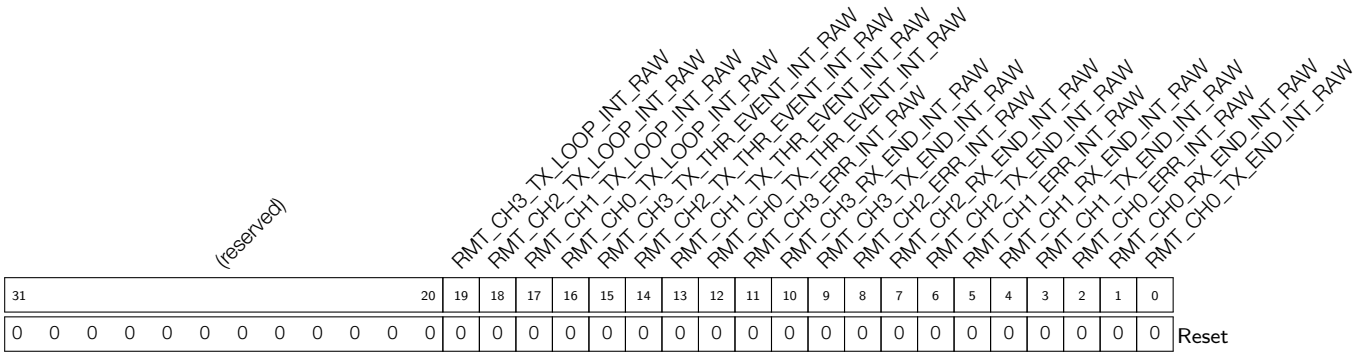
RMT_DATE Version control register. (R/W)

Register 7.12: RMT_CH*n*DATA_REG (*n*: 0-3) (0x0000+4n*)**



RMT_CH*n*_DATA_REG This register is used to read and write data for channel *n* via APB FIFO. (RO)

Register 7.13: RMT_INT_RAW_REG (0x0050)



RMT_CH*n*_TX_END_INT_RAW The interrupt raw bit for channel *n*. Triggered when transmitting ends. (RO)

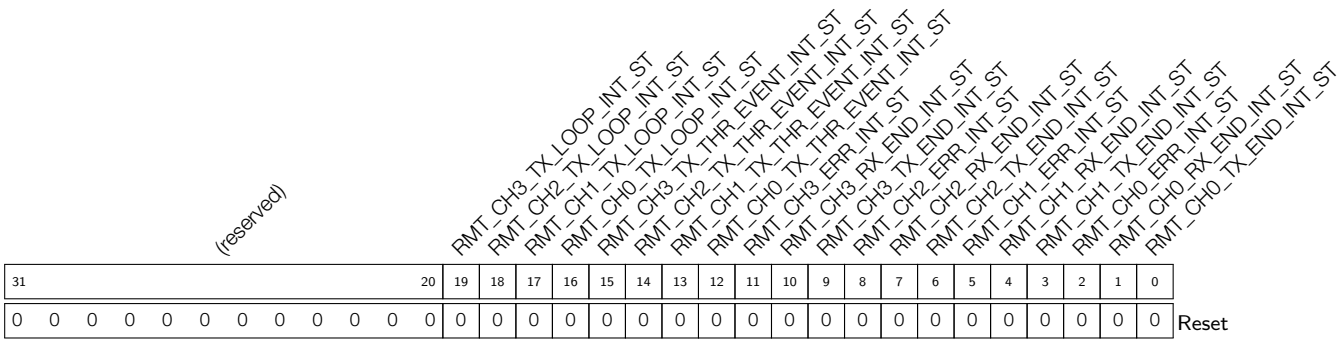
RMT_CH*n*_RX_END_INT_RAW The interrupt raw bit for channel *n*. Triggered when receiving ends. (RO)

RMT_CH*n*_ERR_INT_RAW The interrupt raw bit for channel *n*. Triggered when error occurs. (RO)

RMT_CH*n*_TX_THR_EVENT_INT_RAW The interrupt raw bit for channel *n*. Triggered when transmitter sends more data than configured value. (RO)

RMT_CH*n*_TX_LOOP_INT_RAW The interrupt raw bit for channel *n*. Triggered when loop counting reaches the configured threshold value. (RO)

Register 7.14: RMT_INT_ST_REG (0x0054)



RMT_CH*n*_TX_END_INT_ST The masked interrupt status bit for RMT_CH*n*_TX_END_INT. (RO)

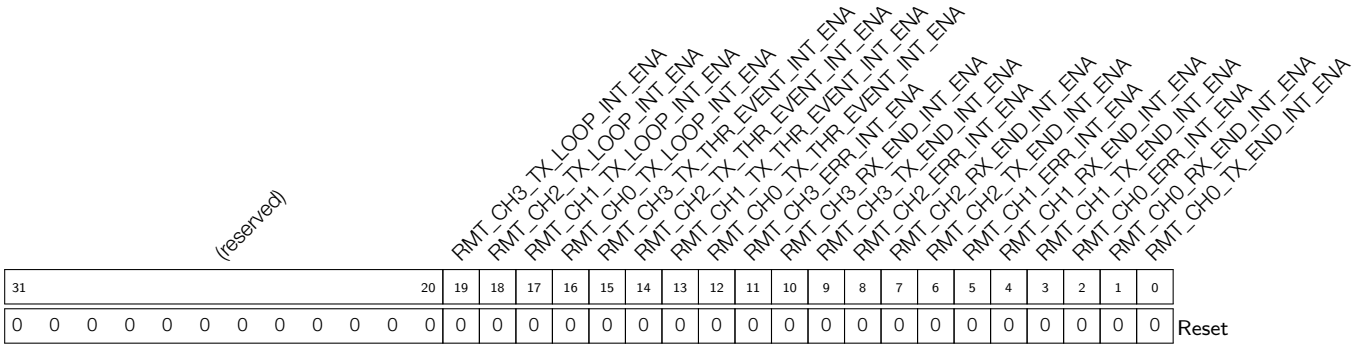
RMT_CH*n*_RX_END_INT_ST The masked interrupt status bit for RMT_CH*n*_RX_END_INT. (RO)

RMT_CH*n*_ERR_INT_ST The masked interrupt status bit for RMT_CH*n*_ERR_INT. (RO)

RMT_CH*n*_TX_THR_EVENT_INT_ST The masked interrupt status bit for RMT_CH*n*_TX_THR_EVENT_INT. (RO)

RMT_CH*n*_TX_LOOP_INT_ST The masked interrupt status bit for RMT_CH*n*_TX_LOOP_INT. (RO)

Register 7.15: RMT_INT_ENA_REG (0x0058)



RMT_CH*n*_TX_END_INT_ENA Interrupt enable bit for RMT_CH*n*_TX_END_INT. (R/W)

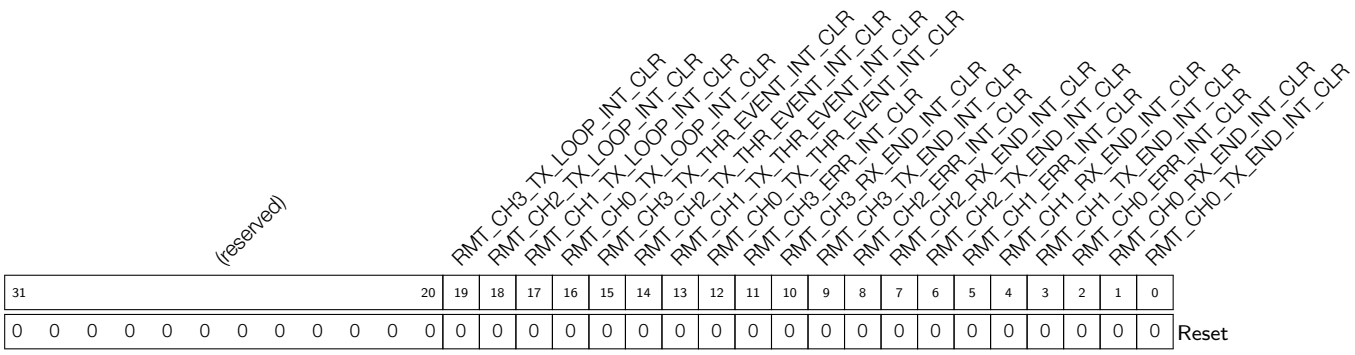
RMT_CH*n*_RX_END_INT_ENA Interrupt enable bit for RMT_CH*n*_RX_END_INT. (R/W)

RMT_CH*n*_ERR_INT_ENA Interrupt enable bit for RMT_CH*n*_ERR_INT. (R/W)

RMT_CH*n*_TX_THR_EVENT_INT_ENA Interrupt enable bit for RMT_CH*n*_TX_THR_EVENT_INT. (R/W)

RMT_CH*n*_TX_LOOP_INT_ENA Interrupt enable bit for RMT_CH*n*_TX_LOOP_INT. (R/W)

Register 7.16: RMT_INT_CLR_REG (0x005C)



RMT_CH*n*_TX_END_INT_CLR Set this bit to clear RMT_CH*n*_TX_END_INT interrupt. (WO)

RMT_CH*n*_RX_END_INT_CLR Set this bit to clear RMT_CH*n*_RX_END_INT interrupt. (WO)

RMT_CH*n*_ERR_INT_CLR Set this bit to clear RMT_CH*n*_ERR_INT interrupt. (WO)

RMT_CH*n*_TX_THR_EVENT_INT_CLR Set this bit to clear RMT_CH*n*_TX_THR_EVENT_INT interrupt. (WO)

RMT_CH*n*_TX_LOOP_INT_CLR Set this bit to clear RMT_CH*n*_TX_LOOP_INT interrupt. (WO)

8. Pulse Count Controller

The pulse count controller (PCNT) is designed to count input pulses and generate interrupts. It can increment or decrement a pulse counter value by keeping track of rising (positive) or falling (negative) edges of the input pulse signal. The PCNT has four independent pulse counters, called units which have their groups of registers. In this chapter, *n* denotes the number of a unit from 0 ~ 3.

Each unit includes two channels (ch0 and ch1) which can independently increment or decrement its pulse counter value. The remainder of the chapter will mostly focus on channel 0 (ch0) as the functionality of the two channels is identical.

As shown in Figure 8-1, each channel has two input signals:

1. One control signal (e.g. ctrl_ch0_u*n*, the control signal for ch0 of unit n)
2. One input pulse signal (e.g. sig_ch0_u*n*, the input pulse signal for ch0 unit n)

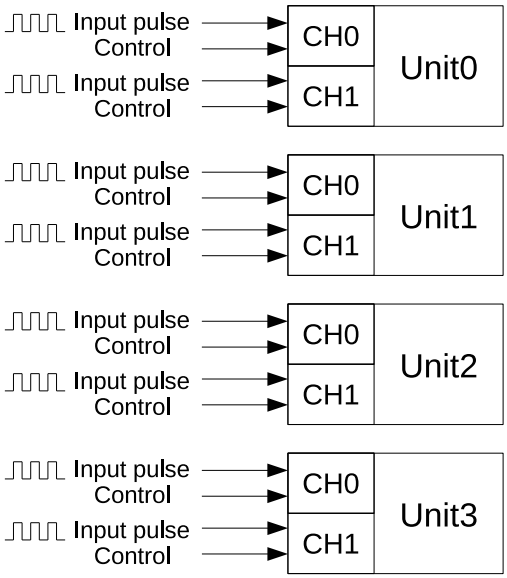


Figure 8-1. PCNT Block Diagram

8.1 Features

A PCNT has the following features:

- Four independent pulse counters (units)
- Each unit consists of two independent channels sharing one pulse counter
- All channels have input pulse signals (e.g. sig_ch0_u*n*) with their corresponding control signals (e.g. ctrl_ch0_u*n*)
- Independent filtering of input pulse signals (sig_ch0_u*n* and sig_ch1_u*n*) and control signals (ctrl_ch0_u*n* and ctrl_ch1_u*n*) on each unit
- Each channel has the following parameters:
    1. Selection between counting on positive or negative edges of the input pulse signal

2. Configuration to Increment, Decrement, or Disable counter mode for control signal's high and low states

8.2 Functional Description

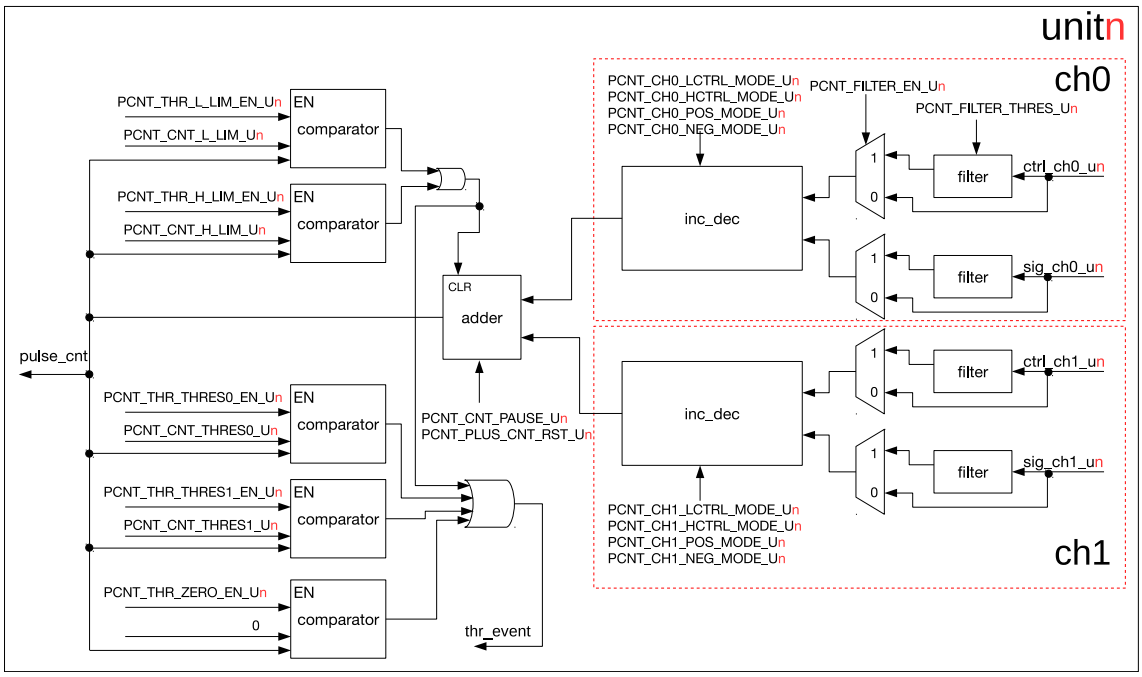


Figure 8-2. PCNT Unit Architecture

Figure 8-2 shows PCNT's architecture. As stated above, ctrl_ch0_u*n* is the control signal for ch0 of unit n. Its high and low states can be assigned different counter modes and used for pulse counting of the channel's input pulse signal sig_ch0_u*n* on negative or positive edges. The available counter modes are as follows:

- Increment mode: When a channel detects an active edge of sig_ch0_u*n* (the one configured for counting), the counter value pulse_cnt increases by 1. Upon reaching PCNT_CNT_H_LIM_U*n*, pulse_cnt is cleared. If the channel's counter mode is changed or if PCNT_CNT_PAUSE_U*n* is set before pulse_cnt reaches PCNT_CNT_H_LIM_U*n*, then pulse_cnt freezes and its counter mode changes.
- Decrement mode: When a channel detects an active edge of sig_ch0_u*n* (the one configured for counting), the counter value pulse_cnt decreases by 1. Upon reaching PCNT_CNT_L_LIM_U*n*, pulse_cnt is cleared. If the channel's counter mode is changed or if PCNT_CNT_PAUSE_U*n* is set before pulse_cnt reaches PCNT_CNT_H_LIM_U*n*, then pulse_cnt freezes and its counter mode changes.
- Disable mode: Counting is disabled, and the counter value pulse_cnt freezes.

Table 8-1 to Table 8-4 provide information on how to configure the counter mode for channel 0.

Each unit has one filter for all its control and input pulse signals. A filter can be enabled with the bit PCNT_FILTER_EN_U*n*. The filter monitors the signals and ignores all the noise, i.e. the glitches with pulse widths shorter than
PCNT_FILTER_THRES_U*n* APB clock cycles in length.

As previously mentioned, each unit has two channels which process different input pulse signals and increase or decrease values via their respective inc_dec modules, then the two channels send these values to the adder

Table 8-1. Counter Mode. Positive Edge of Input Pulse Signal. Control Signal in Low State

PCNT_CH0_POS_MODE_U*n*	PCNT_CH0_LCTRL_MODE_U*n*	Counter Mode
1	0	Increment
	1	Decrement
	Others	Disable
2	0	Decrement
	1	Increment
	Others	Disable
Others	N/A	Disable

Table 8-2. Counter Mode. Positive Edge of Input Pulse Signal. Control Signal in High State

PCNT_CH0_POS_MODE_U*n*	PCNT_CH0_HCTRL_MODE_U*n*	Counter Mode
1	0	Increment
	1	Decrement
	Others	Disable
2	0	Decrement
	1	Increment
	Others	Disable
Others	N/A	Disable

Table 8-3. Counter Mode. Negative Edge of Input Pulse Signal. Control Signal in Low State

PCNT_CH0_NEG_MODE_U*n*	PCNT_CH0_LCTRL_MODE_U*n*	Counter Mode
1	0	Increment
	1	Decrement
	Others	Disable
2	0	Decrement
	1	Increment
	Others	Disable
Others	N/A	Disable

Table 8-4. Counter Mode. Negative Edge of Input Pulse Signal. Control Signal in High State

PCNT_CH0_NEG_MODE_U*n*	PCNT_CH0_HCTRL_MODE_U*n*	Counter Mode
1	0	Increment
	1	Decrement
	Others	Disable
2	0	Decrement
	1	Increment
	Others	Disable
Others	N/A	Disable

module that is 16-bit wide with a sign bit. This adder can be suspended by setting PCNT_CNT_PAUSE_U*n*, and cleared by setting PCNT_PULSE_CNT_RST_U*n*.

The PCNT has five watchpoints that share one interrupt. The interrupt can be enabled or disabled by interrupt enable signals of each individual watchpoint.

- Maximum count value: When pulse_cnt reaches PCNT_CNT_H_LIM_U*n*, an interrupt is triggered and PCNT_CNT_THR_H_LIM_LAT_U*n* is high.
- Minimum count value: When pulse_cnt reaches PCNT_CNT_L_LIM_U*n*, an interrupt is triggered and PCNT_CNT_THR_L_LIM_LAT_U*n* is high.
- Two threshold values: When pulse_cnt equals either PCNT_CNT_THRES0_U*n* or PCNT_CNT_THRES1_U*n*, an interrupt is triggered and either PCNT_CNT_THR_THRES0_LAT_U*n* or PCNT_CNT_THR_THRES1_LAT_U*n* is high respectively.
- Zero: When pulse_cnt is 0, an interrupt is triggered and PCNT_CNT_THR_ZERO_LAT_U*n* is valid.

8.3 Applications

In each unit, channel 0 and channel 1 can be configured to work independently or together. The three subsections below provide details of channel 0 incrementing independently, channel 0 decrementing independently, and channel 0 and channel 1 incrementing together. For other working modes not elaborated in this section (e.g. channel 1 incrementing/decremeting independently, or one channel incrementing while the other decrementing), reference can be made to these three subsections.

8.3.1 Channel 0 Incrementing Independently

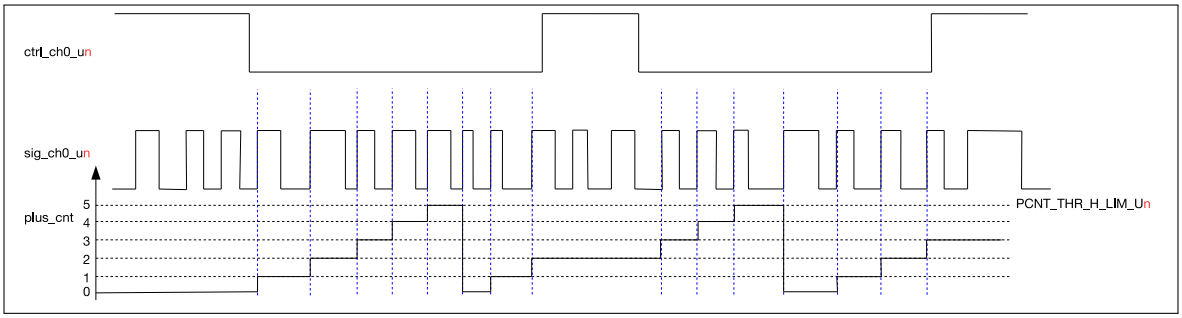


Figure 8-3. Channel 0 Up Counting Diagram

Figure 8-3 illustrates how channel 0 is configured to increment independently on the positive edge of sig_ch0_u*n* while channel 1 is disabled (see subsection 8.2 for how to disable channel 1). The configuration of channel 0 is shown below.

- PCNT_CH0_LCTRL_MODE_U*n*=0: When ctrl_ch0_u*n* is low, the counter mode specified for the low state turns on, in this case it is Increment mode.
- PCNT_CH0_HCTRL_MODE_U*n*=2: When ctrl_ch0_u*n* is high, the counter mode specified for the low state turns on, in this case it is Disable mode.
- PCNT_CH0_POS_MODE_U*n*=1: The counter increments on the positive edge of sig_ch0_u*n*.
- PCNT_CH0_NEG_MODE_U*n*=0: The counter idles on the negative edge of sig_ch0_u*n*.

- PCNT_CNT_H_LIM_U*n*=5: When pulse_cnt counts up to PCNT_CNT_H_LIM_U*n*, it is cleared.

8.3.2 Channel 0 Decrementing Independently

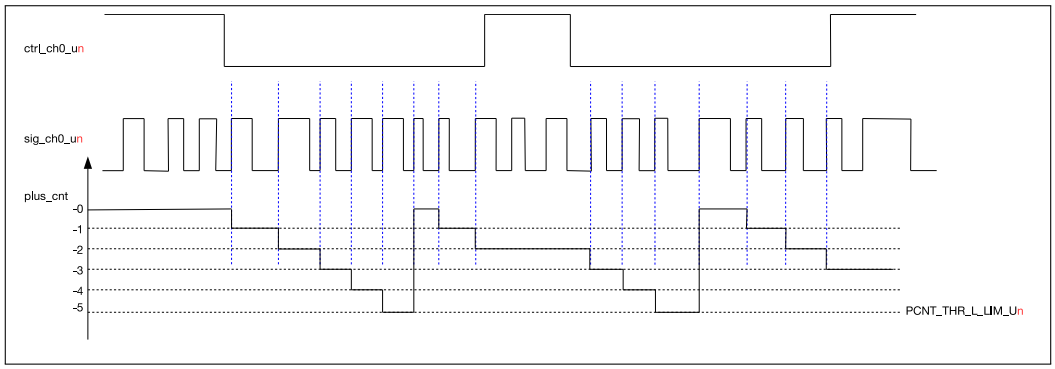


Figure 8-4. Channel 0 Down Counting Diagram

Figure 8-4 illustrates how channel 0 is configured to decrement independently on the positive edge of sig_ch0_u*n* while channel 1 is disabled. The configuration of channel 0 in this case differs from that in Figure 8-3 in the following aspects:

- PCNT_CH0_POS_MODE_U*n*=2: the counter decrements on the positive edge of sig_ch0_u*n*.
- PCNT_CNT_L_LIM_U*n*=-5: when pulse_cnt counts down to PCNT_CNT_L_LIM_U*n*, it is cleared.

8.3.3 Channel 0 and Channel 1 Incrementing Together

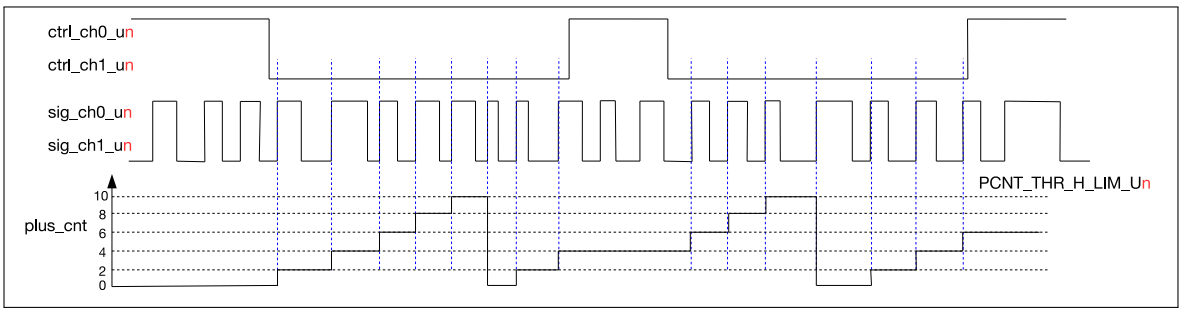


Figure 8-5. Two Channels Up Counting Diagram

Figure 8-5 illustrates how channel 0 and channel 1 are configured to increment on the positive edge of sig_ch0_u*n* and sig_ch1_u*n* respectively at the same time. It can be seen in Figure 8-5 that control signal ctrl_ch0_u*n* and ctrl_ch1_u*n* have the same waveform, so as input pulse signal sig_ch0_u*n* and sig_ch1_u*n*. The configuration procedure is shown below.

- For channel 0:
  - PCNT_CH0_LCTRL_MODE_U*n*=0: When ctrl_ch0_u*n* is low, the counter mode specified for the low state turns on, in this case it is Increment mode.
  - PCNT_CH0_HCTRL_MODE_U*n*=2: When ctrl_ch0_u*n* is high, the counter mode specified for the low state turns on, in this case it is Disable mode.
  - PCNT_CH0_POS_MODE_U*n*=1: The counter increments on the positive edge of sig_ch0_u*n*.
  - PCNT_CH0_NEG_MODE_U*n*=0: The counter idles on the negative edge of sig_ch0_u*n*.

- For channel 1:
  - PCNT_CH1_LCTRL_MODE_U*n*=0: When ctrl_ch1_u*n* is low, the counter mode specified for the low state turns on, in this case it is Increment mode.
  - PCNT_CH1_HCTRL_MODE_U*n*=2: When ctrl_ch1_u*n* is high, the counter mode specified for the low state turns on, in this case it is Disable mode.
  - PCNT_CH1_POS_MODE_U*n*=1: The counter increments on the positive edge of sig_ch1_u*n*.
  - PCNT_CH1_NEG_MODE_U*n*=0: The counter idles on the negative edge of sig_ch1_u*n*.
- PCNT_CNT_H_LIM_U*n*=10: When pulse_cnt counts up to PCNT_CNT_H_LIM_U*n*, it is cleared.

8.4 Base Address

Users can access the PCNT registers with two base addresses, which can be seen in the following table. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 8-5. PCNT Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F417000
PeriBUS2	0x60017000

8.5 Register Summary

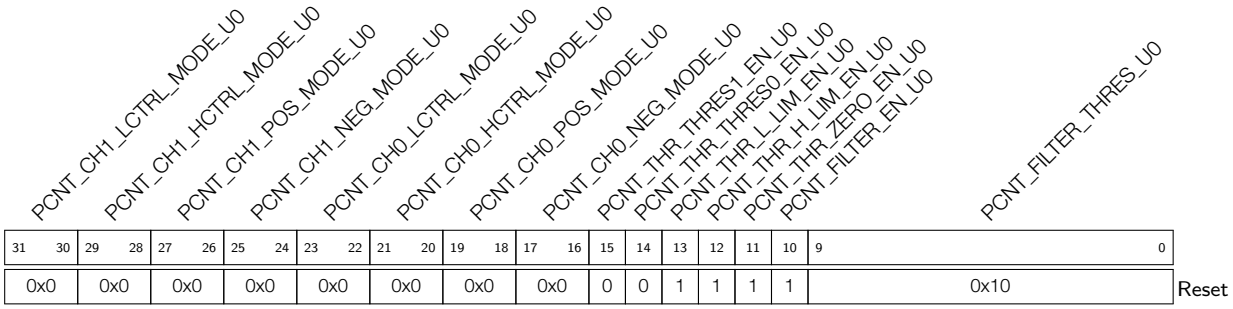
The addresses in the following table are relative to the PCNT base addresses provided in Section 8.4.

Name	Description	Address	Access
Configuration Register			
PCNT_U0_CONF0_REG	Configuration register 0 for unit 0	0x0000	R/W
PCNT_U0_CONF1_REG	Configuration register 1 for unit 0	0x0004	R/W
PCNT_U0_CONF2_REG	Configuration register 2 for unit 0	0x0008	R/W
PCNT_U1_CONF0_REG	Configuration register 0 for unit 1	0x000C	R/W
PCNT_U1_CONF1_REG	Configuration register 1 for unit 1	0x0010	R/W
PCNT_U1_CONF2_REG	Configuration register 2 for unit 1	0x0014	R/W
PCNT_U2_CONF0_REG	Configuration register 0 for unit 2	0x0018	R/W
PCNT_U2_CONF1_REG	Configuration register 1 for unit 2	0x001C	R/W
PCNT_U2_CONF2_REG	Configuration register 2 for unit 2	0x0020	R/W
PCNT_U3_CONF0_REG	Configuration register 0 for unit 3	0x0024	R/W
PCNT_U3_CONF1_REG	Configuration register 1 for unit 3	0x0028	R/W
PCNT_U3_CONF2_REG	Configuration register 2 for unit 3	0x002C	R/W
PCNT_CTRL_REG	Control register for all counters	0x0060	R/W
Status Register			
PCNT_U0_CNT_REG	Counter value for unit 0	0x0030	RO
PCNT_U1_CNT_REG	Counter value for unit 1	0x0034	RO

Name	Description	Address	Access
PCNT_U2_CNT_REG	Counter value for unit 2	0x0038	RO
PCNT_U3_CNT_REG	Counter value for unit 3	0x003C	RO
PCNT_U0_STATUS_REG	PNCT unit0 status register	0x0050	RO
PCNT_U1_STATUS_REG	PNCT unit1 status register	0x0054	RO
PCNT_U2_STATUS_REG	PNCT unit2 status register	0x0058	RO
PCNT_U3_STATUS_REG	PNCT unit3 status register	0x005C	RO
Interrupt Register			
PCNT_INT_RAW_REG	Interrupt raw status register	0x0040	RO
PCNT_INT_ST_REG	Interrupt status register	0x0044	RO
PCNT_INT_ENA_REG	Interrupt enable register	0x0048	R/W
PCNT_INT_CLR_REG	Interrupt clear register	0x004C	WO
Version Register			
PCNT_DATE_REG	PCNT version control register	0x00FC	R/W

8.6 Registers

Register 8.1: PCNT_U*n*_CONF0_REG (*n*: 0-3) (0x0000+12n*)**



PCNT_FILTER_THRES_U*n* This sets the maximum threshold, in APB_CLK cycles, for the filter. Any pulses with width less than this will be ignored when the filter is enabled. (R/W)

PCNT_FILTER_EN_U*n* This is the enable bit for unit *n*'s input filter. (R/W)

PCNT_THR_ZERO_EN_U*n* This is the enable bit for unit *n*'s zero comparator. (R/W)

PCNT_THR_H_LIM_EN_U*n* This is the enable bit for unit *n*'s thr_h_lim comparator. (R/W)

PCNT_THR_L_LIM_EN_U*n* This is the enable bit for unit *n*'s thr_l_lim comparator. (R/W)

PCNT_THR_THRES0_EN_U*n* This is the enable bit for unit *n*'s thres0 comparator. (R/W)

PCNT_THR_THRES1_EN_U*n* This is the enable bit for unit *n*'s thres1 comparator. (R/W)

PCNT_CH0_NEG_MODE_U*n* This register sets the behavior when the signal input of channel 0 detects a negative edge. 1: Increase the counter; 2: Decrease the counter; 0, 3: No effect on counter (R/W)

PCNT_CH0_POS_MODE_U*n* This register sets the behavior when the signal input of channel 0 detects a positive edge. 1: Increase the counter; 2: Decrease the counter; 0, 3: No effect on counter (R/W)

PCNT_CH0_HCTRL_MODE_U*n* This register configures how the CH*n*_POS_MODE/CH*n*_NEG_MODE settings will be modified when the control signal is high. 0: No modification; 1: Invert behavior (increase -> decrease, decrease -> increase); 2, 3: Inhibit counter modification (R/W)

PCNT_CH0_LCTRL_MODE_U*n* This register configures how the CH*n*_POS_MODE/CH*n*_NEG_MODE settings will be modified when the control signal is low. 0: No modification; 1: Invert behavior (increase -> decrease, decrease -> increase); 2, 3: Inhibit counter modification (R/W)

PCNT_CH1_NEG_MODE_U*n* This register sets the behavior when the signal input of channel 1 detects a negative edge. 1: Increment the counter; 2: Decrement the counter; 0, 3: No effect on counter (R/W)

PCNT_CH1_POS_MODE_U*n* This register sets the behavior when the signal input of channel 1 detects a positive edge. 1: Increment the counter; 2: Decrement the counter; 0, 3: No effect on counter (R/W)

Continued on the next page...

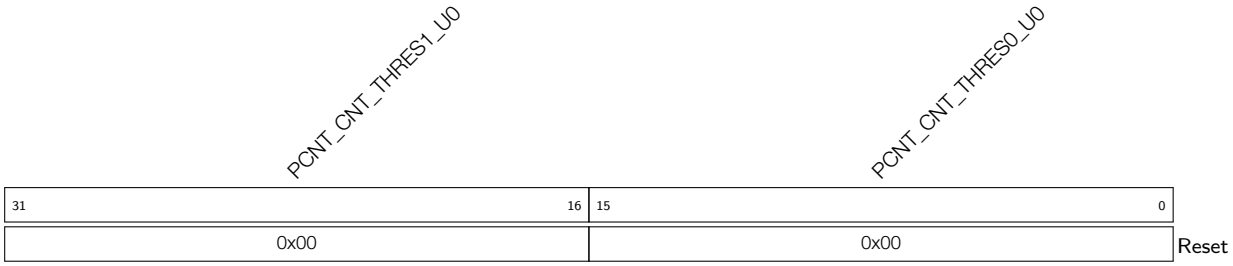
Register 8.1: PCNT_U*n*_CONF0_REG (*n*: 0-3) (0x0000+12n*)**

Continued from the previous page ...

PCNT_CH1_HCTRL_MODE_U*n* This register configures how the CH*n*_POS_MODE/CH*n*_NEG_MODE settings will be modified when the control signal is high. 0: No modification; 1: Invert behavior (increase -> decrease, decrease -> increase); 2, 3: Inhibit counter modification (R/W)

PCNT_CH1_LCTRL_MODE_U*n* This register configures how the CH*n*_POS_MODE/CH*n*_NEG_MODE settings will be modified when the control signal is low. 0: No modification; 1: Invert behavior (increase -> decrease, decrease -> increase); 2, 3: Inhibit counter modification (R/W)

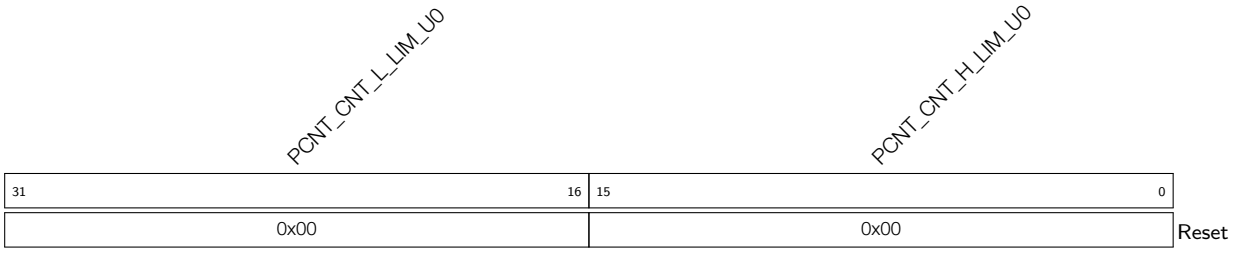
Register 8.2: PCNT_U*n*_CONF1_REG (*n*: 0-3) (0x0004+12n*)**



PCNT_CNT_THRES0_U*n* This register is used to configure the thres0 value for unit *n*. (R/W)

PCNT_CNT_THRES1_U*n* This register is used to configure the thres1 value for unit *n*. (R/W)

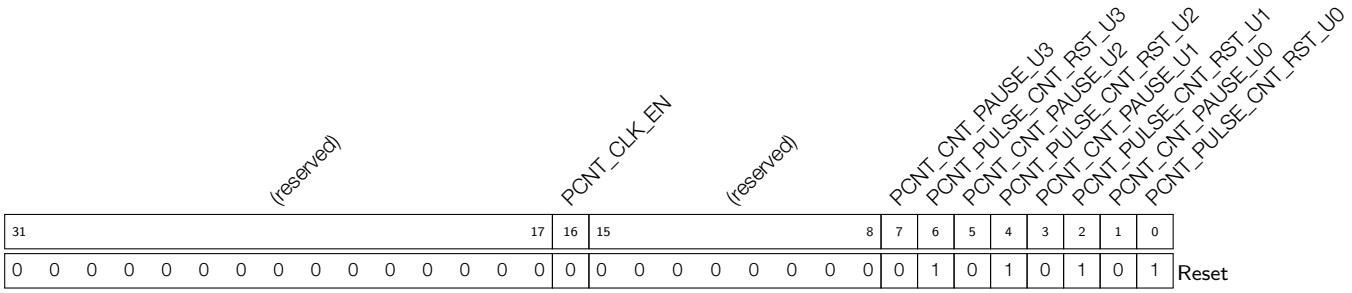
Register 8.3: PCNT_U*n*_CONF2_REG (*n*: 0-3) (0x0008+12n*)**



PCNT_CNT_H_LIM_U*n* This register is used to configure the thr_h_lim value for unit *n*. (R/W)

PCNT_CNT_L_LIM_U*n* This register is used to configure the thr_l_lim value for unit *n*. (R/W)

Register 8.4: PCNT_CTRL_REG (0x0060)

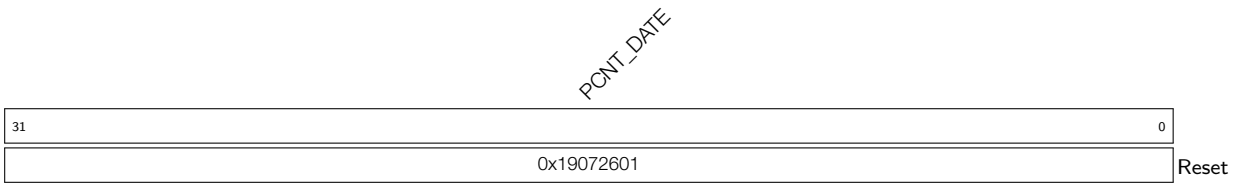


PCNT_PULSE_CNT_RST_U*n* Set this bit to clear unit *n*'s counter. (R/W)

PCNT_CNT_PAUSE_U*n* Set this bit to freeze unit *n*'s counter. (R/W)

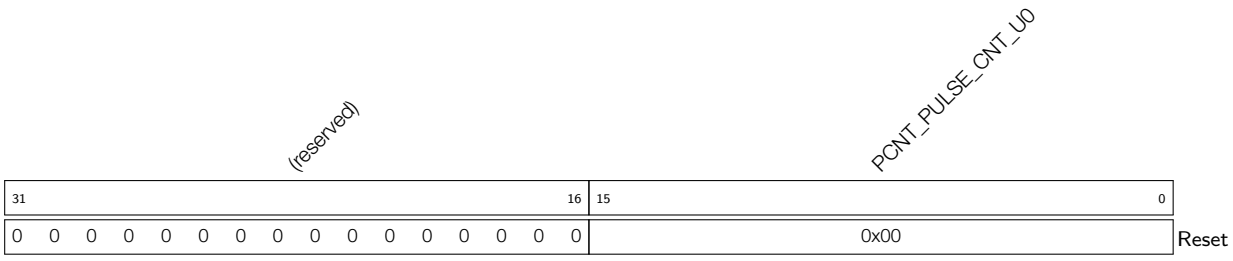
PCNT_CLK_EN The registers clock gate enable signal of PCNT module. 1: the registers can be read and written by application. 0: the registers can not be read or written by application (R/W)

Register 8.5: PCNT_DATE_REG (0x00FC)



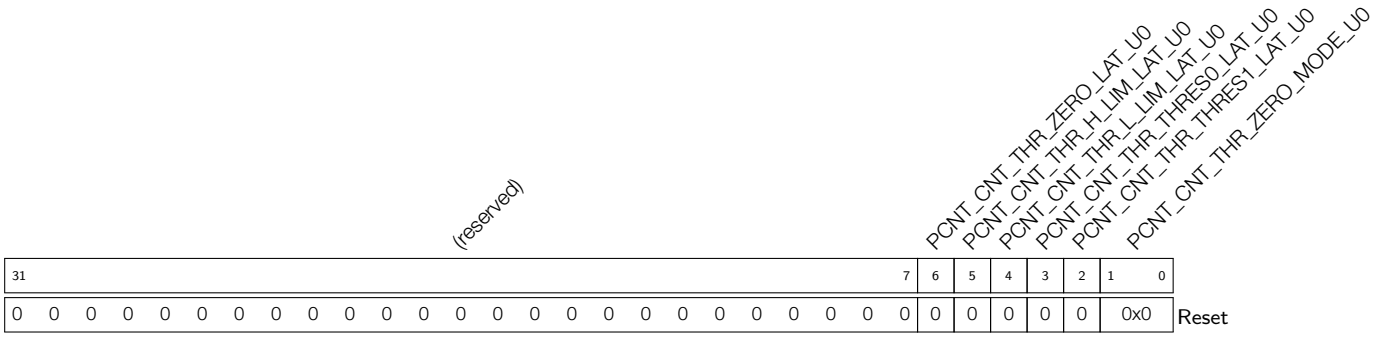
PCNT_DATE This is the PCNT version control register. (R/W)

Register 8.6: PCNT_U*n*_CNT_REG (*n*: 0-3) (0x0030+4n*)**



PCNT_PULSE_CNT_U*n* This register stores the current pulse count value for unit *n*. (RO)

Register 8.7: PCNT_U*n*_STATUS_REG (*n*: 0-3) (0x0050+4*n)



PCNT_CNT_THR_ZERO_MODE_U*n* The pulse counter status of PCNT_U*n* corresponding to 0. 0: pulse counter decreases from positive to 0. 1: pulse counter increases from negative to 0. 2: pulse counter is negative. 3: pulse counter is positive. (RO)

PCNT_CNT_THR_THRES1_LAT_U*n* The latched value of thres1 event of PCNT_U*n* when threshold event interrupt is valid. 1: the current pulse counter equals to thres1 and thres1 event is valid. 0: others (RO)

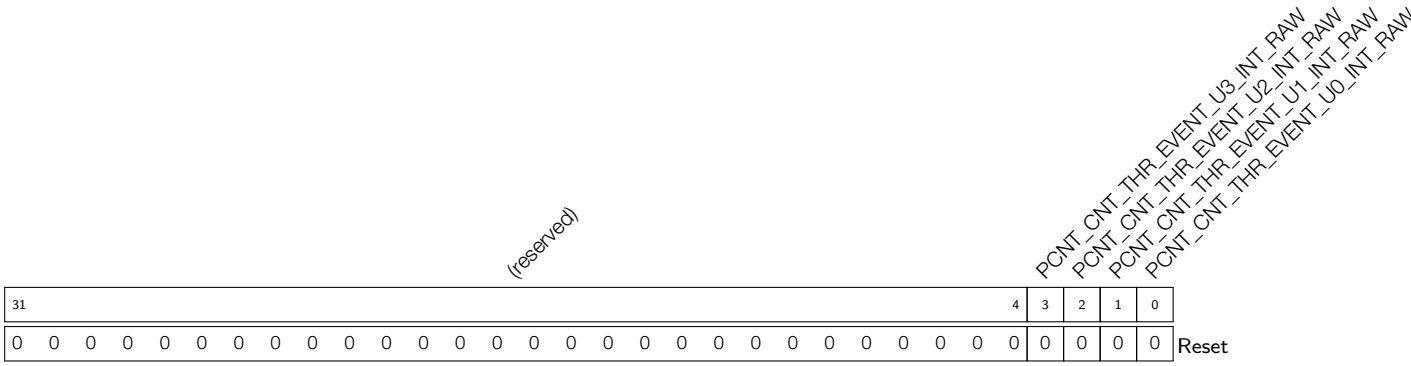
PCNT_CNT_THR_THRES0_LAT_U*n* The latched value of thres0 event of PCNT_U*n* when threshold event interrupt is valid. 1: the current pulse counter equals to thres0 and thres0 event is valid. 0: others (RO)

PCNT_CNT_THR_L_LIM_LAT_U*n* The latched value of low limit event of PCNT_U*n* when threshold event interrupt is valid. 1: the current pulse counter equals to thr_l_lim and low limit event is valid. 0: others (RO)

PCNT_CNT_THR_H_LIM_LAT_U*n* The latched value of high limit event of PCNT_U*n* when threshold event interrupt is valid. 1: the current pulse counter equals to thr_h_lim and high limit event is valid. 0: others (RO)

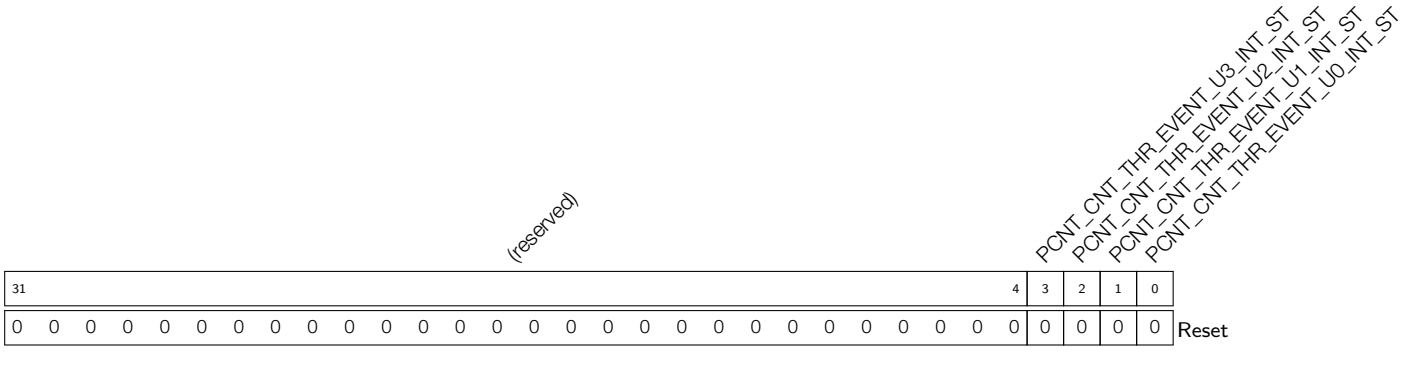
PCNT_CNT_THR_ZERO_LAT_U*n* The latched value of zero threshold event of PCNT_U*n* when threshold event interrupt is valid. 1: the current pulse counter equals to 0 and zero threshold event is valid. 0: others (RO)

Register 8.8: PCNT_INT_RAW_REG (0x0040)



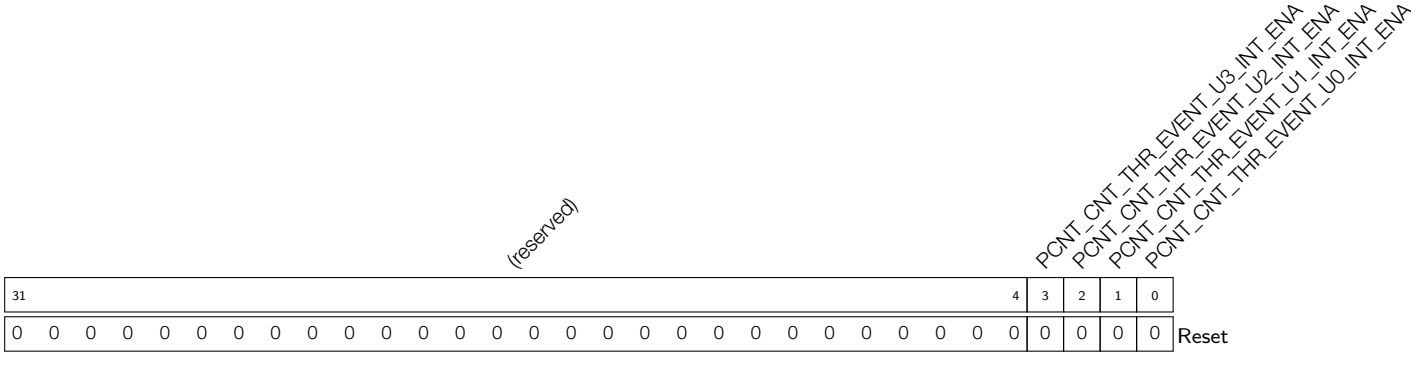
PCNT_CNT_THR_EVENT_U*n*_INT_RAW The raw interrupt status bit for the PCNT_CNT_THR_EVENT_U*n*_INT interrupt. (RO)

Register 8.9: PCNT_INT_ST_REG (0x0044)



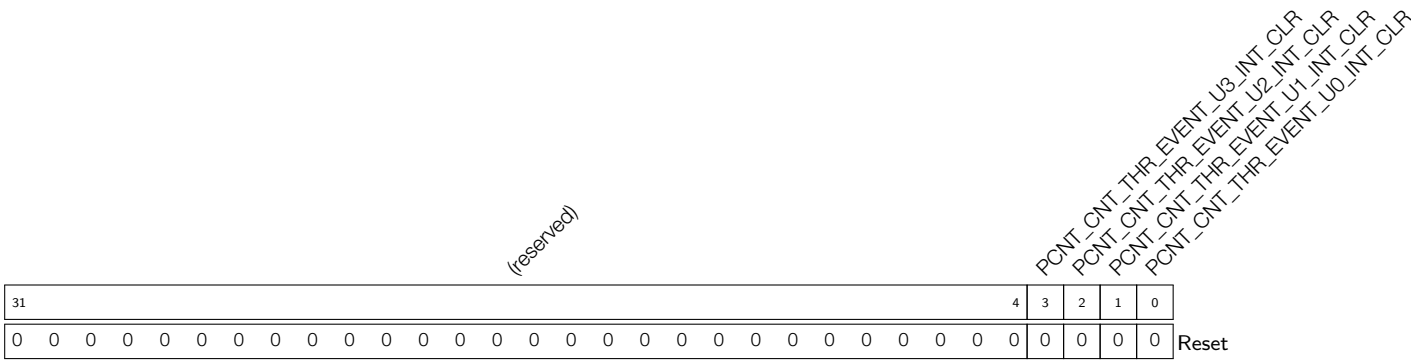
PCNT_CNT_THR_EVENT_U*n*_INT_ST The masked interrupt status bit for the PCNT_CNT_THR_EVENT_U*n*_INT interrupt. (RO)

Register 8.10: PCNT_INT_ENA_REG (0x0048)



PCNT_CNT_THR_EVENT_U*n*_INT_ENA The interrupt enable bit for the PCNT_CNT_THR_EVENT_U*n*_INT interrupt. (R/W)

Register 8.11: PCNT_INT_CLR_REG (0x004C)



PCNT_CNT_THR_EVENT_U*n*_INT_CLR Set this bit to clear the PCNT_CNT_THR_EVENT_U*n*_INT interrupt. (WO)

9. 64-bit Timers

9.1 Overview

General purpose timers can be used to precisely time an interval, trigger an interrupt after a particular interval (periodically and aperiodically), or act as a hardware clock. As shown in Figure 9-1, the ESP32-S2 chip contains two timer groups, namely timer group 0 and timer group 1. Each timer group consists of two general purpose timers referred to as T*x* (where *x* is 0 or 1) and one Main System Watchdog Timer. All general purpose timers are based on 16-bit prescalers and 64-bit auto-reload-capable up/down counters.

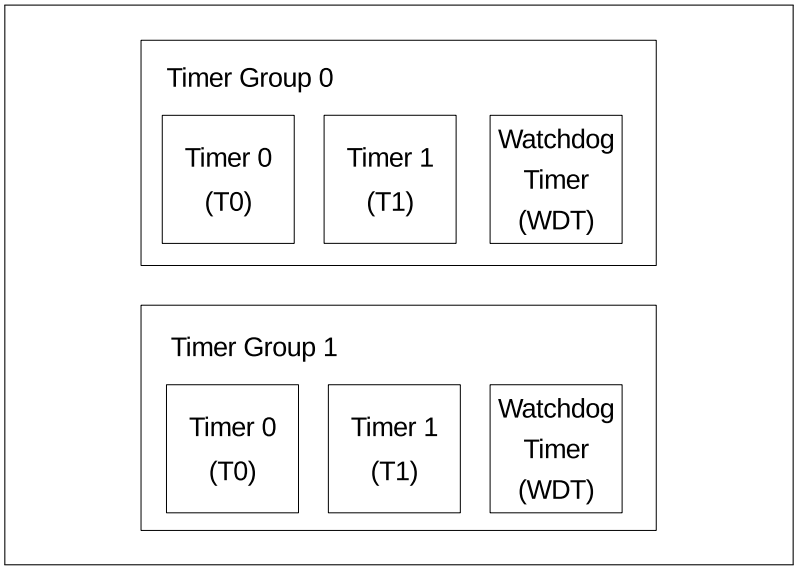


Figure 9-1. Timer Units within Groups

Note that while the Main System Watchdog Timer registers are described in this chapter, their functional description is included in the Chapter 10: *Watchdog Timers*. Therefore, the term 'timers' within this chapter refers to the general purpose timers.

The timers' features are summarized as follows:

- A 16-bit clock prescaler, from 1 to 65536
- A 64-bit time-base counter programmable to be incrementing or decrementing
- Able to read real-time value of the time-base counter
- Halting and resuming the time-base counter
- Programmable alarm generation
- Timer value reload (Auto-reload at alarm or software-controlled instant reload)
- Level and edge interrupt generation

9.2 Functional Description

9.2.1 16-bit Prescaler and Clock Selection

Each timer can select between the APB clock (APB_CLK) or external clock (XTAL_CLK) as its clock source by setting the TIMG_T*x*_USE_XTAL field of the TIMG_T*x*CONFIG_REG register. The clock is then divided by a 16-bit prescaler to generate the time-base counter clock (TB_CLK) used by the time-base counter. The 16-bit prescaler is configured by the TIMG_T*x*_DIVIDER field and can take any value from 1 to 65536. Note that programming a value of 0 in TIMG_T*x*_DIVIDER will result in the divisor being 65536.

The timer must be disabled (i.e. TIMG_T*x*_EN should be cleared) before modifying the 16-bit prescaler. Modifying the 16-bit prescaler whilst the timer is enabled can lead to unpredictable results.

9.2.2 64-bit Time-based Counter

The 64-bit time-base counters are based on TB_CLK and can the configured to increment or decrement via the TIMG_T*x*_INCREASE field. The time-base counter can be enabled/disabled by setting/clearing the TIMG_T*x*_EN field. Whilst enabled, the time-base counter will increment/decrement on each cycle of TB_CLK. When disabled, the time-base counter is essentially frozen. Note that the TIMG_T*x*_INCREASE field can be changed whilst TIMG_T*x*_EN is set and will cause the time-base counter to change direction instantly.

To read the 64-bit current timer value of the time-base counter, the timer value must be latched to two registers before being read by the CPU (due to the CPU being 32-bit). By writing any value to the TIMG_T*x*UPDATE_REG, the current value of the 64-bit timer is instantly latched into the TIMG_T*x*LO_REG and TIMG_T*x*HI_REG registers containing the lower and upper 32-bits respectively. TIMG_T*x*LO_REG and TIMG_T*x*HI_REG registers will remain unchanged for the CPU to read in its own time until TIMG_T*x*UPDATE_REG is written to again.

9.2.3 Alarm Generation

A timer can be configured to trigger an alarm when the timer's current value matches the alarm value. An alarm will cause an interrupt to occur and (optionally) an automatic reload of the timer's current value (see Section 9.2.4). The 64-bit alarm value is configured in the TIMG_T*x*ALARMLO_REG and TIMG_T*x*ALARMHI_REG representing the lower and upper 32-bits of the alarm value respectively. However, the configured alarm value is ineffective until the alarm is enabled by setting the TIMG_T*x*_ALARM_EN field. In order to simply the scenario where the alarm is enabled 'too late' (i.e. the timer value has already passed the alarm value when the alarm is enabled), the alarm value will also trigger immediately if the current timer value is larger/smaller than the alarm value for an up-counting/down-counting timer.

When an alarm occurs, the TIMG_T*x*_ALARM_EN field is automatically cleared and no alarm will occur again until the TIMG_T*x*_ALARM_EN is set.

9.2.4 Timer Reload

A timer is reloaded when a timer's current value is overwritten with a reload value stored in the TIMG_T*x*_LOAD_LO and TIMG_T*x*_LOAD_HI registers that correspond to the lower and upper 32-bits of the timer's new value respectively. However, writing a reload value to TIMG_T*x*_LOAD_LO and TIMG_T*x*_LOAD_HI will not cause the timer's current value to change. Instead, the reload value is ignored by the timer until a reload event occurs. A reload event can be triggered either by a software instant reload or an auto-reload at alarm.

A software instant reload is triggered by the CPU writing any value to TIMG_TxLOAD_REG causing the timer's current value to be instantly reloaded. If TIMG_Tx_EN is set, the timer will continue incrementing/decrementing from the new value. If TIMG_Tx_EN is cleared, the timer will remain frozen at the new value until counting is re-enabled.

An auto-reload at alarm will cause a timer reload when an alarm occurs thus allowing the timer to continue incrementing/decrementing from the reload value. This is generally useful for resetting the timer's value when using periodic alarms. To enable auto-reload at alarm, the TIMG_Tx_AUTORELOAD field should be set. If not enabled, the timer's value will continue to increment/decrement past the alarm value after an alarm.

9.2.5 Interrupts

Each timer has its own pair of interrupt lines (for edge and level interrupts) that can be routed to the CPU. Thus, there are a total of six interrupt lines per timer group and they are named as follows:

- TIMG_WDT_LEVEL_INT: Level interrupt line for the watchdog timer in the group, generated when a watchdog timer interrupt stage times out.
- TIMG_WDT_EDGE_INT: Edge interrupt line for the watchdog timer in the group, generated when a watchdog timer interrupt stage times out.
- TIMG_Tx_LEVEL_INT: Level interrupt for one of the general purpose timers, generated when an alarm event happens.
- TIMG_Tx_EDGE_INT: Edge interrupt for one of the general purpose timer, generated when an alarm event happens.

Interrupts are triggered after an alarm (or stage timeout for watchdog timers) occurs. Level interrupt lines will be held high after an alarm (or stage timeout) occurs, and will remain so until manually cleared. Conversely, edge interrupts will generate a short pulse after an alarm (or stage timeout) occurs. To enable a timer's level or edge interrupt lines, the TIMG_Tx_LEVEL_INT_EN or TIMG_Tx_EDGE_INT_EN bits should be set respectively.

The interrupts of each timer group are governed by a set of registers. Each timer within the group will have a corresponding bit in each of these registers:

- TIMG_Tx_INT_RAW : An alarm event sets it to 1. The bit will remain set until writing to the timer's corresponding bit in TIMG_Tx_INT_CLR.
- TIMG_WDT_INT_RAW : A stage time out will set the timer's bit to 1. The bit will remain set until writing to the timer's corresponding bit in TIMG_WDT_INT_CLR.
- TIMG_Tx_INT_ST : Reflects the status of each timer's interrupt and is generated by masking the bits of TIMG_Tx_INT_RAW with TIMG_Tx_INT_ENA. For level interrupts, these bits reflect the level on the watchdog timer's level interrupt line.
- TIMG_WDT_INT_ST : Reflects the status of each watchdog timer's interrupt and is generated by masking the bits of TIMG_WDT_INT_RAW with TIMG_WDT_INT_ENA. For level interrupts, these bits reflect the level on the watchdog timer's level interrupt line.
- TIMG_Tx_INT_ENA : Used to enable or mask the interrupt status bits of timers within the group.
- TIMG_WDT_INT_ENA : Used to enable or mask the interrupt status bits of watchdog timer within the group.
- TIMG_Tx_INT_CLR : Used to clear a timer's interrupt by setting its corresponding bit to 1. The timer's corresponding bit in TIMG_Tx_INT_RAW and TIMG_Tx_INT_ST will be cleared as a result. Note that a

timer's interrupt must be cleared before the next interrupt occurs when using level interrupts.

- TIMG_WDT_INT_CLR : Used to clear a timer's interrupt by setting its corresponding bit to 1. The watchdog timer's corresponding bit in TIMG_WDT_INT_RAW and TIMG_WDT_INT_ST will be cleared as a result. Note that a wachdog timer's interrupt must be cleared before the next interrupt occurs when using level interrupts.

9.3 Configuration and Usage

9.3.1 Timer as a Simple Clock

1. Configure the time-base counter
   - Select clock source by setting TIMG_T*x*_USE_XTAL field.
   - Configure the 16-bit prescaler by setting TIMG_T*x*_DIVIDER.
   - Configure the timer direction by setting/clearing TIMG_T*x*_INCREASE.
   - Set the timer's starting value by writing the starting value to TIMG_T*x*_LOAD_LO and TIMG_T*x*_LOAD_HI, then reloading it into the timer by writing any value to TIMG_T*x*LOAD_REG.
2. Start the timer by setting TIMG_T*x*_EN.
3. Getting the timer's current value.
   - Write any value to TIMG_T*x*UPDATE_REG to latch the timer's current value.
   - Read the latched timer value from TIMG_T*x*LO_REG and TIMG_T*x*HI_REG.

9.3.2 Timer as One-shot Alarm

1. Configure the time-base counter following step 1 of Section 9.3.1.
2. Configure the alarm.
   - Configure the alarm value by setting TIMG_T*x*ALARMLO_REG and TIMG_T*x*ALARMHI_REG.
   - Enable interrupt by setting TIMG_T*x*_LEVEL_INT_EN or TIMG_T*x*_EDGE_INT_EN for level or edge interrupts respectively.
3. Disable auto reload by clearing TIMG_T*x*_AUTORELOAD.
4. Start the timer by setting TIMG_T*x*_EN.
5. Handle the alarm interrupt.
   - Clear the interrupt by setting the timer's corresponding bit in TIMG_T*x*_INT_CLR.
   - Disable the timer by clearing TIMG_T*x*_EN.

9.3.3 Timer as Periodic Alarm

1. Configure the time-base counter following step 1 of Section 9.3.1.
2. Configure the alarm following step 2 of Section 9.3.2.
3. Enable auto reload by setting TIMG_T*x*_AUTORELOAD and setting the reload value in TIMG_T*x*_LOAD_LO and TIMG_T*x*_LOAD_HI.
4. Start the timer by setting TIMG_T*x*_EN.
5. Handle the alarm interrupt (repeat on each alarm iteration).
   - Clear the interrupt by setting the timer's corresponding bit in TIMG_T*x*_INT_CLR.
   - If the next alarm requires a new alarm value and reload value (i.e. different alarm interval per iteration), then TIMG_T*x*ALARMLO_REG, TIMG_T*x*ALARMHI_REG, TIMG_T*x*_LOAD_LO, and TIMG_T*x*_LOAD_HI should be reconfigured as needed. Otherwise, the aforementioned registers should remain unchanged.
   - Re-enable the alarm by setting TIMG_T*x*_ALARM_EN.
6. Stopping the timer (on final alarm iteration).
   - Clear the interrupt by setting the timer's corresponding bit in TIMG_T*x*_INT_CLR.
   - Disable the timer by clearing TIMG_T*x*_EN.

9.4 Base Address

Users can access the 64-bit Timer with four base addresses, which can be seen in the following table. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 9-1. 64-bit Timers Base Address

	Bus to Access Peripheral	Base Address
TIMG0	PeriBUS1	0x3F41F000
	PeriBUS2	0x6001F000
TIMG1	PeriBUS1	0x3F420000
	PeriBUS2	0x60020000

9.5 Register Summary

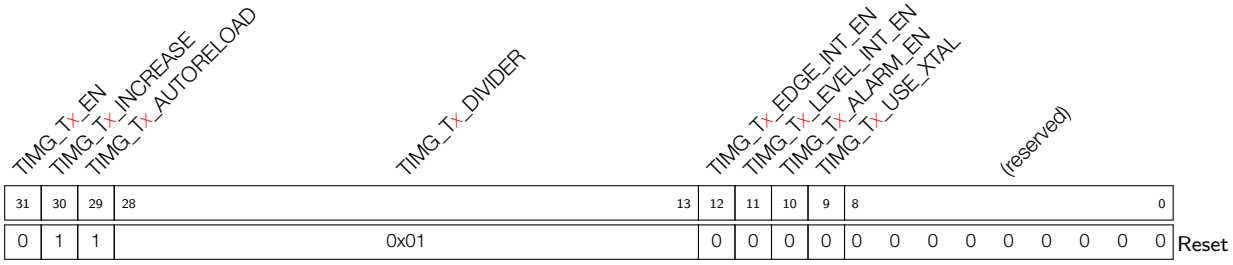
The addresses in the following table are relative to the 64-bit Timer base addresses provided in Section 9.4.

Name	Description	Address	Access
Configuration and Control Register for Timer 0			
TIMG_T0CONFIG_REG	Timer 0 configuration register	0x0000	R/W
TIMG_T0LO_REG	Timer 0 current value, low 32 bits	0x0004	RO
TIMG_T0HI_REG	Timer 0 current value, high 32 bits	0x0008	RO

Name	Description	Address	Access
TIMG_T0UPDATE_REG	Write to copy current timer value to TIMG_T0LO_REG and TIMG_T0LO_REG	0x000C	R/W
TIMG_T0ALARMLO_REG	Timer 0 alarm value, low 32 bits	0x0010	R/W
TIMG_T0ALARMHI_REG	Timer 0 alarm value, high bits	0x0014	R/W
TIMG_T0LOADLO_REG	Timer 0 reload value, low 32 bits	0x0018	R/W
TIMG_T0LOADHI_REG	Timer 0 reload value, high 32 bits	0x001C	R/W
TIMG_T0LOAD_REG	Write to reload timer from TIMG_T0LOADLO_REG and TIMG_T0LOADHI_REG	0x0020	WO
Configuration and Control Register for Timer 1			
TIMG_T1CONFIG_REG	Timer 1 configuration register	0x0024	R/W
TIMG_T1LO_REG	Timer 1 current value, low 32 bits	0x0028	RO
TIMG_T1HI_REG	Timer 1 current value, high 32 bits	0x002C	RO
TIMG_T1UPDATE_REG	Write to copy current timer value to TIMG_T1LO_REG and TIMG_T1LO_REG	0x0030	R/W
TIMG_T1ALARMLO_REG	Timer 1 alarm value, low 32 bits	0x0034	R/W
TIMG_T1ALARMHI_REG	Timer 1 alarm value, high bits	0x0038	R/W
TIMG_T1LOADLO_REG	Timer 1 reload value, low 32 bits	0x003C	R/W
TIMG_T1LOADHI_REG	Timer 1 reload value, high 32 bits	0x0040	R/W
TIMG_T1LOAD_REG	Write to reload timer from TIMG_T0LOADLO_REG and TIMG_T0LOADHI_REG	0x0044	WO
Configuration and Control Register for WDT			
TIMG_WDTCONFIG0_REG	Watchdog timer configuration register	0x0048	R/W
TIMG_WDTCONFIG1_REG	Watchdog timer prescaler register	0x004C	R/W
TIMG_WDTCONFIG2_REG	Watchdog timer stage 0 timeout value	0x0050	R/W
TIMG_WDTCONFIG3_REG	Watchdog timer stage 1 timeout value	0x0054	R/W
TIMG_WDTCONFIG4_REG	Watchdog timer stage 2 timeout value	0x0058	R/W
TIMG_WDTCONFIG5_REG	Watchdog timer stage 3 timeout value	0x005C	R/W
TIMG_WDTFEED_REG	Write to feed the watchdog timer	0x0060	WO
TIMG_WDTWPROTECT_REG	Watchdog write protect register	0x0064	R/W
Configuration and Control Register for RTC CALI			
TIMG_RTCCALICFG2_REG	Timer group calibration register	0x00A8	varies
Interrupt Register			
TIMG_INT_ENA_TIMERS_REG	Interrupt enable bits	0x0098	R/W
TIMG_INT_RAW_TIMERS_REG	Raw interrupt status	0x009C	RO
TIMG_INT_ST_TIMERS_REG	Masked interrupt status	0x00A0	RO
TIMG_INT_CLR_TIMERS_REG	Interrupt clear bits	0x00A4	WO
Version Register			
TIMG_TIMERS_DATE_REG	Version control register	0x00F8	R/W
Configuration Register			
TIMG_REGCLK_REG	Timer group clock gate register	0x00FC	R/W

9.6 Registers

Register 9.1: TIMG_TxCONFIG_REG (x: 0-1) (0x0000+36*x)



TIMG_Tx_USE_XTAL 1: Use XTAL_CLK as the source clock of timer group. 0: Use APB_CLK as the source clock of timer group. (R/W)

TIMG_Tx_ALARM_EN When set, the alarm is enabled. This bit is automatically cleared once an alarm occurs. (R/W)

TIMG_Tx_LEVEL_INT_EN When set, an alarm will generate a level type interrupt. (R/W)

TIMG_Tx_EDGE_INT_EN When set, an alarm will generate an edge type interrupt. (R/W)

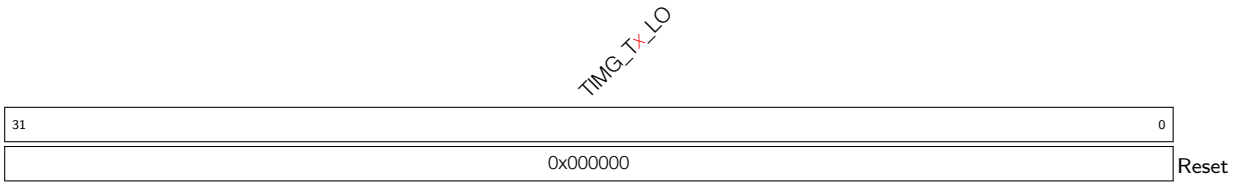
TIMG_Tx_DIVIDER Timer x clock (Tx_clk) prescaler value. (R/W)

TIMG_Tx_AUTORELOAD When set, timer x auto-reload at alarm is enabled. (R/W)

TIMG_Tx_INCREASE When set, the timer x time-base counter will increment every clock tick. When cleared, the timer x time-base counter will decrement. (R/W)

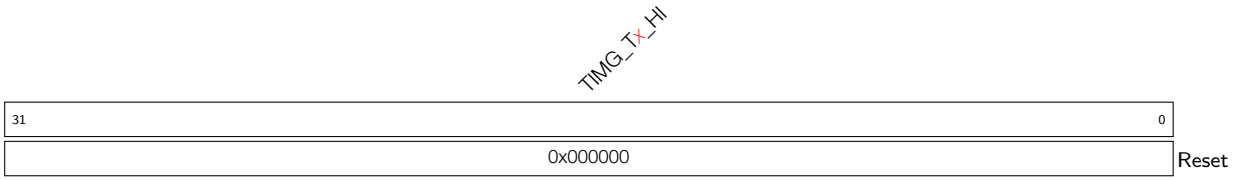
TIMG_Tx_EN When set, the timer x time-base counter is enabled. (R/W)

Register 9.2: TIMG_TxLO_REG (x: 0-1) (0x0004+36*x)



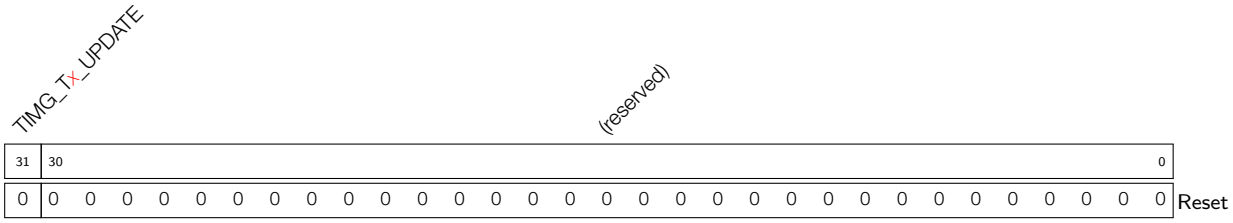
TIMG_Tx_LO After writing to TIMG_TxUPDATE_REG, the low 32 bits of the time-base counter of timer x can be read here. (RO)

Register 9.3: TIMG_TxHI_REG (x: 0-1) (0x0008+36*x)



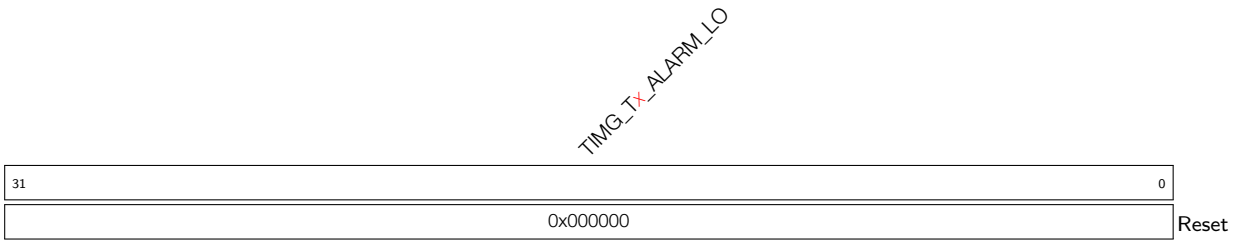
TIMG_Tx_HI After writing to TIMG_TxUPDATE_REG, the high 32 bits of the time-base counter of timer *x* can be read here. (RO)

Register 9.4: TIMG_TxUPDATE_REG (x: 0-1) (0x000C+36*x)



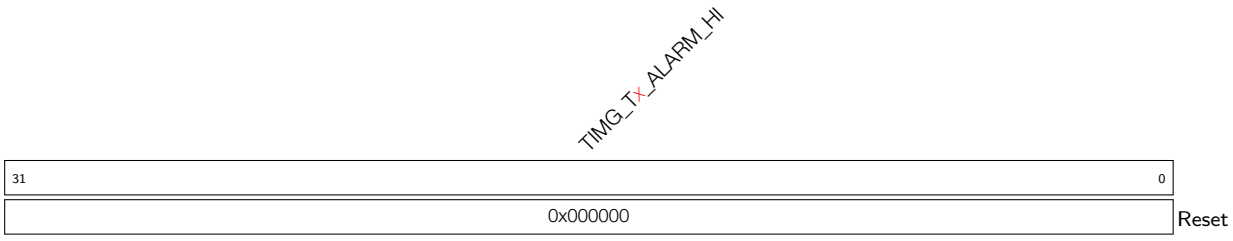
TIMG_Tx_UPDATE After writing 0 or 1 to TIMG_TxUPDATE_REG, the counter value is latched. (R/W)

Register 9.5: TIMG_TxALARMLO_REG (x: 0-1) (0x0010+36*x)



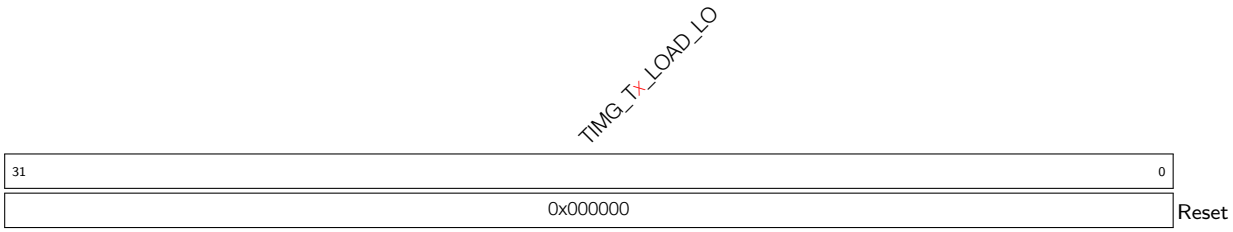
TIMG_Tx_ALARM_LO Timer *x* alarm trigger time-base counter value, low 32 bits. (R/W)

Register 9.6: TIMG_TxALARMHI_REG (x: 0-1) (0x0014+36*x)



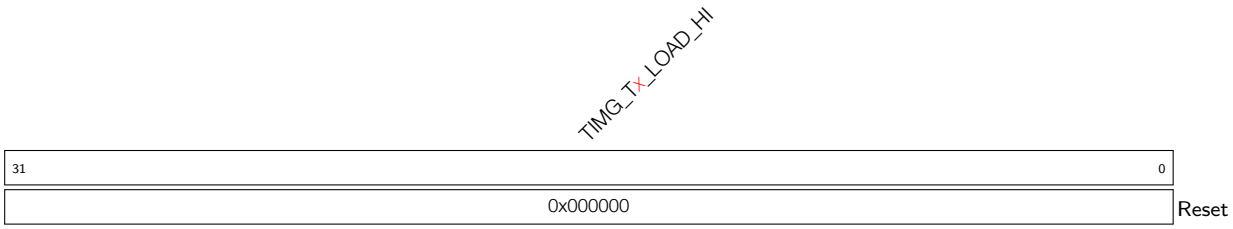
TIMG_Tx_ALARM_HI Timer *x* alarm trigger time-base counter value, high 32 bits. (R/W)

Register 9.7: TIMG_TxLOADLO_REG (x: 0-1) (0x0018+36*x)



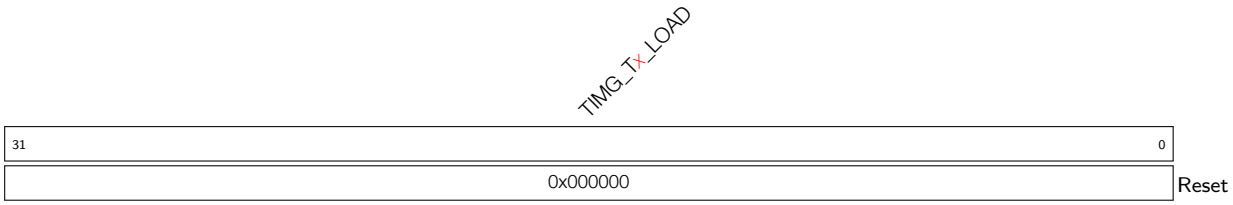
TIMG_Tx_LOAD_LO Low 32 bits of the value that a reload will load onto timer *x* time-base Counter. (R/W)

Register 9.8: TIMG_TxLOADHI_REG (x: 0-1) (0x001C+36*x)



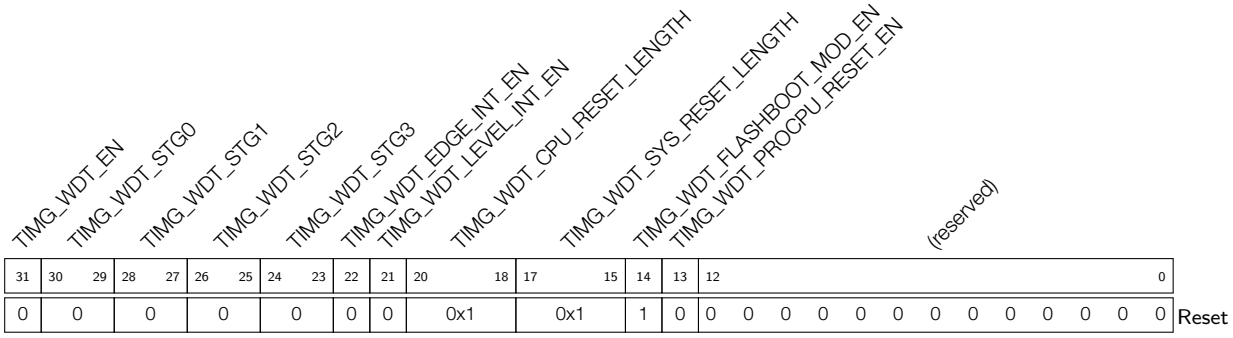
TIMG_Tx_LOAD_HI High 32 bits of the value that a reload will load onto timer *x* time-base counter. (R/W)

Register 9.9: TIMG_TxLOAD_REG (x: 0-1) (0x0020+36*x)



TIMG_Tx_LOAD Write any value to trigger a timer *x* time-base counter reload. (WO)

Register 9.10: TIMG_WDTCONFIG0_REG (0x0048)



TIMG_WDT_PROCPU_RESET_EN WDT reset CPU enable. (R/W)

TIMG_WDT_FLASHBOOT_MOD_EN When set, Flash boot protection is enabled. (R/W)

TIMG_WDT_SYS_RESET_LENGTH System reset signal length selection. 0: 100 ns, 1: 200 ns, 2: 300 ns, 3: 400 ns, 4: 500 ns, 5: 800 ns, 6: 1.6 us, 7: 3.2 us. (R/W)

TIMG_WDT_CPU_RESET_LENGTH CPU reset signal length selection. 0: 100 ns, 1: 200 ns, 2: 300 ns, 3: 400 ns, 4: 500 ns, 5: 800 ns, 6: 1.6 us, 7: 3.2 us. (R/W)

TIMG_WDT_LEVEL_INT_EN When set, a level type interrupt will occur at the timeout of a stage configured to generate an interrupt. (R/W)

TIMG_WDT_EDGE_INT_EN When set, an edge type interrupt will occur at the timeout of a stage configured to generate an interrupt. (R/W)

TIMG_WDT_STG3 Stage 3 configuration. 0: off, 1: interrupt, 2: reset CPU, 3: reset system. (R/W)

TIMG_WDT_STG2 Stage 2 configuration. 0: off, 1: interrupt, 2: reset CPU, 3: reset system. (R/W)

TIMG_WDT_STG1 Stage 1 configuration. 0: off, 1: interrupt, 2: reset CPU, 3: reset system. (R/W)

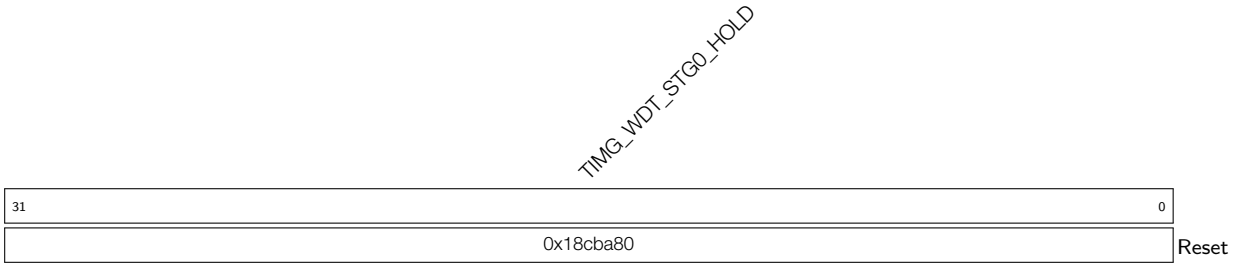
TIMG_WDT_STG0 Stage 0 configuration. 0: off, 1: interrupt, 2: reset CPU, 3: reset system. (R/W)

TIMG_WDT_EN When set, MWDT is enabled. (R/W)

Register 9.11: TIMG_WDTCONFIG1_REG (0x004C)

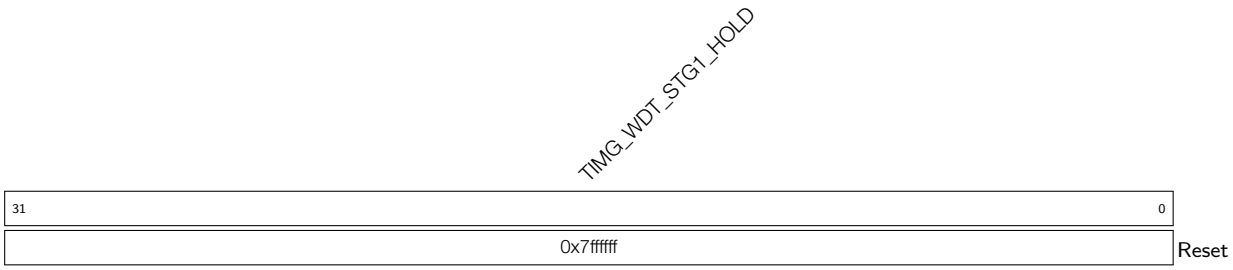
TIMG_WDT_CLK_PRESCALER MWDT clock prescaler value. MWDT clock period = 12.5 ns * TIMG_WDT_CLK_PRESCALE. (R/W)

Register 9.12: TIMG_WDTCONFIG2_REG (0x0050)



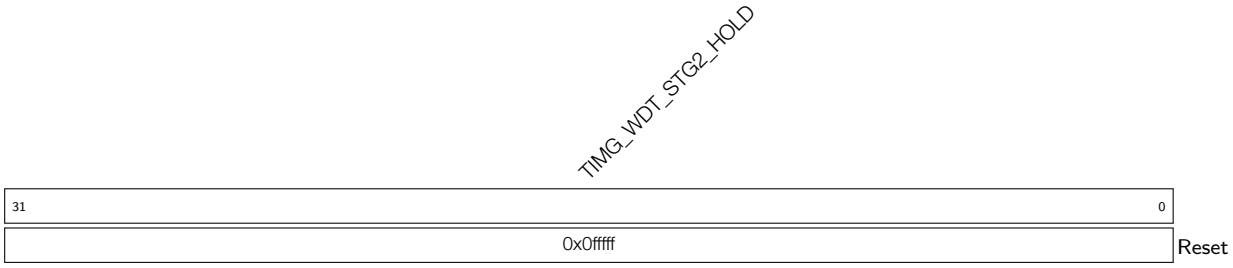
TIMG_WDT_STG0_HOLD Stage 0 timeout value, in MWDT clock cycles. (R/W)

Register 9.13: TIMG_WDTCONFIG3_REG (0x0054)



TIMG_WDT_STG1_HOLD Stage 1 timeout value, in MWDT clock cycles. (R/W)

Register 9.14: TIMG_WDTCONFIG4_REG (0x0058)

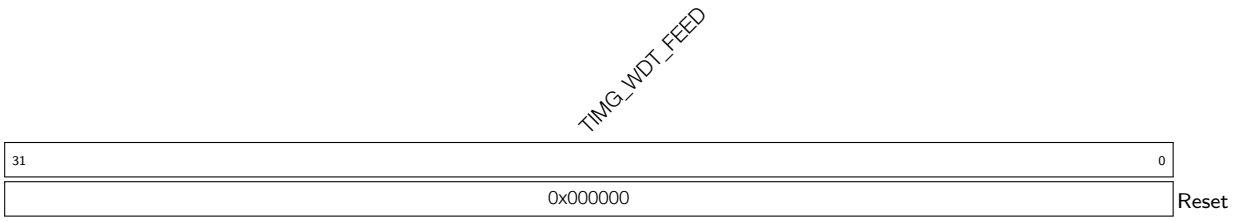


TIMG_WDT_STG2_HOLD Stage 2 timeout value, in MWDT clock cycles. (R/W)

Register 9.15: TIMG_WDTCONFIG5_REG (0x005C)

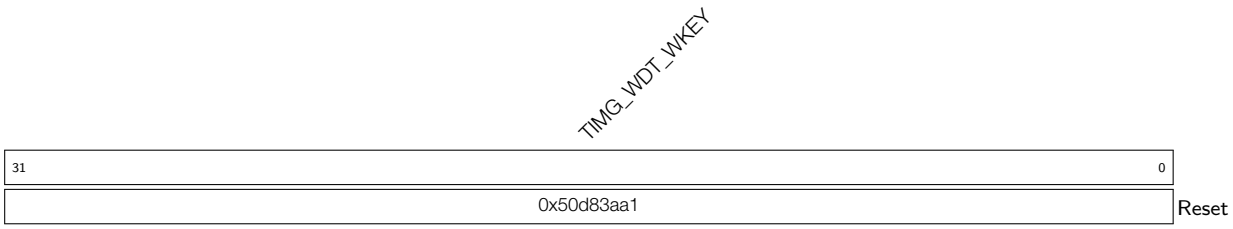
TIMG_WDT_STG3_HOLD Stage 3 timeout value, in MWDT clock cycles. (R/W)

Register 9.16: TIMG_WDTFEED_REG (0x0060)



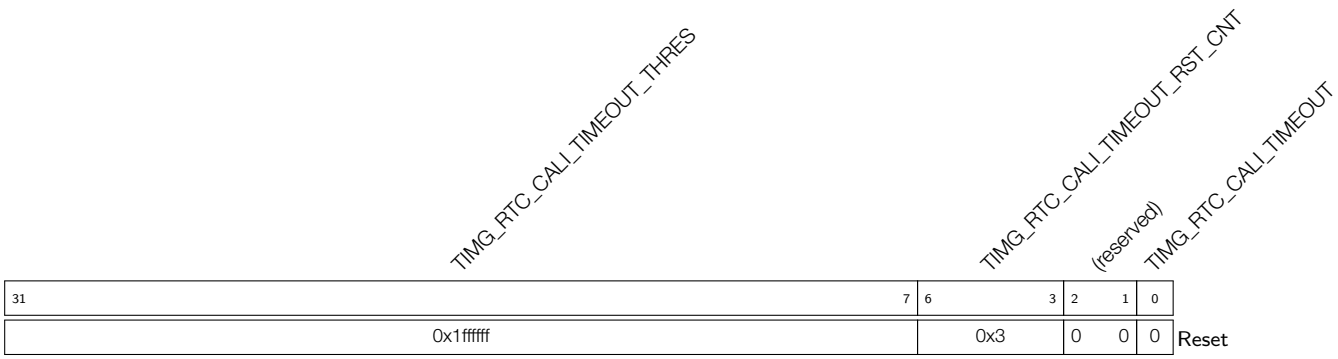
TIMG_WDT_FEED Write any value to feed the MWDT. (WO) (WO)

Register 9.17: TIMG_WDTWPROTECT_REG (0x0064)



TIMG_WDT_WKEY If the register contains a different value than its reset value, write protection is enabled. (R/W)

Register 9.18: TIMG_RTCCALICFG2_REG (0x00A8)

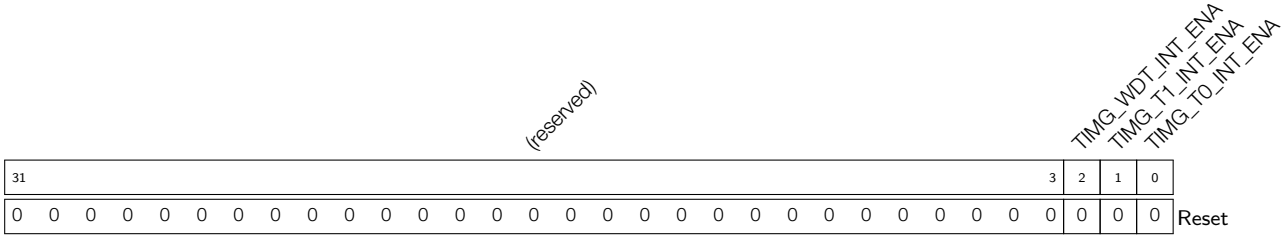


TIMG_RTC_CALI_TIMEOUT RTC calibration timeout indicator (RO)

TIMG_RTC_CALI_TIMEOUT_RST_CNT Cycles that release calibration timeout reset (R/W)

TIMG_RTC_CALI_TIMEOUT_THRES Threshold value for the RTC calibration timer. If the calibration timer's value exceeds this threshold, a timeout is triggered. (R/W)

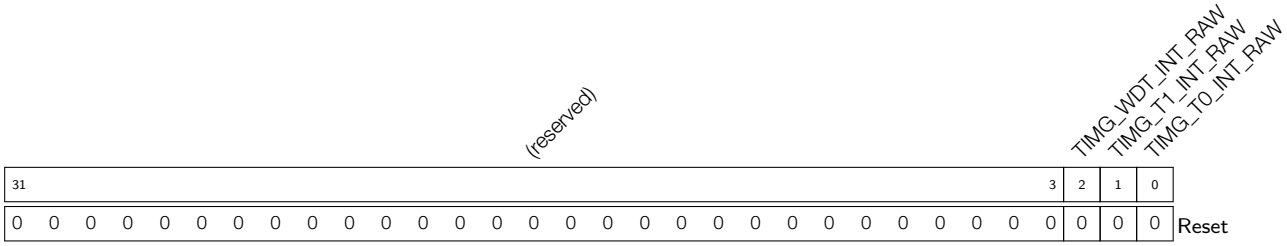
Register 9.19: TIMG_INT_ENA_TIMERS_REG (0x0098)



TIMG_T*x*_INT_ENA The interrupt enable bit for the TIMG_T*x*_INT interrupt. (R/W)

TIMG_WDT_INT_ENA The interrupt enable bit for the TIMG_WDT_INT interrupt. (R/W)

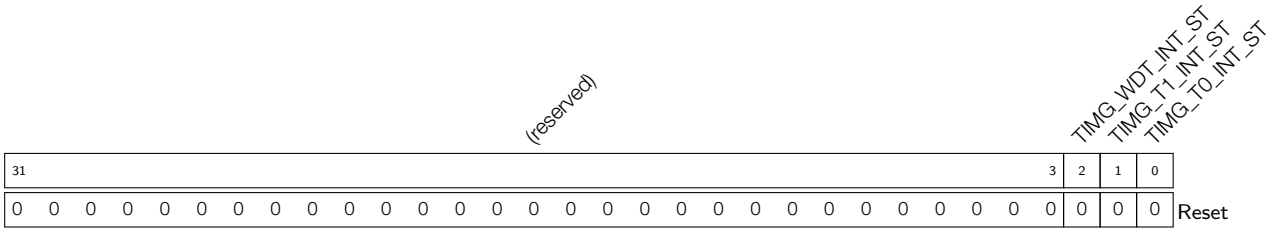
Register 9.20: TIMG_INT_RAW_TIMERS_REG (0x009C)



TIMG_T*x*_INT_RAW The raw interrupt status bit for the TIMG_T*x*_INT interrupt. (RO)

TIMG_WDT_INT_RAW The raw interrupt status bit for the TIMG_WDT_INT interrupt. (RO)

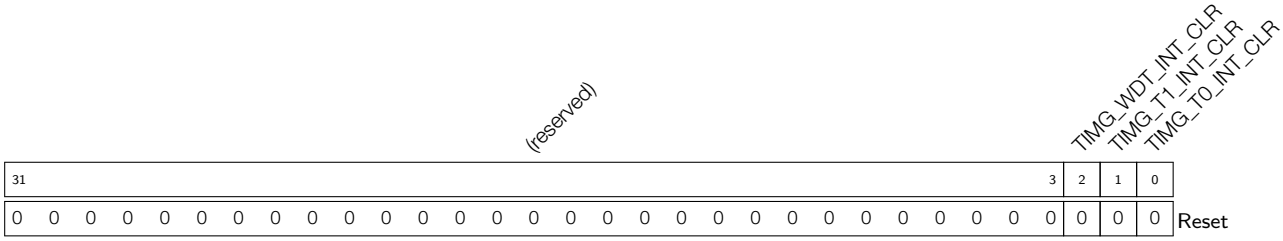
Register 9.21: TIMG_INT_ST_TIMERS_REG (0x00A0)



TIMG_T*x*_INT_ST The masked interrupt status bit for the TIMG_T*x*_INT interrupt. (RO)

TIMG_WDT_INT_ST The masked interrupt status bit for the TIMG_WDT_INT interrupt. (RO)

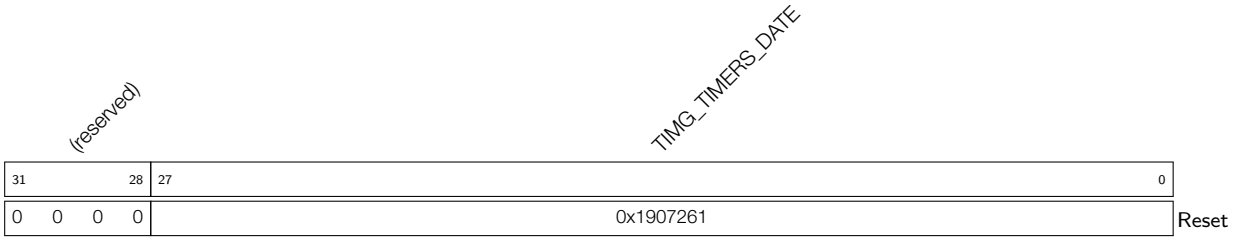
Register 9.22: TIMG_INT_CLR_TIMERS_REG (0x00A4)



TIMG_T*x*_INT_CLR Set this bit to clear the TIMG_T*x*_INT interrupt. (WO)

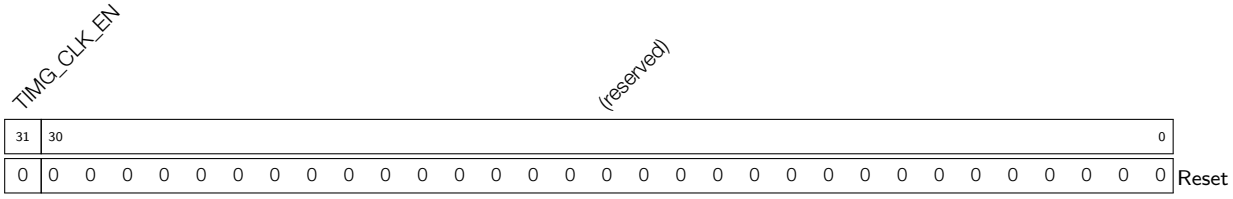
TIMG_WDT_INT_CLR Set this bit to clear the TIMG_WDT_INT interrupt. (WO)

Register 9.23: TIMG_TIMERS_DATE_REG (0x00F8)



TIMG_TIMERS_DATE Version control register. (R/W)

Register 9.24: TIMG_REGCLK_REG (0x00FC)



TIMG_CLK_EN Register clock gate signal. 1: Registers can be read and written to by software. 0: Registers can not be read or written to by software. (R/W)

10. Watchdog Timers

10.1 Overview

Watchdog timers are hardware timers used to detect and recover from malfunctions. They must be periodically fed (reset) to prevent a timeout. A system/software that is behaving unexpectedly (e.g. is stuck in a software loop or in overdue events) will fail to feed the watchdog thus trigger a watchdog time out. Therefore, watchdog timers are useful for detecting and handling erroneous system/software behavior.

The ESP32-S2 contains three watchdog timers: one in each of the two timer groups (called Main System Watchdog Timers, or MWDT) and one in the RTC Module (called the RTC Watchdog Timer, or RWDT). Each watchdog timer allows for four separately configurable stages and each stage can be programmed to take one of three (or four for RWDT) actions upon expiry, unless the watchdog is fed or disabled. The actions upon expiry are: interrupt, CPU reset, core reset and system reset. Only RWDT can trigger a system reset that will reset the entire digital circuits, which is the main system including the RTC itself. A timeout value can be set for each stage individually.

During the flash boot process, RWDT and the first MWDT are enabled automatically in order to detect and recover from booting errors.

Note that while this chapter provides the functional descriptions of the watchdog timer's, their register descriptions are provided in Chapter 9: *64-bit Timers*.

10.2 Features

Watchdog timers have the following features:

- Four stages, each with a programmable timeout value. Each stage can be configured and enabled/disabled separately
- One of three/four (for MWDTs/ RWDT) possible actions (interrupt, CPU reset, core reset and system reset) available upon expiry of each stage
- 32-bit expiry counter
- Write protection, to prevent RWDT and MWDT configuration from being altered inadvertently
- Flash boot protection
  If the boot process from an SPI flash does not complete within a predetermined period of time, the watchdog will reboot the entire main system.

10.3 Functional Description

10.3.1 Clock Source and 32-Bit Counter

At the core of each watchdog timer is a 32-bit counter. The clock source of MWDTs is derived from the APB clock via a pre-MWDT 16-bit configurable prescaler. Conversely, the clock source of RWDT is derived directly from a RTC slow clock (without a prescaler) which is usually running at 32 kHz. The 16-bit prescaler for MWDTs is configured via the TIMG_WDT_CLK_PRESCALER field of TIMG_WDTCONFIG1_REG.

MWDTs and RWDT are enabled by setting the TIMG_WDT_EN and RTC_CNTL_WDT_EN fields respectively. When enabled, the 32-bit counters of each watchdog will increment on each source clock cycle until the timeout value of the current stage is reached (i.e. expiry of the current stage). When this occurs, the current counter value is reset to zero and the next stage will become active. If a watchdog timer is fed by software, the timer will return to stage 0 and reset its counter value to zero. Software can feed a watchdog timer by writing any value to TIMG_WDTFEED_REG for MDWTs and RTC_CNTL_RTC_WDT_FEED for RWDT.

10.3.2 Stages and Timeout Actions

Timer stages allow for a timer to have a series of different timeout values and corresponding expiry action. When one stage expires, the expiry action is triggered, the counter value is reset to zero, and the next stage becomes active. MWDTs/ RWDT provide four stages (called stages 0 to 3). The watchdog timers will progress through each stage in a loop (i.e. from stage 0 to 3, then back to stage 0).

Timeout values of each stage for MWDTs are configured in TIMG_WDTCONFIG*i*_REG (where *i* ranges from 2 to 5), whilst timeout values for RWDT are configured in RTC_CNTL_WDT_STG*j*_HOLD_REG registers (where *j* ranges from 0 to 3).

Please note that the timeout value of stage 0 for RWDT (T_{hold0}) is determined together by EFUSE_WDT_DELAY_SEL field of an eFuse register and RTC_CTRL_WDT_STG0_HOLD_REG. The relationship is as follows:

T_{hold0} = RTC_CTRL_WDT_STG0_HOLD_REG « EFUSE_WDT_DELAY_SEL + 1

Upon the expiry of each stage, one of the following expiry actions will be executed:

- Trigger an interrupt
  When the stage expires, an interrupt is triggered.
- Reset a CPU core
  When the stage expires, the CPU core will be reset.
- Reset the main system
  When the stage expires, the main system (which includes MWDTs, CPU, and all peripherals) will be reset. The RTC will not be reset.
- Reset the main system and RTC
  When the stage expires the main system and the RTC will both be reset. This action is only available in RWDT.
- Disabled
  This stage will have no effects on the system.

For MWDTs, the expiry action of all stages is configured in TIMG_WDTCONFIG0_REG. Likewise for RWDT, the expiry action is configured in RTC_WDTCONFIG0.

10.3.3 Write Protection

Watchdog timers are critical to detecting and handling erroneous system/software behavior, thus should not be disabled easily (e.g. due to a misplaced register write). Therefore, MWDTs and RWDT incorporate a write protection mechanism that prevent the watchdogs from being disabled or tampered with due to an accidental write. The write protection mechanism is implemented using a write-key register for each timer (TIMG_WDT_WKEY for MWDT, RTC_CNTL_WDT_WKEY for RWDT). The value 0x50D83AA1 must be written to

the watchdog timer's write-key register before any other register of the same watchdog timer can be changed. Any attempts to write to a watchdog timer's registers (other than the write-key register itself) whilst the write-key register's value is not 0x50D83AA1 will be ignored. The recommended procedure for accessing a watchdog timer is as follows:

1. Disable the write protection by writing the value 0x50D83AA1 to the timer's write-key register.
2. Make the required modification of the watchdog such as feeding or changing its configuration.
3. Re-enable write protection by writing any value other than 0x50D83AA1 to the timer's write-key register.

10.3.4 Flash Boot Protection

During flash booting process, MWDT in timer group 0 (TIMG0), as well as RWDT, are automatically enabled. Stage 0 for the enabled MWDT is automatically configured to reset the system upon expiry. Likewise, stage 0 for RWDT is configured to reset the main system and RTC when it expires. After booting, TIMG_WDT_FLASHBOOT_MOD_EN and RTC_CNTL_WDT_FLASHBOOT_MOD_EN should be cleared to stop the flash boot protection procedure for both MWDT and RWDT respectively. After this, MWDT and RWDT can be configured by software.

10.4 Registers

MWDT registers are part of the timer submodule and are described in the Chapter 9: *64-bit Timers* Timer Registers section.

11. eFuse Controller

11.1 Overview

ESP32-S2 has a 4096-bit eFuse that stores parameters in the SoC. Once an eFuse bit is programmed to 1, it can never be reverted to 0. Software can instruct the eFuse Controller to program individual bits for individual parameters as needed. Some of these parameters can be read by software using the eFuse Controller registers, while some can be directly used by hardware modules.

11.2 Features

- One-time programmable storage
- Configurable write protection
- Configurable software read protection
- Parameters use different hardware encoding schemes to protect against corruption

11.3 Functional Description

11.3.1 Structure

There are 11 eFuse blocks (BLOCK0 ~ BLOCK10).

BLOCK0 holds most core system parameters. Among these parameters, 25 bits are invisible to software and can only be used by hardware and 37 bits are reserved for future use.

Table 11-1 lists all the parameters in BLOCK0 and their offsets, bit widths, functional description, as well as information on whether they can be used by hardware, and whether they are protected from programming.

The **EFUSE_WR_DIS** parameter is used to restrict the programming of other parameters, while **EFUSE_RD_DIS** is used to restrict software from reading BLOCK4 ~ BLOCK10. More information on these two parameters can be found in sections 11.3.1.1 and 11.3.1.2.

Table 11-1. Parameters in BLOCK0

Parameters	Offset	Bit Width	Hardware Use	Programming-Protection by EFUSE_WR_DIS Bit Number	Description
EFUSE_WR_DIS	0	32	Y	N/A	Disables programming of individual eFuses
EFUSE_RD_DIS	32	7	Y	0	Disables software reading from individual eFuse blocks BLOCK4-10
EFUSE_DIS_RTC_RAM_BOOT	39	1	N	1	Disables boot from RTC RAM
EFUSE_DIS_ICACHE	40	1	Y	2	Disables ICache
EFUSE_DIS_DCACHE	41	1	Y	2	Disables DCache

Parameters	Offset	Bit Width	Hardware Use	Programming-Protection by EFUSE_WR_DIS Bit Number	Description
EFUSE_DIS_DOWNLOAD_ICACHE	42	1	Y	2	Disables Icache when SoC is in Download mode
EFUSE_DIS_DOWNLOAD_DCACHE	43	1	Y	2	Disables Dcache when SoC is in Download mode
EFUSE_DIS_FORCE_DOWNLOAD	44	1	Y	2	Disables forcing chip into Download mode
EFUSE_DIS_USB	45	1	Y	2	Disables the USB OTG hardware
EFUSE_DIS_CAN	46	1	Y	2	Disables the TWAI Controller hardware
EFUSE_DIS_BOOT_REMAP	47	1	Y	2	Disables capability to Remap RAM to ROM address space
EFUSE_SOFT_DIS_JTAG	49	1	Y	2	Software disables JTAG. When software disabled, JTAG can be activated temporarily by HMAC peripheral.
EFUSE_HARD_DIS_JTAG	50	1	Y	2	Hardware disables JTAG permanently.
EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT	51	1	Y	2	Disables flash encryption when in download boot modes
EFUSE_USB_EXCHG_PINS	56	1	Y	30	Exchanges USB D+ and D- pins
EFUSE_EXT_PHY_ENABLE	57	1	N	30	Enables external USB PHY
EFUSE_USB_FORCE_NOPERSIST	58	1	N	30	Forces to set USB BVALID to 1
EFUSE_VDD_SPI_XPD	68	1	Y	3	If VDD_SPI_FORCE is 1, determines if the VDD_SPI regulator is powered on
EFUSE_VDD_SPI_TIEH	69	1	Y	3	If VDD_SPI_FORCE is 1, determines VDD_SPI voltage. 0: VDD_SPI connects to 1.8 V LDO; 1: VDD_SPI connects to VDD_RTC_IO
EFUSE_VDD_SPI_FORCE	70	1	Y	3	When set, XPD_VDD_PSI_REG and VDD_SPI_TIEH will be used to configure VDD_SPI LDO
EFUSE_WDT_DELAY_SEL	80	2	Y	3	Selects RTC WDT timeout threshold at startup

Parameters	Offset	Bit Width	Hardware Use	Programming-Protection by EFUSE_WR_DIS Bit Number	Description
EFUSE_SPI_BOOT_CRYPT_CNT	82	3	Y	4	Enables encryption and decryption, when an SPI boot mode is set. Feature is enabled when 1 or 3 bits are set in the eFuse, disabled otherwise.
EFUSE_SECURE_BOOT_KEY_REVOKE0	85	1	N	5	If set, revokes use of secure boot key digest 0
EFUSE_SECURE_BOOT_KEY_REVOKE1	86	1	N	6	If set, revokes use of secure boot key digest 1
EFUSE_SECURE_BOOT_KEY_REVOKE2	87	1	N	7	If set, revokes use of secure boot key digest 2
EFUSE_KEY_PURPOSE_0	88	4	Y	8	KEY0 purpose, see Table 11-2
EFUSE_KEY_PURPOSE_1	92	4	Y	9	KEY1 purpose, see Table 11-2
EFUSE_KEY_PURPOSE_2	96	4	Y	10	KEY2 purpose, see Table 11-2
EFUSE_KEY_PURPOSE_3	100	4	Y	11	KEY3 purpose, see Table 11-2
EFUSE_KEY_PURPOSE_4	104	4	Y	12	KEY4 purpose, see Table 11-2
EFUSE_KEY_PURPOSE_5	108	4	Y	13	KEY5 purpose, see Table 11-2
EFUSE_SECURE_BOOT_EN	116	1	N	15	Enables secure boot
EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE	117	1	N	16	Enables aggressive secure boot key revocation mode
EFUSE_FLASH_TPUW	124	4	N	18	Configures flash startup delay after SoC power-up, unit is (ms/2). When the value is 15, delay is 7.5 ms.
EFUSE_DIS_DOWNLOAD_MODE	128	1	N	18	Disables all Download boot modes
EFUSE_DIS_LEGACY_SPI_BOOT	129	1	N	18	Disables Legacy SPI boot mode
EFUSE_UART_PRINT_CONTROL	130	1	N	18	Selects the default UART for printing boot messages, 0: UART0; 1: UART1
EFUSE_DIS_USB_DOWNLOAD_MODE	132	1	N	18	Disables use of USB in UART download boot mode
EFUSE_ENABLE_SECURITY_DOWNLOAD	133	1	N	18	Enables secure UART download mode (read/write flash only)
EFUSE_UART_PRINT_CONTROL	134	2	N	18	Sets the default UART boot message output mode. 2'b00: Enabled; 2'b01: Enable when GPIO 46 is low at reset; 2'b10: Enable when GPIO 46 is high at rest; 2'b11: Disabled.

Parameters	Offset	Bit Width	Hardware Use	Programming-Protection by EFUSE_WR_DIS Bit Number	Description
EFUSE_PIN_POWER_SELECTION	136	1	N	18	Sets default power supply for GPIO33 ~ GPIO37, set when SPI flash is initialized. 0: VDD3P3_CPU; 1: VDD_SPI
EFUSE_FLASH_TYPE	137	1	N	18	Selects SPI flash type, 0: maximum four data lines; 1: eight data lines
EFUSE_FORCE_SEND_RESUME	138	1	N	18	Forces ROM code to send an SPI flash resume command during SPI boot
EFUSE_SECURE_VERSION	139	16	N	18	Secure version (used by ESP-IDF anti-rollback feature)

Table 11-2 lists all key purpose and their values. Setting the eFuse parameter EFUSE_KEY_PURPOSE_*n* programs the purpose for eFuse block KEY*n* (*n*: 0 ~ 5).

Table 11-2. Key Purpose Values

Key Purpose Values	Purposes
0	User purposes (software-only use)
1	Reserved
2	XTS_AES_256_KEY_1 (flash/PSRAM encryption)
3	XTS_AES_256_KEY_2 (flash/PSRAM encryption)
4	XTS_AES_128_KEY (flash/PSRAM encryption)
5	HMAC Downstream mode
6	JTAG soft enable key (uses HMAC Downstream mode)
7	Digital Signature peripheral key (uses HMAC Downstream mode)
8	HMAC Upstream mode
9	SECURE_BOOT_DIGEST0 (Secure Boot key digest)
10	SECURE_BOOT_DIGEST1 (Secure Boot key digest)
11	SECURE_BOOT_DIGEST2 (Secure Boot key digest)

Table 11-3 provides the details on the parameters in BLOCK1 ~ BLOCK10.

Table 11-3. Parameters in BLOCK1-10

BLOCK	Parameters	Bit Width	Hardware Use	Write Protection by EFUSE_WR_DIS Bit Number	Software Read Protection by EFUSE_RD_DIS Bit Number	Description
BLOCK1	EFUSE_MAC	48	N	20	N/A	MAC address
	EFUSE_SPI_PAD_CONFIGURE	[0:5]	N	20	N/A	CLK
		[6:11]	N	20	N/A	Q (D1)
		[12:17]	N	20	N/A	D (D0)
		[18:23]	N	20	N/A	CS

BLOCK	Parameters	Bit Width	Hardware Use	Write Protection by EFUSE_WR_DIS Bit Number	Software Read Protection by EFUSE_RD_DIS Bit Number	Description
		[24:29]	N	20	N/A	HD (D3)
		[30:35]	N	20	N/A	WP (D2)
		[36:41]	N	20	N/A	DQS
		[42:47]	N	20	N/A	D4
		[48:53]	N	20	N/A	D5
		[54:53]	N	20	N/A	D6
		[60:53]	N	20	N/A	D7
	EFUSE_SYS_DATA_PART0	78	N	20	N/A	System data
BLOCK2	EFUSE_SYS_DATA_PART1	256	N	21	N/A	System data
BLOCK3	EFUSE_USR_DATA	256	N	22	N/A	User data
BLOCK4	EFUSE_KEY0_DATA	256	Y	23	0	Key0 or user data
BLOCK5	EFUSE_KEY1_DATA	256	Y	24	1	Key1 or user data
BLOCK6	EFUSE_KEY2_DATA	256	Y	25	2	Key2 or user data
BLOCK7	EFUSE_KEY3_DATA	256	Y	26	3	Key3 or user data
BLOCK8	EFUSE_KEY4_DATA	256	Y	27	4	Key4 or user data
BLOCK9	EFUSE_KEY5_DATA	256	Y	28	5	Key5 or user data
BLOCK10	EFUSE_SYS_DATA_PART2	256	N	29	6	System data

Among these blocks, BLOCK4 ~ 9 stores KEY0 ~ 5, respectively. Up to six 256-bit keys can be programmed into eFuse. Whenever a key is programmed, its purpose value should also be programmed (see table 11-2). For example, a key for the JTAG function in HMAC Downstream mode is programmed to KEY3 (i.e., BLOCK7), then, the key purpose value 6 should be programmed to EFUSE_KEY_PURPOSE_3.

BLOCK1 ~ BLOCK10 use the RS coding scheme, so there are some restrictions on writing to these parameters (refer to Section 11.3.1.3: *Data Storage* and Section 11.3.2: *Software Programming of Parameters*).

11.3.1.1 EFUSE_WR_DIS

Parameter EFUSE_WR_DIS determines whether individual eFuse parameters are write-protected. After burning EFUSE_WR_DIS, execute an eFuse read operation so the new values will take effect (refer to *Updating eFuse read registers* in Section 11.3.3).

The columns "Programming-Protected by EFUSE_WR_DIS" in Table 11-1 and Table 11-3 list the specific bits of EFUSE_WR_DIS that determine the write protected status of each parameter.

When the corresponding bit is 0, the parameter is not write protected and can be programmed if the parameter has not been programmed.

When the corresponding bit is 1, the parameter is write protected and none of its bits can be modified. The non-programmed bits always remain 0, while programmed bits always remain 1.

11.3.1.2 EFUSE_RD_DIS

Only the eFuse blocks BLCOK4 ~ BLOCK10 can be individually software read protected. The corresponding bit in EFUSE_RD_DIS is shown in Table 11-3. After burning EFUSE_RD_DIS, execute an eFuse read operation so the new values will take effect (refer to *Updating eFuse read registers* in Section 11.3.3).

If the corresponding EFUSE_RD_DIS bit is 0, then the eFuse block can be read by software. If the corresponding EFUSE_RD_DIS bit is 1, then the parameter controlled by this bit is software read protected.

Other parameters that are not in BLOCK4 ~ BLOCK10 can always be read by software.

Although BLOCK4 ~ BLOCK10 can be set to read-protected, they can still be used by hardware modules, if the EFUSE_KEY_PURPOSE_*n* bit is set accordingly.

11.3.1.3 Data Storage

Internal to the SoC, eFuses use hardware encoding schemes to protect against data corruption.

All BLOCK0 parameters except for EFUSE_WR_DIS are stored with four backups, meaning each bit is stored four times. This backup scheme is automatically applied by the hardware and is not visible to software.

BLOCK1 ~ BLOCK10 use RS (44, 32) coding scheme that supports up to 5 bytes of automatic error correction. The primitive polynomial of RS (44, 32) is $p(x) = x^8 + x^4 + x^3 + x^2 + 1$. The shift register circuit that generates the check code is shown in Figure 11-1, where gf_mul_n (n is an integer) is the result of multiplying a byte of data in the $GF(2^8)$ field by the element α^n.

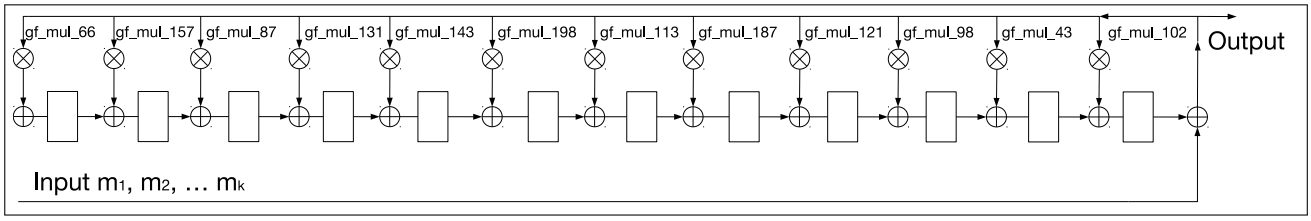


Figure 11-1. Shift Register Circuit

Software must encode the 32-byte parameter using RS (44, 32) to generate a 12-byte check code, and then burn the parameter and the check code into eFuse at the same time. The eFuse Controller automatically decodes the RS encoding and applies error correction when reading back the eFuse block.

Because the RS check codes are generated across the entire 256-bit eFuse block, each block can only be written to one time.

11.3.2 Software Programming of Parameters

The eFuse controller can only write to eFuse parameters in one block at a time. BLOCK0 ~ BLOCK10 share the same registers to store the parameters to be programmed. Configure parameter EFUSE_BLK_NUM to indicate which block is to be programmed.

Programming BLOCK0

When EFUSE_BLK_NUM = 0, BLOCK0 is programmed. Register EFUSE_PGM_DATA0_REG stores EFUSE_WR_DIS. EFUSE_PGM_DATA1_REG ~ EFUSE_PGM_DATA5_REG store the information of new BLOCK0 parameters to be programmed. Note that 25 BLOCK0 bits are reserved and can only be used by hardware. These must always be set to 0 in the programming registers. The specific bits are:

- EFUSE_PGM_DATA1_REG[29:31]
- EFUSE_PGM_DATA1_REG[20:23]
- EFUSE_PGM_DATA1_REG[16]
- EFUSE_PGM_DATA2_REG[7:15]
- EFUSE_PGM_DATA2_REG[0:3]
- EFUSE_PGM_DATA3_REG[16:19]

Values written to EFUSE_PGM_DATA6_REG ~ EFUSE_PGM_DATA7_REG and EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG are ignored when programming BLOCK0.

Programming BLOCK1

When EFUSE_BLK_NUM = 1, EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA5_REG store the parameters to be programmed. EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_DATA2_REG store the corresponding RS check codes. Values written to EFUSE_PGM_DATA6_REG ~ EFUSE_PGM_DATA7_REG are ignored when programming BLOCK1, the RS check codes should be calculated as if these bits were all 0.

Programming BLOCK2 ~ 10

When EFUSE_BLK_NUM = 2 ~ 10, EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG store the parameters to be programmed to this block. EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG store the corresponding RS check codes.

Programming process

The process of programming parameters is as follows:

1. Set EFUSE_BLK_NUM parameter as described above.
2. Write the parameters to be programmed into registers EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG and EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG.
3. Ensure the eFuse clock registers are set correctly as described in Section 11.3.4.1: *eFuse-Programming Timing*.
4. Ensure the eFuse programming voltage VDDQ is set correctly as described in Section 11.3.4.2: *eFuse VDDQ Setting*.
5. Set EFUSE_OP_CODE field in register EFUSE_CONF_REG to 0x5A5A.
6. Set EFUSE_PGM_CMD field in register EFUSE_CMD_REG to 1.
7. Poll register EFUSE_CMD_REG until it is 0x0, or wait for a pgm_done interrupt. Information on how to identify a pgm/read_done interrupt is provided at the end of Section 11.3.3.

8. Clear the parameters written into the register.
9. Trigger an eFuse read operation (see Section 11.3.3: *Software Reading of Parameters*) to update eFuse registers with the new values.

Limitations

For BLOCK0, the programming of different parameters and even the programming of different bits of the same parameter does not need to be done at once. It is, however, recommended that users minimize programming cycles and program all the bits of a parameter in one programming action. In addition, after all parameters controlled by a certain bit of EFUSE_WR_DIS are programmed, that bit should be immediately programmed. The programming of parameters controlled by a certain bit of EFUSE_WR_DIS, and the programming of the bit itself can even be completed at the same time. Repeated programming of already programmed bits is strictly forbidden, otherwise, programming errors will occur.

BLOCK1 cannot be programmed by users as it has been programmed at manufacturing.

BLOCK2 ~ 10 can only be programmed once. Repeated programming is not allowed.

11.3.3 Software Reading of Parameters

Software cannot read eFuse bits directly. The eFuse Controller hardware reads all eFuse bits and stores the results to their corresponding registers in the memory space. Then, software can read eFuse bits by reading the registers that start with EFUSE_RD_. Details are provided in the table below.

BLOCK	Read Registers	When Programming This Block
0	EFUSE_RD_WR_DIS_REG	EFUSE_PGM_DATA0_REG
0	EFUSE_RD_REPEAT_DATA0 ~ 4_REG	EFUSE_PGM_DATA1 ~ 5_REG
1	EFUSE_RD_MAC_SPI_SYS_0 ~ 5_REG	EFUSE_PGM_DATA0 ~ 5_REG
2	EFUSE_RD_SYS_DATA_PART1_0 ~ 7_REG	EFUSE_PGM_DATA0 ~ 7_REG
3	EFUSE_RD_USR_DATA0 ~ 7_REG	EFUSE_PGM_DATA0 ~ 7_REG
4-9	EFUSE_RD_KEY*n*_DATA0 ~ 7_REG (*n*: 0 ~ 5)	EFUSE_PGM_DATA0 ~ 7_REG
10	EFUSE_RD_SYS_DATA_PART2_0 ~ 7_REG	EFUSE_PGM_DATA0 ~ 7_REG

Updating eFuse read registers

The eFuse Controller hardware populates the read registers from the internal eFuse storage. This read operation happens on system reset and can also be triggered manually by software as needed, for example if new eFuse values have been programmed.

The process of triggering an eFuse controller read from software is as follows:

1. Configure the eFuse read timing registers as described in Section 11.3.4.3: *eFuse-Read Timing*.
2. Set the EFUSE_OP_CODE field in register EFUSE_CONF_REG to 0x5AA5.
3. Set the EFUSE_READ_CMD field in register EFUSE_CMD_REG to 1.
4. Poll the EFUSE_CMD_REG register until it is 0x0, or wait for a read_done interrupt. Information on how to identify a pgm/read_done interrupt is provided below.
5. Software reads the values of each parameter from memory.

The eFuse read registers will now hold updated values for all eFuse parameters.

Error detection

Error registers allow software to detect an inconsistency in the stored eFuses.

EFUSE_RD_REPEAT_ERR0 ~ 3_REG indicate inconsistencies in the stored backup copies of the parameters in BLOCK0 (except for EFUSE_WR_DIS) . Value 1 indicates an error was detected, and the bit became invalid. Value 0 indicates no error.

Registers EFUSE_RD_RS_ERR0 ~ 1_REG store the number of corrected bytes as well as the result of RS decoding during eFuse reading BLOCK1 ~ BLOCK10.

Software can read the values of above registers only after the eFuse read registers have been updated.

Identifying program/read operation completion

The two methods to identify the completion of program/read operation are described below. Please note that bit 1 corresponds to program operation, and bit 0 corresponds to read operation.

- Method one:
  1. Poll bit 1/0 in register EFUSE_INT_RAW_REG until it is 1, which represents the completion of a program/read operation.
- Method two:
  1. Set bit 1/0 in register EFUSE_INT_ENA_REG to 1 to enable eFuse Controller to post a pgm/read_done interrupt.
  2. Configure Interrupt Matrix to enable the CPU to respond to eFuse interrupt signal.
  3. Wait for the pgm/read_done interrupt.
  4. Set the bit 1/0 in register EFUSE_INT_CLR_REG to 1 to clear the pgm/read_done interrupt.

11.3.4 Timing

11.3.4.1 eFuse-Programming Timing

Figure 11-2 shows the timing for programming eFuse. Four registers EFUSE_TSUP_A, EFUSE_TPGM, EFUSE_THP_A, and EFUSE_TPGM_INACTIVE are used to configure the timing. Terms used in the timing diagrams in this section are described as follows:

- CSB: Chip select, active low
- VDDQ: eFuse programming voltage
- PGENB: eFuse programming enable signal, active low

The eFuse block uses the CLK_APB clock, which is configurable. Therefore, the timing parameters should be configured according to the specific clock frequency. After reset, the initial parameters are based on 20 MHz clock frequency.

Table 11-5. Configuration of eFuse-Programming Timing Parameters

APB Frequency	EFUSE_TSUP_A (> 6.669 ns)	EFUSE_TPGM (9-11 μs, usually 10 μs)	EFUSE_THP_A (> 6.166 ns)	EFUSE_TPGM_INACTIVE (> 35.96 ns)
80 MHz	0x1	0x320	0x1	0x3

APB Frequency	EFUSE_TSUP_A (> 6.669 ns)	EFUSE_TPGM (9-11 μs, usually 10 μs)	EFUSE_THP_A (> 6.166 ns)	EFUSE_TPGM_INACTIVE (> 35.96 ns)
40 MHz	0x1	0x190	0x1	0x2
20 MHz	0x1	0xC8	0x1	0x1

In Figure 11-2, Address A0 is programmed, then the corresponding eFuse bit is 1; Address A1 is not programmed, then the corresponding eFuse bit is 0.

11.3.4.2 eFuse VDDQ Timing Setting

VDDQ is the eFuse programming voltage, and its timing parameters should be configured according to the APB clock frequency:

Table 11-6. Configuration of VDDQ Timing Parameters

APB Frequency	EFUSE_DAC_CLK_DIV (> 1 μs)	EFUSE_PWR_ON_NUM (> EFUSE_DAC_CLK_DIV*255)	EFUSE_PWR_OFF_NUM (> 3 μs)
80 MHz	0xA0	0xA200	0x100
40 MHz	0x50	0x5100	0x80
20 MHz	0x28	0x2880	0x40

11.3.4.3 eFuse-Read Timing

Figure 11-3 shows the timing for reading eFuse. Three registers EFUSE_TSUR_A, EFUSE_TRD, and EFUSE_THR_A are used to configure the timing.

The parameters should be configured according to the specific APB clock frequency. Details can be found in the table below.

Table 11-7. Configuration of eFuse-Reading Parameters

APB Frequency	EFUSE_TSUR_A (> 6.669 ns)	EFUSE_TRD (> 35.96 ns)	EFUSE_THR_A (> 6.166 ns)
80 MHz	0x1	0x3	0x1
40 MHz	0x1	0x2	0x1
20 MHz	0x1	0x1	0x1

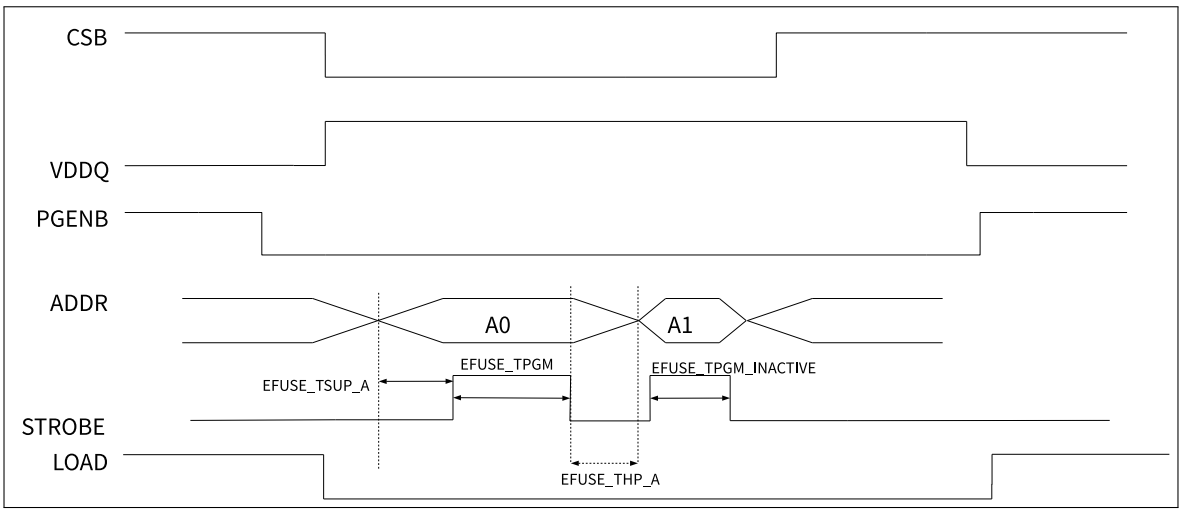


Figure 11-2. eFuse-Programming Timing Diagram

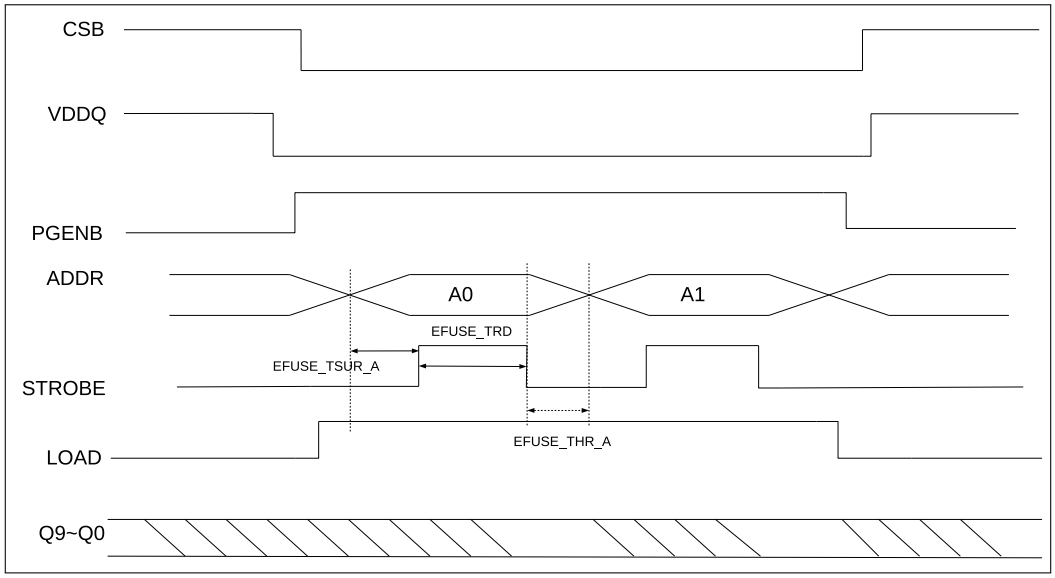


Figure 11-3. Timing Diagram for Reading eFuse

11.3.5 The Use of Parameters by Hardware Modules

Hardware modules are directly hardwired to the ESP32-S2 in order to use the parameters listed in Table 11-1 and 11-3, specifically those marked with "Y" in columns "Hardware Use". Software cannot change this behavior.

11.3.6 Interrupts

- pgm_done interrupt: Triggered when eFuse programming has finished. To enable this interrupt, set EFUSE_PGM_DONE_INT_ENA to 1.
- read_done interrupt: Triggered when eFuse reading has finished. To enable this interrupt, set EFUSE_READ_DONE_INT_ENA to 1.

11.4 Base Address

Users can access the eFuse Controller with two base addresses, which can be seen in the following table. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 11-8. eFuse Controller Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x6001A000
PeriBUS2	0x3FC1A000

11.5 Register Summary

The addresses in the following table are relative to the eFuse base addresses provided in Section 11.4.

Name	Description	Address	Access
PGM Data Registers			
EFUSE_PGM_DATA0_REG	Register 0 that stores data to be programmed.	0x0000	R/W
EFUSE_PGM_DATA1_REG	Register 1 that stores data to be programmed.	0x0004	R/W
EFUSE_PGM_DATA2_REG	Register 2 that stores data to be programmed.	0x0008	R/W
EFUSE_PGM_DATA3_REG	Register 3 that stores data to be programmed.	0x000C	R/W
EFUSE_PGM_DATA4_REG	Register 4 that stores data to be programmed.	0x0010	R/W
EFUSE_PGM_DATA5_REG	Register 5 that stores data to be programmed.	0x0014	R/W
EFUSE_PGM_DATA6_REG	Register 6 that stores data to be programmed.	0x0018	R/W
EFUSE_PGM_DATA7_REG	Register 7 that stores data to be programmed.	0x001C	R/W
EFUSE_PGM_CHECK_VALUE0_REG	Register 0 that stores the RS code to be programmed.	0x0020	R/W
EFUSE_PGM_CHECK_VALUE1_REG	Register 1 that stores the RS code to be programmed.	0x0024	R/W
EFUSE_PGM_CHECK_VALUE2_REG	Register 2 that stores the RS code to be programmed.	0x0028	R/W
Read Data Registers			
EFUSE_RD_WR_DIS_REG	Register 0 of BLOCK0.	0x002C	RO
EFUSE_RD_REPEAT_DATA0_REG	Register 1 of BLOCK0.	0x0030	RO
EFUSE_RD_REPEAT_DATA1_REG	Register 2 of BLOCK0.	0x0034	RO
EFUSE_RD_REPEAT_DATA2_REG	Register 3 of BLOCK0.	0x0038	RO
EFUSE_RD_REPEAT_DATA3_REG	Register 4 of BLOCK0.	0x003C	RO
EFUSE_RD_REPEAT_DATA4_REG	Register 5 of BLOCK0.	0x0040	RO
EFUSE_RD_MAC_SPI_SYS_0_REG	Register 0 of BLOCK1.	0x0044	RO
EFUSE_RD_MAC_SPI_SYS_1_REG	Register 1 of BLOCK1.	0x0048	RO
EFUSE_RD_MAC_SPI_SYS_2_REG	Register 2 of BLOCK1.	0x004C	RO
EFUSE_RD_MAC_SPI_SYS_3_REG	Register 3 of BLOCK1.	0x0050	RO
EFUSE_RD_MAC_SPI_SYS_4_REG	Register 4 of BLOCK1.	0x0054	RO
EFUSE_RD_MAC_SPI_SYS_5_REG	Register 5 of BLOCK1.	0x0058	RO

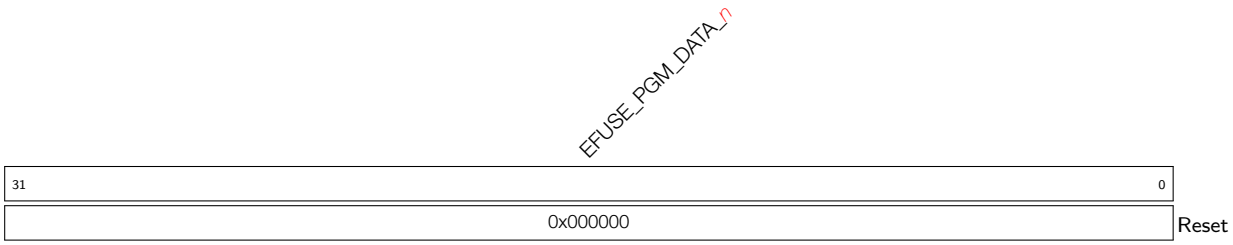
Name	Description	Address	Access
EFUSE_RD_SYS_DATA_PART1_0_REG	Register 0 of BLOCK2 (system).	0x005C	RO
EFUSE_RD_SYS_DATA_PART1_1_REG	Register 1 of BLOCK2 (system).	0x0060	RO
EFUSE_RD_SYS_DATA_PART1_2_REG	Register 2 of BLOCK2 (system).	0x0064	RO
EFUSE_RD_SYS_DATA_PART1_3_REG	Register 3 of BLOCK2 (system).	0x0068	RO
EFUSE_RD_SYS_DATA_PART1_4_REG	Register 4 of BLOCK2 (system).	0x006C	RO
EFUSE_RD_SYS_DATA_PART1_5_REG	Register 5 of BLOCK2 (system).	0x0070	RO
EFUSE_RD_SYS_DATA_PART1_6_REG	Register 6 of BLOCK2 (system).	0x0074	RO
EFUSE_RD_SYS_DATA_PART1_7_REG	Register 7 of BLOCK2 (system).	0x0078	RO
EFUSE_RD_USR_DATA0_REG	Register 0 of BLOCK3 (user).	0x007C	RO
EFUSE_RD_USR_DATA1_REG	Register 1 of BLOCK3 (user).	0x0080	RO
EFUSE_RD_USR_DATA2_REG	Register 2 of BLOCK3 (user).	0x0084	RO
EFUSE_RD_USR_DATA3_REG	Register 3 of BLOCK3 (user).	0x0088	RO
EFUSE_RD_USR_DATA4_REG	Register 4 of BLOCK3 (user).	0x008C	RO
EFUSE_RD_USR_DATA5_REG	Register 5 of BLOCK3 (user).	0x0090	RO
EFUSE_RD_USR_DATA6_REG	Register 6 of BLOCK3 (user).	0x0094	RO
EFUSE_RD_USR_DATA7_REG	Register 7 of BLOCK3 (user).	0x0098	RO
EFUSE_RD_KEY0_DATA0_REG	Register 0 of BLOCK4 (KEY0).	0x009C	RO
EFUSE_RD_KEY0_DATA1_REG	Register 1 of BLOCK4 (KEY0).	0x00A0	RO
EFUSE_RD_KEY0_DATA2_REG	Register 2 of BLOCK4 (KEY0).	0x00A4	RO
EFUSE_RD_KEY0_DATA3_REG	Register 3 of BLOCK4 (KEY0).	0x00A8	RO
EFUSE_RD_KEY0_DATA4_REG	Register 4 of BLOCK4 (KEY0).	0x00AC	RO
EFUSE_RD_KEY0_DATA5_REG	Register 5 of BLOCK4 (KEY0).	0x00B0	RO
EFUSE_RD_KEY0_DATA6_REG	Register 6 of BLOCK4 (KEY0).	0x00B4	RO
EFUSE_RD_KEY0_DATA7_REG	Register 7 of BLOCK4 (KEY0).	0x00B8	RO
EFUSE_RD_KEY1_DATA0_REG	Register 0 of BLOCK5 (KEY1).	0x00BC	RO
EFUSE_RD_KEY1_DATA1_REG	Register 1 of BLOCK5 (KEY1).	0x00C0	RO
EFUSE_RD_KEY1_DATA2_REG	Register 2 of BLOCK5 (KEY1).	0x00C4	RO
EFUSE_RD_KEY1_DATA3_REG	Register 3 of BLOCK5 (KEY1).	0x00C8	RO
EFUSE_RD_KEY1_DATA4_REG	Register 4 of BLOCK5 (KEY1).	0x00CC	RO
EFUSE_RD_KEY1_DATA5_REG	Register 5 of BLOCK5 (KEY1).	0x00D0	RO
EFUSE_RD_KEY1_DATA6_REG	Register 6 of BLOCK5 (KEY1).	0x00D4	RO
EFUSE_RD_KEY1_DATA7_REG	Register 7 of BLOCK5 (KEY1).	0x00D8	RO
EFUSE_RD_KEY2_DATA0_REG	Register 0 of BLOCK6 (KEY2).	0x00DC	RO
EFUSE_RD_KEY2_DATA1_REG	Register 1 of BLOCK6 (KEY2).	0x00E0	RO
EFUSE_RD_KEY2_DATA2_REG	Register 2 of BLOCK6 (KEY2).	0x00E4	RO
EFUSE_RD_KEY2_DATA3_REG	Register 3 of BLOCK6 (KEY2).	0x00E8	RO
EFUSE_RD_KEY2_DATA4_REG	Register 4 of BLOCK6 (KEY2).	0x00EC	RO
EFUSE_RD_KEY2_DATA5_REG	Register 5 of BLOCK6 (KEY2).	0x00F0	RO
EFUSE_RD_KEY2_DATA6_REG	Register 6 of BLOCK6 (KEY2).	0x00F4	RO
EFUSE_RD_KEY2_DATA7_REG	Register 7 of BLOCK6 (KEY2).	0x00F8	RO
EFUSE_RD_KEY3_DATA0_REG	Register 0 of BLOCK7 (KEY3).	0x00FC	RO
EFUSE_RD_KEY3_DATA1_REG	Register 1 of BLOCK7 (KEY3).	0x0100	RO

Name	Description	Address	Access
EFUSE_RD_KEY3_DATA2_REG	Register 2 of BLOCK7 (KEY3).	0x0104	RO
EFUSE_RD_KEY3_DATA3_REG	Register 3 of BLOCK7 (KEY3).	0x0108	RO
EFUSE_RD_KEY3_DATA4_REG	Register 4 of BLOCK7 (KEY3).	0x010C	RO
EFUSE_RD_KEY3_DATA5_REG	Register 5 of BLOCK7 (KEY3).	0x0110	RO
EFUSE_RD_KEY3_DATA6_REG	Register 6 of BLOCK7 (KEY3).	0x0114	RO
EFUSE_RD_KEY3_DATA7_REG	Register 7 of BLOCK7 (KEY3).	0x0118	RO
EFUSE_RD_KEY4_DATA0_REG	Register 0 of BLOCK8 (KEY4).	0x011C	RO
EFUSE_RD_KEY4_DATA1_REG	Register 1 of BLOCK8 (KEY4).	0x0120	RO
EFUSE_RD_KEY4_DATA2_REG	Register 2 of BLOCK8 (KEY4).	0x0124	RO
EFUSE_RD_KEY4_DATA3_REG	Register 3 of BLOCK8 (KEY4).	0x0128	RO
EFUSE_RD_KEY4_DATA4_REG	Register 4 of BLOCK8 (KEY4).	0x012C	RO
EFUSE_RD_KEY4_DATA5_REG	Register 5 of BLOCK8 (KEY4).	0x0130	RO
EFUSE_RD_KEY4_DATA6_REG	Register 6 of BLOCK8 (KEY4).	0x0134	RO
EFUSE_RD_KEY4_DATA7_REG	Register 7 of BLOCK8 (KEY4).	0x0138	RO
EFUSE_RD_KEY5_DATA0_REG	Register 0 of BLOCK9 (KEY5).	0x013C	RO
EFUSE_RD_KEY5_DATA1_REG	Register 1 of BLOCK9 (KEY5).	0x0140	RO
EFUSE_RD_KEY5_DATA2_REG	Register 2 of BLOCK9 (KEY5).	0x0144	RO
EFUSE_RD_KEY5_DATA3_REG	Register 3 of BLOCK9 (KEY5).	0x0148	RO
EFUSE_RD_KEY5_DATA4_REG	Register 4 of BLOCK9 (KEY5).	0x014C	RO
EFUSE_RD_KEY5_DATA5_REG	Register 5 of BLOCK9 (KEY5).	0x0150	RO
EFUSE_RD_KEY5_DATA6_REG	Register 6 of BLOCK9 (KEY5).	0x0154	RO
EFUSE_RD_KEY5_DATA7_REG	Register 7 of BLOCK9 (KEY5).	0x0158	RO
EFUSE_RD_SYS_DATA_PART2_0_REG	Register 0 of BLOCK10 (system).	0x015C	RO
EFUSE_RD_SYS_DATA_PART2_1_REG	Register 1 of BLOCK10 (system).	0x0160	RO
EFUSE_RD_SYS_DATA_PART2_2_REG	Register 2 of BLOCK10 (system).	0x0164	RO
EFUSE_RD_SYS_DATA_PART2_3_REG	Register 3 of BLOCK10 (system).	0x0168	RO
EFUSE_RD_SYS_DATA_PART2_4_REG	Register 4 of BLOCK10 (system).	0x016C	RO
EFUSE_RD_SYS_DATA_PART2_5_REG	Register 5 of BLOCK10 (system).	0x0170	RO
EFUSE_RD_SYS_DATA_PART2_6_REG	Register 6 of BLOCK10 (system).	0x0174	RO
EFUSE_RD_SYS_DATA_PART2_7_REG	Register 7 of BLOCK10 (system).	0x0178	RO
Error Status Registers			
EFUSE_RD_REPEAT_ERR0_REG	Programming error record register 0 of BLOCK0.	0x017C	RO
EFUSE_RD_REPEAT_ERR1_REG	Programming error record register 1 of BLOCK0.	0x0180	RO
EFUSE_RD_REPEAT_ERR2_REG	Programming error record register 2 of BLOCK0.	0x0184	RO
EFUSE_RD_REPEAT_ERR3_REG	Programming error record register 3 of BLOCK0.	0x0188	RO
EFUSE_RD_REPEAT_ERR4_REG	Programming error record register 4 of BLOCK0.	0x0190	RO
EFUSE_RD_RS_ERR0_REG	Programming error record register 0 of BLOCK1-10.	0x01C0	RO
EFUSE_RD_RS_ERR1_REG	Programming error record register 1 of BLOCK1-10.	0x01C4	RO
Control/Status Registers			
EFUSE_CLK_REG	eFuse clock configuration register.	0x01C8	R/W

Name	Description	Address	Access
EFUSE_CONF_REG	eFuse operation mode configuration register.	0x01CC	R/W
EFUSE_CMD_REG	eFuse command register.	0x01D4	R/W
EFUSE_DAC_CONF_REG	Controls the eFuse programming voltage.	0x01E8	R/W
EFUSE_STATUS_REG	eFuse status register.	0x01D0	RO
Interrupt Registers			
EFUSE_INT_RAW_REG	eFuse raw interrupt register.	0x01D8	RO
EFUSE_INT_ST_REG	eFuse interrupt status register.	0x01DC	RO
EFUSE_INT_ENA_REG	eFuse interrupt enable register.	0x01E0	R/W
EFUSE_INT_CLR_REG	eFuse interrupt clear register.	0x01E4	WO
Configuration Registers			
EFUSE_RD_TIM_CONF_REG	Configures read timing parameters.	0x01EC	R/W
EFUSE_WR_TIM_CONF0_REG	Configuration register 0 of eFuse programming timing parameters.	0x01F0	R/W
EFUSE_WR_TIM_CONF1_REG	Configuration register 1 of eFuse programming timing parameters.	0x01F4	R/W
EFUSE_WR_TIM_CONF2_REG	Configuration register 2 of eFuse programming timing parameters.	0x01F8	R/W
Version Register			
EFUSE_DATE_REG	Version control register.	0x01FC	R/W

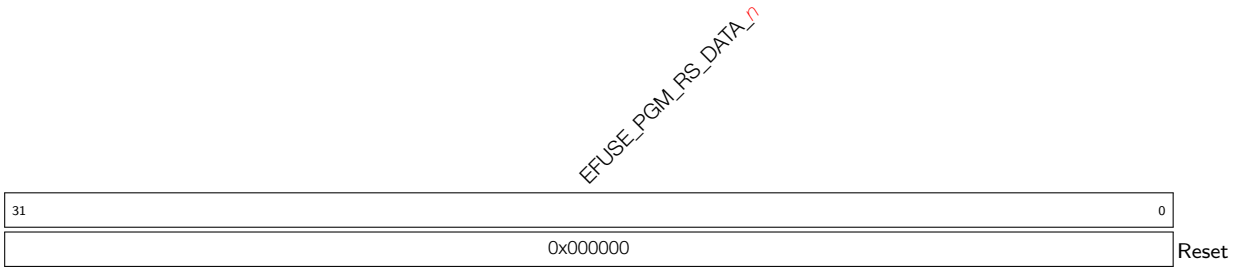
11.6 Registers

Register 11.1: EFUSE_PGM_DATA*n*_REG (*n*: 0-7) (0x0000+4n*)**



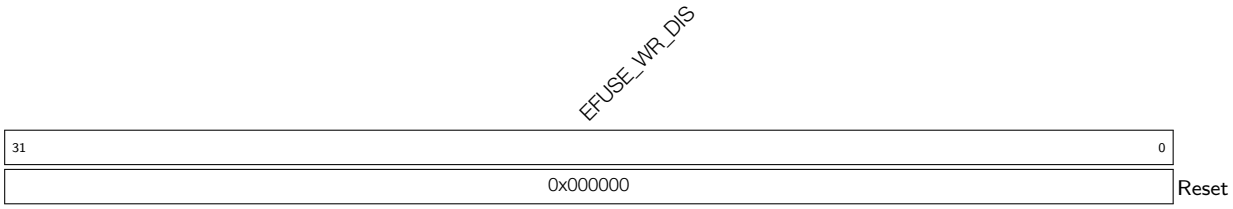
EFUSE_PGM_DATA_*n* The content of the *n*th 32-bit data to be programmed. (R/W)

Register 11.2: EFUSE_PGM_CHECK_VALUE*n*_REG (*n*: 0-2) (0x0020+4n*)**



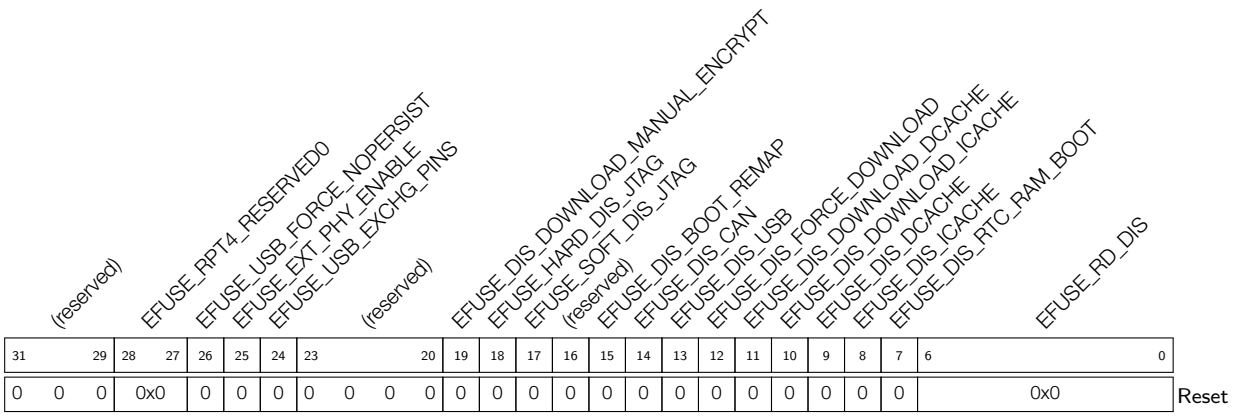
EFUSE_PGM_RS_DATA_*n* The content of the *n*th 32-bit RS code to be programmed. (R/W)

Register 11.3: EFUSE_RD_WR_DIS_REG (0x002C)



EFUSE_WR_DIS Disables programming of individual eFuses. (RO)

Register 11.4: EFUSE_RD_REPEAT_DATA0_REG (0x0030)



EFUSE_RD_DIS Disables software reading from individual eFuse blocks (BLOCK4-10). (RO)

EFUSE_DIS_RTC_RAM_BOOT Set this bit to disable boot from RTC RAM. (RO)

EFUSE_DIS_ICACHE Set this bit to disable ICache. (RO)

EFUSE_DIS_DCACHE Set this bit to disable DCache. (RO)

EFUSE_DIS_DOWNLOAD_ICACHE Disables Icache when SoC is in Download mode. (RO)

EFUSE_DIS_DOWNLOAD_DCACHE Disables Dcache when SoC is in Download mode. (RO)

EFUSE_DIS_FORCE_DOWNLOAD Set this bit to disable the function that forces chip into download mode. (RO)

EFUSE_DIS_USB Set this bit to disable USB OTG function. (RO)

EFUSE_DIS_CAN Set this bit to disable the TWAI Controller function. (RO)

EFUSE_DIS_BOOT_REMAP Disables capability to Remap RAM to ROM address space. (RO)

EFUSE_SOFT_DIS_JTAG Software disables JTAG. When software disabled, JTAG can be activated temporarily by HMAC peripheral. (RO)

EFUSE_HARD_DIS_JTAG Hardware disables JTAG permanently. (RO)

EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT Disables flash encryption when in download boot modes. (RO)

EFUSE_USB_EXCHG_PINS Set this bit to exchange USB D+ and D- pins. (RO)

EFUSE_EXT_PHY_ENABLE Set this bit to enable external USB PHY. (RO)

EFUSE_USB_FORCE_NOPERSIST If set, forces USB BVALID to 1. (RO)

EFUSE_RPT4_RESERVED0 Reserved (used for four backups method). (RO)

Register 11.5: EFUSE_RD_REPEAT_DATA1_REG (0x0034)

EFUSE_VDD_SPI_XPD If VDD_SPI_FORCE is 1, this value determines if the VDD_SPI regulator is powered on. (RO)

EFUSE_VDD_SPI_TIEH If VDD_SPI_FORCE is 1, determines VDD_SPI voltage. 0: VDD_SPI connects to 1.8 V LDO; 1: VDD_SPI connects to VDD_RTC_IO. (RO)

EFUSE_VDD_SPI_FORCE Set this bit to use XPD_VDD_PSI_REG and VDD_SPI_TIEH to configure VDD_SPI LDO. (RO)

EFUSE_WDT_DELAY_SEL Selects RTC watchdog timeout threshold at startup. 0: 40,000 slow clock cycles; 1: 80,000 slow clock cycles; 2: 160,000 slow clock cycles; 3: 320,000 slow clock cycles. (RO)

EFUSE_SPI_BOOT_CRYPT_CNT Enables encryption and decryption, when an SPI boot mode is set. Feature is enabled 1 or 3 bits are set in the eFuse, disabled otherwise. (RO)

EFUSE_SECURE_BOOT_KEY_REVOKE0 If set, revokes use of secure boot key digest 0. (RO)

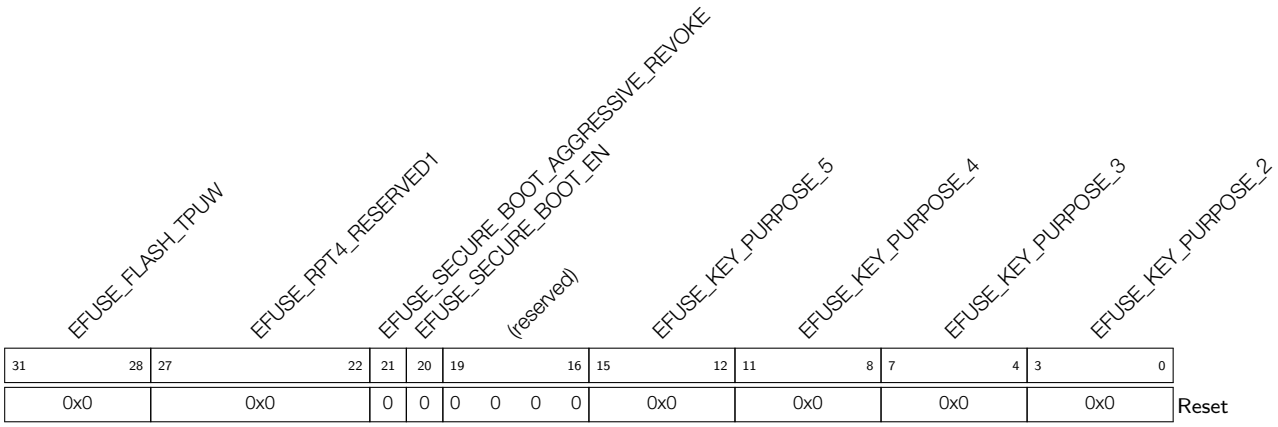
EFUSE_SECURE_BOOT_KEY_REVOKE1 If set, revokes use of secure boot key digest 1. (RO)

EFUSE_SECURE_BOOT_KEY_REVOKE2 If set, revokes use of secure boot key digest 2. (RO)

EFUSE_KEY_PURPOSE_0 Purpose of KEY0. Refer to Table 11-2 *Key Purpose Values*. (RO)

EFUSE_KEY_PURPOSE_1 Purpose of KEY1. Refer to Table 11-2 *Key Purpose Values*. (RO)

Register 11.6: EFUSE_RD_REPEAT_DATA2_REG (0x0038)



EFUSE_KEY_PURPOSE_2 Purpose of KEY2. Refer to Table 11-2 *Key Purpose Values*. (RO)

EFUSE_KEY_PURPOSE_3 Purpose of KEY3. Refer to Table 11-2 *Key Purpose Values*. (RO)

EFUSE_KEY_PURPOSE_4 Purpose of KEY4. Refer to Table 11-2 *Key Purpose Values*. (RO)

EFUSE_KEY_PURPOSE_5 Purpose of KEY5. Refer to Table 11-2 *Key Purpose Values*. (RO)

EFUSE_SECURE_BOOT_EN Set this bit to enable secure boot. (RO)

EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE Set this bit to enable aggressive secure boot key revocation mode. (RO)

EFUSE_RPT4_RESERVED1 Reserved (used for four backups method). (RO)

EFUSE_FLASH_TPUW Configures flash startup delay after SoC power-up, in unit of (ms/2). When the value is 15, delay is 7.5 ms. (RO)

Register 11.7: EFUSE_RD_REPEAT_DATA3_REG (0x003C)

EFUSE_DIS_DOWNLOAD_MODE Set this bit to disable all download boot modes. (RO)

EFUSE_DIS_LEGACY_SPI_BOOT Set this bit to disable Legacy SPI boot mode. (RO)

EFUSE_UART_PRINT_CHANNEL Selects the default UART for printing boot messages. 0: UART0; 1: UART1. (RO)

EFUSE_RPT4_RESERVED3 Reserved (used for four backups method). (RO)

EFUSE_DIS_USB_DOWNLOAD_MODE Set this bit to disable use of USB OTG in UART download boot mode. (RO)

EFUSE_ENABLE_SECURITY_DOWNLOAD Set this bit to enable secure UART download mode (read/write flash only). (RO)

EFUSE_UART_PRINT_CONTROL Set the default UART boot message output mode. 00: Enabled. 01: Enable when GPIO46 is low at reset. 10: Enable when GPIO46 is high at reset. 11: Disabled. (RO)

EFUSE_PIN_POWER_SELECTION Set default power supply for GPIO33-GPIO37, set when SPI flash is initialized. 0: VDD3P3_CPU; 1: VDD_SPI. (RO)

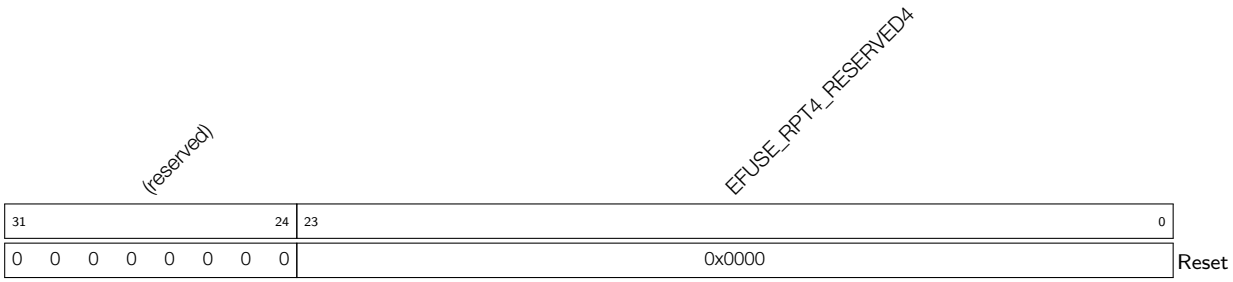
EFUSE_FLASH_TYPE SPI flash type. 0: maximum four data lines, 1: eight data lines. (RO)

EFUSE_FORCE_SEND_RESUME If set, forces ROM code to send an SPI flash resume command during SPI boot. (RO)

EFUSE_SECURE_VERSION Secure version (used by ESP-IDF anti-rollback feature). (RO)

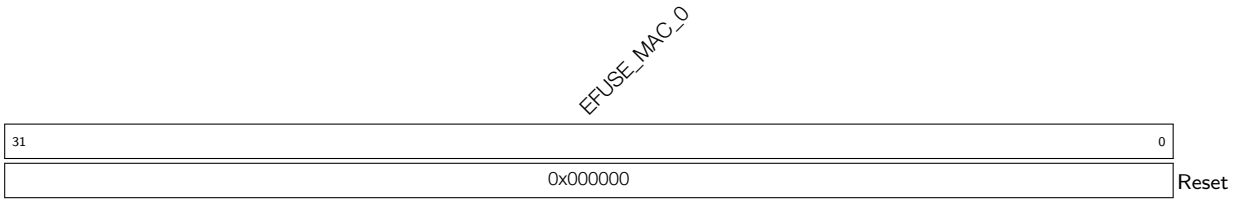
EFUSE_RPT4_RESERVED2 Reserved (used for four backups method). (RO)

Register 11.8: EFUSE_RD_REPEAT_DATA4_REG (0x0040)



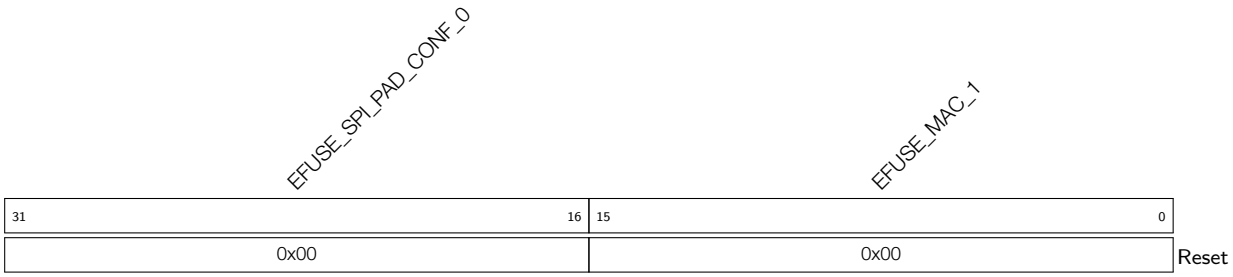
EFUSE_RPT4_RESERVED4 Reserved (used for four backups method). (RO)

Register 11.9: EFUSE_RD_MAC_SPI_SYS_0_REG (0x0044)



EFUSE_MAC_0 Stores the low 32 bits of MAC address. (RO)

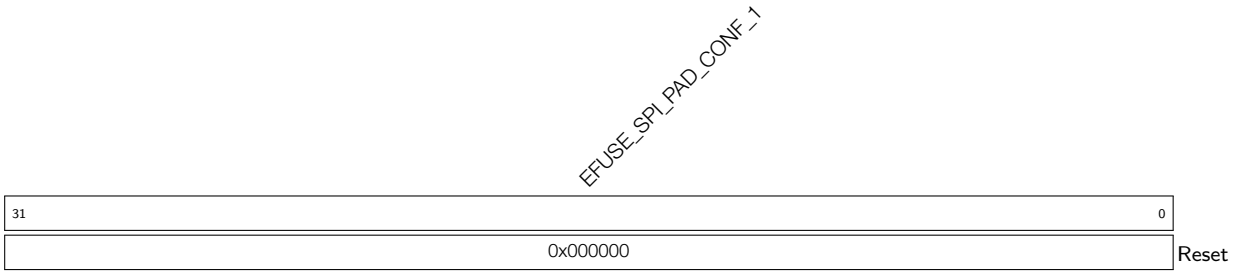
Register 11.10: EFUSE_RD_MAC_SPI_SYS_1_REG (0x0048)



EFUSE_MAC_1 Stores the high 16 bits of MAC address. (RO)

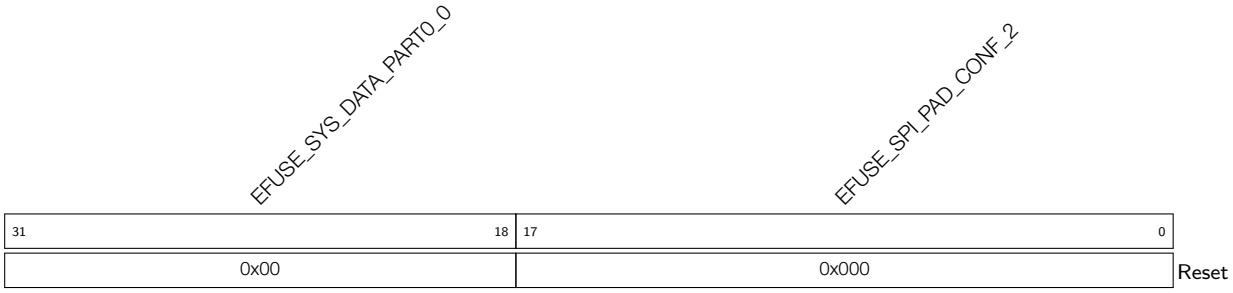
EFUSE_SPI_PAD_CONF_0 Stores the zeroth part of SPI_PAD_CONF. (RO)

Register 11.11: EFUSE_RD_MAC_SPI_SYS_2_REG (0x004C)



EFUSE_SPI_PAD_CONF_1 Stores the first part of SPI_PAD_CONF. (RO)

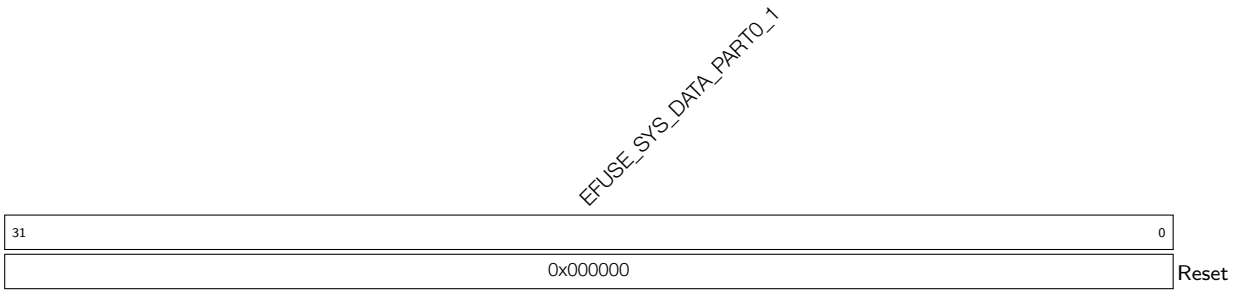
Register 11.12: EFUSE_RD_MAC_SPI_SYS_3_REG (0x0050)



EFUSE_SPI_PAD_CONF_2 Stores the second part of SPI_PAD_CONF. (RO)

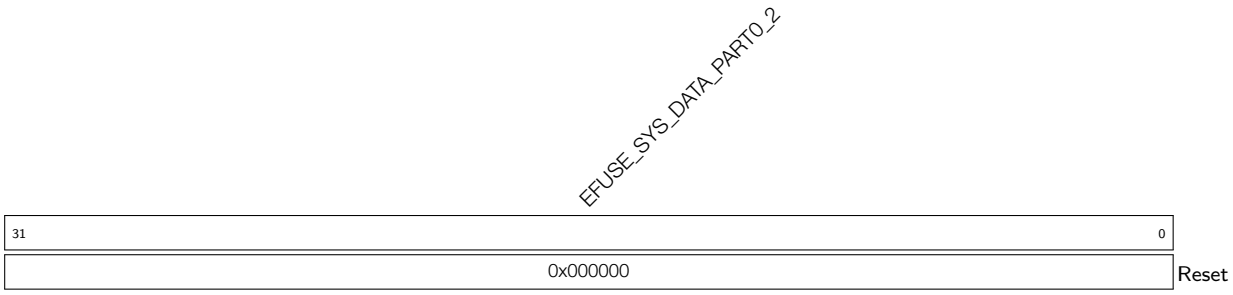
EFUSE_SYS_DATA_PART0_0 Stores the zeroth part of the zeroth part of system data. (RO)

Register 11.13: EFUSE_RD_MAC_SPI_SYS_4_REG (0x0054)



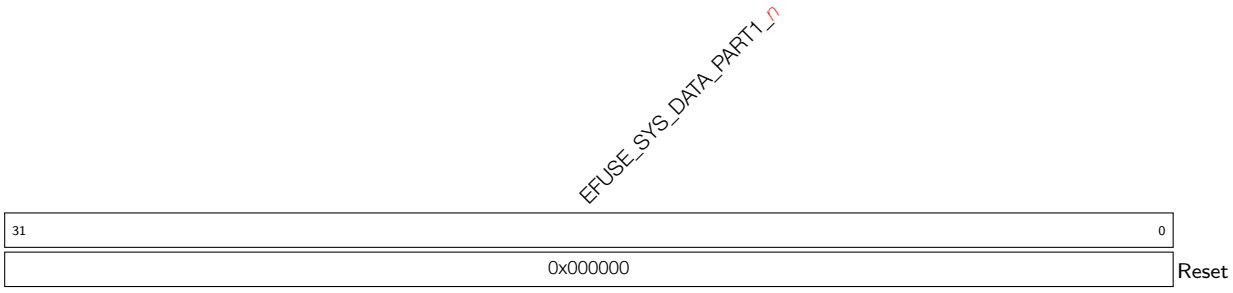
EFUSE_SYS_DATA_PART0_1 Stores the fist part of the zeroth part of system data. (RO)

Register 11.14: EFUSE_RD_MAC_SPI_SYS_5_REG (0x0058)



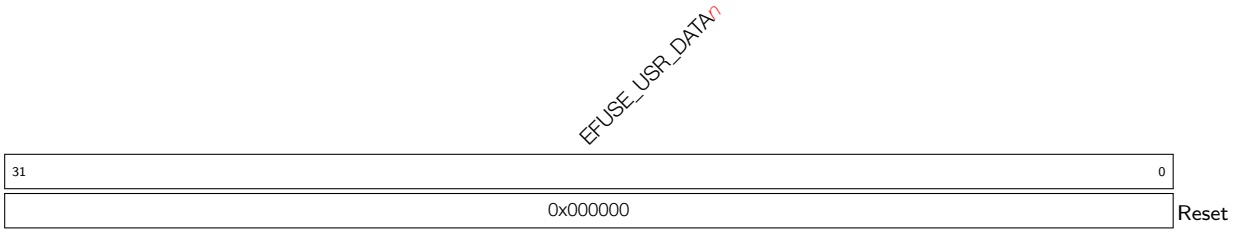
EFUSE_SYS_DATA_PART0_2 Stores the second part of the zeroth part of system data. (RO)

Register 11.15: EFUSE_RD_SYS_DATA_PART1_*n*_REG (*n*: 0-7) (0x005C+4n*)**



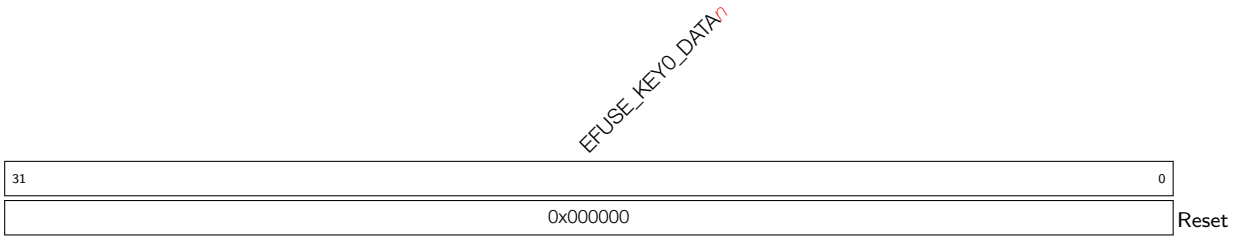
EFUSE_SYS_DATA_PART1_*n* Stores the *n*th 32 bits of the first part of system data. (RO)

Register 11.16: EFUSE_RD_USR_DATA*n*_REG (*n*: 0-7) (0x007C+4n*)**



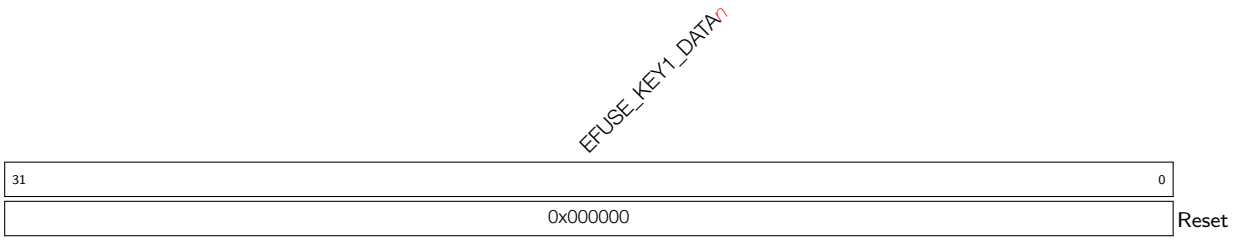
EFUSE_USR_DATA*n* Stores the *n*th 32 bits of BLOCK3 (user). (RO)

Register 11.17: EFUSE_RD_KEY0_DATA*n*_REG (*n*: 0-7) (0x009C+4n*)**



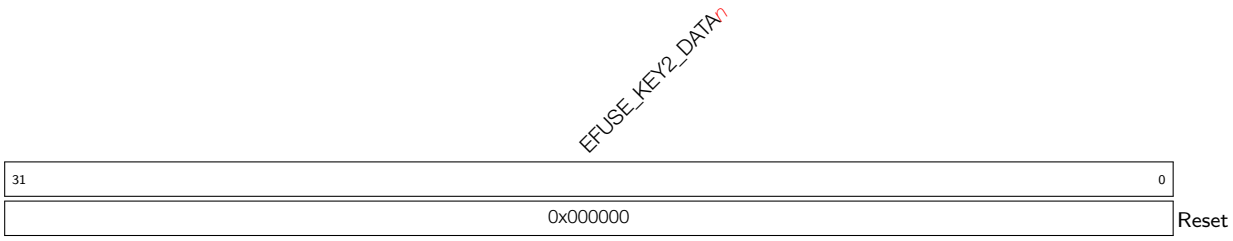
EFUSE_KEY0_DATA*n* Stores the *n*th 32 bits of KEY0. (RO)

Register 11.18: EFUSE_RD_KEY1_DATA*n*_REG (*n*: 0-7) (0x00BC+4n*)**



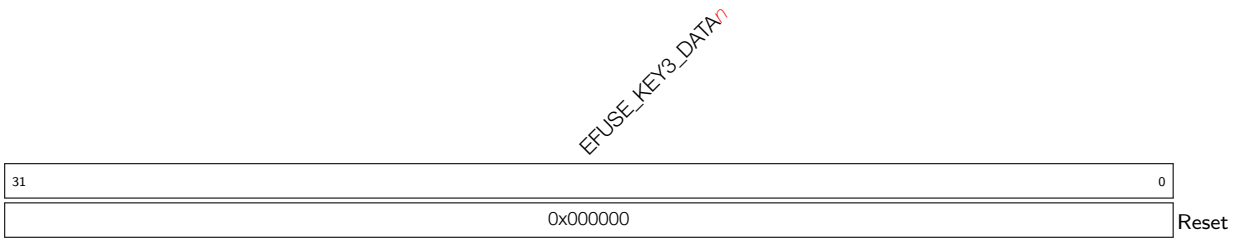
EFUSE_KEY1_DATA*n* Stores the *n*th 32 bits of KEY1. (RO)

Register 11.19: EFUSE_RD_KEY2_DATA*n*_REG (*n*: 0-7) (0x00DC+4n*)**



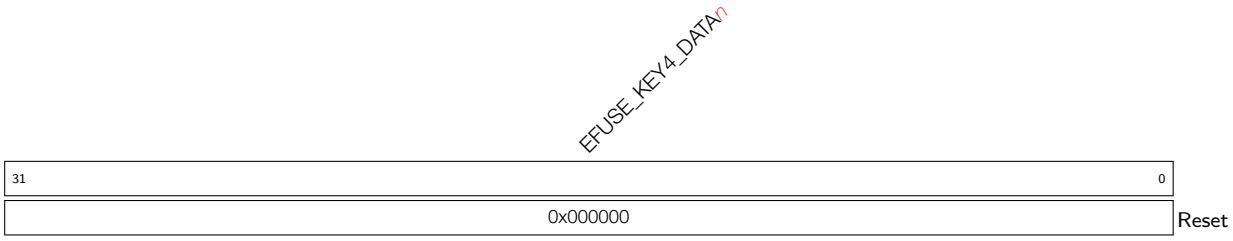
EFUSE_KEY2_DATA*n* Stores the *n*th 32 bits of KEY2. (RO)

Register 11.20: EFUSE_RD_KEY3_DATA*n*_REG (*n*: 0-7) (0x00FC+4n*)**



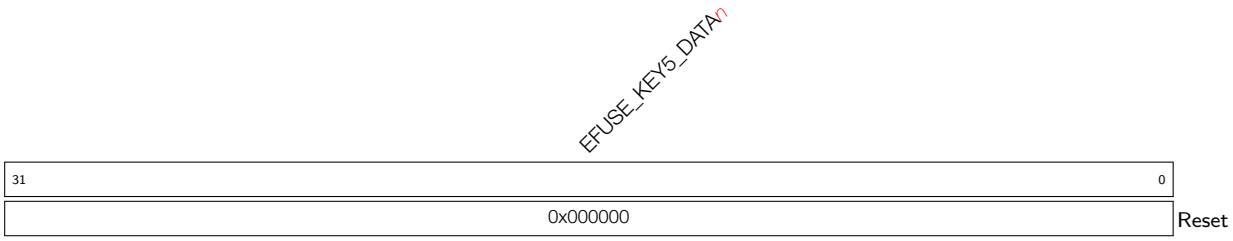
EFUSE_KEY3_DATA*n* Stores the *n*th 32 bits of KEY3. (RO)

Register 11.21: EFUSE_RD_KEY4_DATA*n*_REG (*n*: 0-7) (0x011C+4n*)**



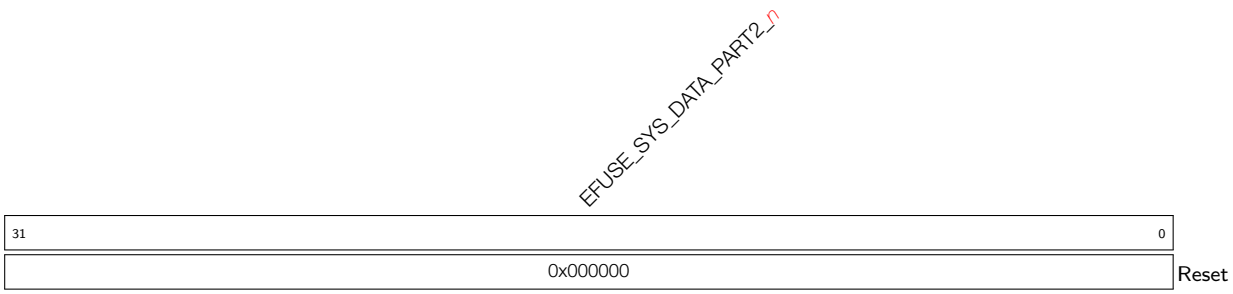
EFUSE_KEY4_DATA*n* Stores the *n*th 32 bits of KEY4. (RO)

Register 11.22: EFUSE_RD_KEY5_DATA*n*_REG (*n*: 0-7) (0x013C+4n*)**



EFUSE_KEY5_DATA*n* Stores the *n*th 32 bits of KEY5. (RO)

Register 11.23: EFUSE_RD_SYS_DATA_PART2_*n*_REG (*n*: 0-7) (0x015C+4n*)**



EFUSE_SYS_DATA_PART2_*n* Stores the *n*th 32 bits of the 2nd part of system data. (RO)

Register 11.24: EFUSE_RD_REPEAT_ERR0_REG (0x017C)

EFUSE_RD_DIS_ERR Any bit equal to 1 denotes a programming error in EFUSE_RD_DIS. (RO)

EFUSE_DIS_RTC_RAM_BOOT_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_RTC_RAM_BOOT. (RO)

EFUSE_DIS_ICACHE_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_ICACHE. (RO)

EFUSE_DIS_DCACHE_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_DCACHE. (RO)

EFUSE_DIS_DOWNLOAD_ICACHE_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_DOWNLOAD_ICACHE. (RO)

EFUSE_DIS_DOWNLOAD_DCACHE_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_DOWNLOAD_DCACHE. (RO)

EFUSE_DIS_FORCE_DOWNLOAD_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_FORCE_DOWNLOAD. (RO)

EFUSE_DIS_USB_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_USB. (RO)

EFUSE_DIS_CAN_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_CAN. (RO)

EFUSE_DIS_BOOT_REMAP_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_BOOT_REMAP. (RO)

EFUSE_SOFT_DIS_JTAG_ERR Any bit equal to 1 denotes a programming error in EFUSE_SOFT_DIS_JTAG. (RO)

EFUSE_HARD_DIS_JTAG_ERR Any bit equal to 1 denotes a programming error in EFUSE_HARD_DIS_JTAG. (RO)

EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT. (RO)

EFUSE_USB_EXCHG_PINS_ERR Any bit equal to 1 denotes a programming error in EFUSE_USB_EXCHG_PINS. (RO)

Continued on the next page...

Register 11.24: EFUSE_RD_REPEAT_ERR0_REG (0x017C)

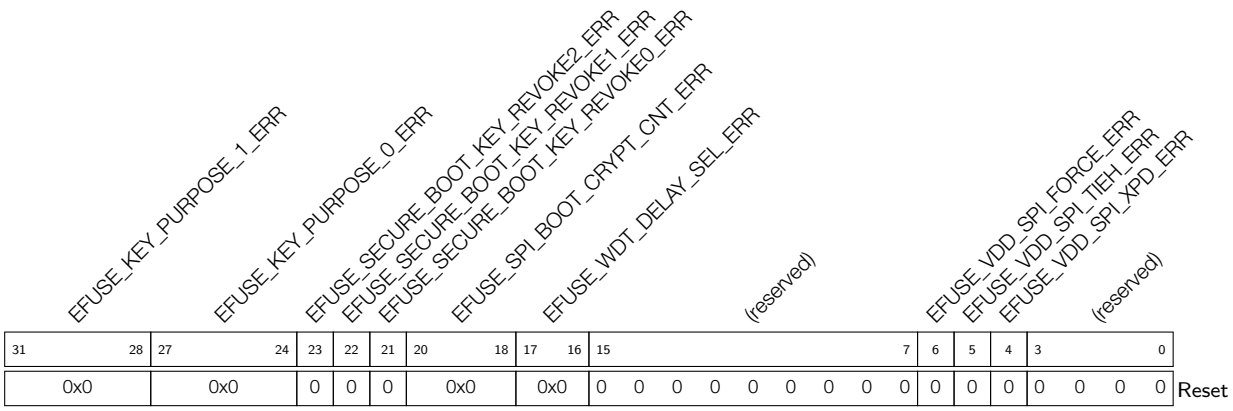
Continued from the previous page...

EFUSE_EXT_PHY_ENABLE_ERR Any bit equal to 1 denotes a programming error in EFUSE_EXT_PHY_ENABLE. (RO)

EFUSE_USB_FORCE_NOPERSIST_ERR Any bit equal to 1 denotes a programming error in EFUSE_USB_FORCE_NOPERSIST. (RO)

EFUSE_RPT4_RESERVED0_ERR Any bit equal to 1 denotes a programming error in EFUSE_RPT4_RESERVED0. (RO)

Register 11.25: EFUSE_RD_REPEAT_ERR1_REG (0x0180)



EFUSE_VDD_SPI_XPD_ERR Any bit equal to 1 denotes a programming error in EFUSE_VDD_SPI_XPD. (RO)

EFUSE_VDD_SPI_TIEH_ERR Any bit equal to 1 denotes a programming error in EFUSE_VDD_SPI_TIEH. (RO)

EFUSE_VDD_SPI_FORCE_ERR Any bit equal to 1 denotes a programming error in EFUSE_VDD_SPI_FORCE. (RO)

EFUSE_WDT_DELAY_SEL_ERR Any bit equal to 1 denotes a programming error in EFUSE_WDT_DELAY_SEL. (RO)

EFUSE_SPI_BOOT_CRYPT_CNT_ERR Any bit equal to 1 denotes a programming error in EFUSE_SPI_BOOT_CRYPT_CNT. (RO)

EFUSE_SECURE_BOOT_KEY_REVOKE0_ERR Any bit equal to 1 denotes a programming error in EFUSE_SECURE_BOOT_KEY_REVOKE0. (RO)

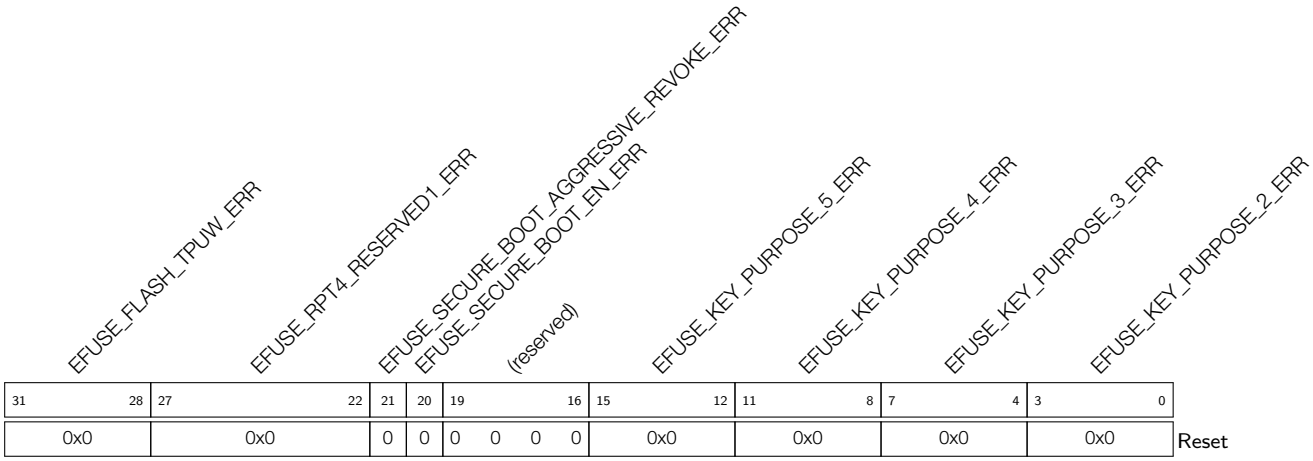
EFUSE_SECURE_BOOT_KEY_REVOKE1_ERR Any bit equal to 1 denotes a programming error in EFUSE_SECURE_BOOT_KEY_REVOKE1. (RO)

EFUSE_SECURE_BOOT_KEY_REVOKE2_ERR Any bit equal to 1 denotes a programming error in EFUSE_SECURE_BOOT_KEY_REVOKE2. (RO)

EFUSE_KEY_PURPOSE_0_ERR Any bit equal to 1 denotes a programming error in EFUSE_KEY_PURPOSE_0. (RO)

EFUSE_KEY_PURPOSE_1_ERR Any bit equal to 1 denotes a programming error in EFUSE_KEY_PURPOSE_1. (RO)

Register 11.26: EFUSE_RD_REPEAT_ERR2_REG (0x0184)



EFUSE_KEY_PURPOSE_2_ERR Any bit equal to 1 denotes a programming error in EFUSE_KEY_PURPOSE_2. (RO)

EFUSE_KEY_PURPOSE_3_ERR Any bit equal to 1 denotes a programming error in EFUSE_KEY_PURPOSE_3. (RO)

EFUSE_KEY_PURPOSE_4_ERR Any bit equal to 1 denotes a programming error in EFUSE_KEY_PURPOSE_4. (RO)

EFUSE_KEY_PURPOSE_5_ERR Any bit equal to 1 denotes a programming error in EFUSE_KEY_PURPOSE_5. (RO)

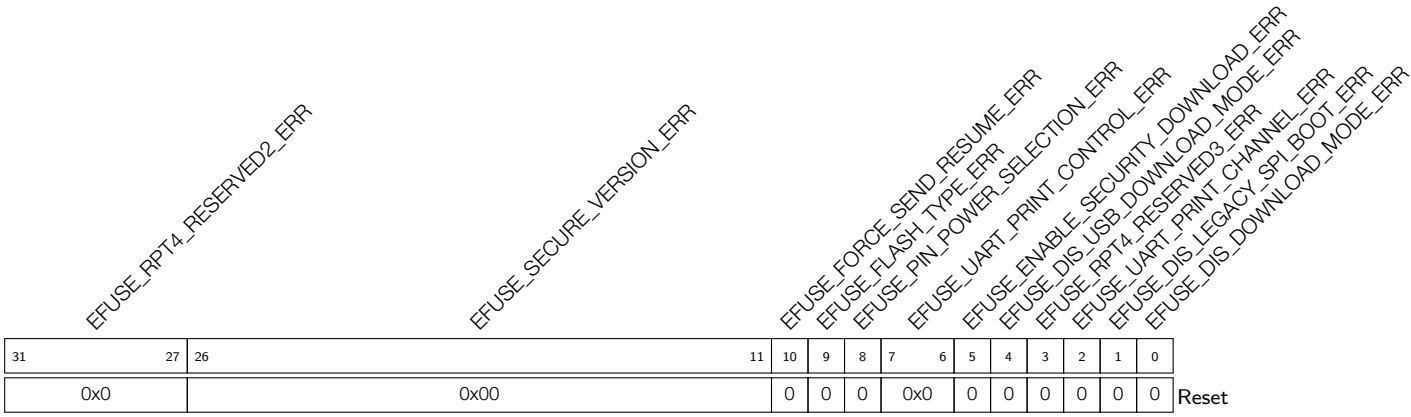
EFUSE_SECURE_BOOT_EN_ERR Any bit equal to 1 denotes a programming error in EFUSE_SECURE_BOOT_EN. (RO)

EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE_ERR Any bit equal to 1 denotes a programming error in EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE. (RO)

EFUSE_RPT4_RESERVED1_ERR Any bit equal to 1 denotes a programming error in EFUSE_RPT4_RESERVED1. (RO)

EFUSE_FLASH_TPUW_ERR Any bit equal to 1 denotes a programming error in EFUSE_FLASH_TPUW. (RO)

Register 11.27: EFUSE_RD_REPEAT_ERR3_REG (0x0188)



EFUSE_DIS_DOWNLOAD_MODE_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_DOWNLOAD_MODE. (RO)

EFUSE_DIS_LEGACY_SPI_BOOT_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_LEGACY_SPI_BOOT. (RO)

EFUSE_UART_PRINT_CHANNEL_ERR Any bit equal to 1 denotes a programming error in EFUSE_UART_PRINT_CHANNEL. (RO)

EFUSE_RPT4_RESERVED3_ERR Any bit equal to 1 denotes a programming error in EFUSE_RPT4_RESERVED3. (RO)

EFUSE_DIS_USB_DOWNLOAD_MODE_ERR Any bit equal to 1 denotes a programming error in EFUSE_DIS_USB_DOWNLOAD_MODE. (RO)

EFUSE_ENABLE_SECURITY_DOWNLOAD_ERR Any bit equal to 1 denotes a programming error in EFUSE_ENABLE_SECURITY_DOWNLOAD. (RO)

EFUSE_UART_PRINT_CONTROL_ERR Any bit equal to 1 denotes a programming error in EFUSE_UART_PRINT_CONTROL. (RO)

EFUSE_PIN_POWER_SELECTION_ERR Any bit equal to 1 denotes a programming error in EFUSE_PIN_POWER_SELECTION. (RO)

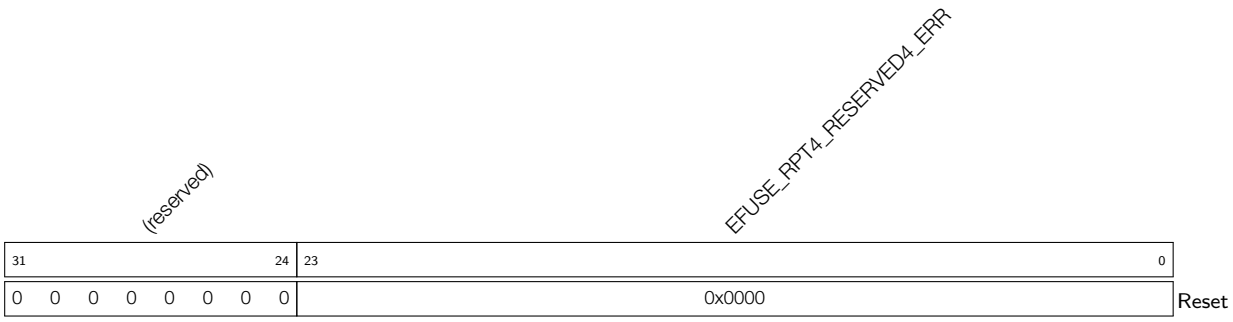
EFUSE_FLASH_TYPE_ERR Any bit equal to 1 denotes a programming error in EFUSE_FLASH_TYPE. (RO)

EFUSE_FORCE_SEND_RESUME_ERR Any bit equal to 1 denotes a programming error in EFUSE_FORCE_SEND_RESUME. (RO)

EFUSE_SECURE_VERSION_ERR Any bit equal to 1 denotes a programming error in EFUSE_SECURE_VERSION. (RO)

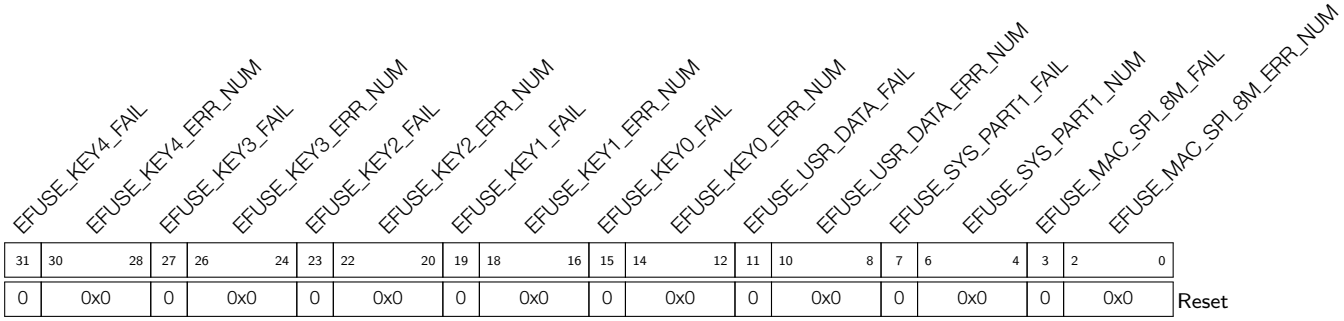
EFUSE_RPT4_RESERVED2_ERR Any bit equal to 1 denotes a programming error in EFUSE_RPT4_RESERVED2. (RO)

Register 11.28: EFUSE_RD_REPEAT_ERR4_REG (0x0190)



EFUSE_RPT4_RESERVED4_ERR If any bit in RPT4_RESERVED4 is 1, there is a programming error in EFUSE_RPT4_RESERVED4. (RO)

Register 11.29: EFUSE_RD_RS_ERR0_REG (0x01C0)



EFUSE_MAC_SPI_8M_ERR_NUM The value of this signal means the number of error bytes in BLOCK1. (RO)

EFUSE_MAC_SPI_8M_FAIL 0: Means no failure and that the data of BLOCK1 is reliable; 1: Means that programming BLOCK1 data failed and the number of error bytes is over 5. (RO)

EFUSE_SYS_PART1_NUM The value of this signal means the number of error bytes in BLOCK2. (RO)

EFUSE_SYS_PART1_FAIL 0: Means no failure and that the data of BLOCK2 is reliable; 1: Means that programming BLOCK2 data failed and the number of error bytes is over 5. (RO)

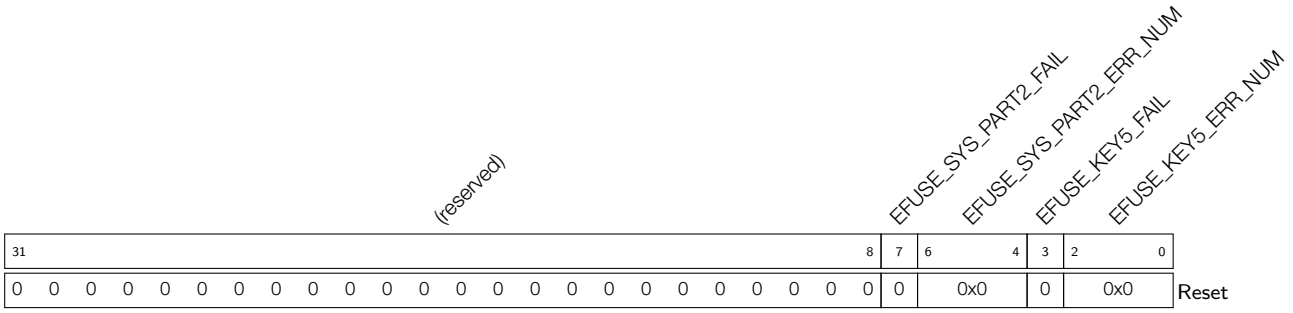
EFUSE_USR_DATA_ERR_NUM The value of this signal means the number of error bytes in BLOCK3. (RO)

EFUSE_USR_DATA_FAIL 0: Means no failure and that the data of BLOCK3 is reliable; 1: Means that programming BLOCK3 data failed and the number of error bytes is over 5. (RO)

EFUSE_KEY*n*_ERR_NUM The value of this signal means the number of error bytes in KEY*n*. (RO)

EFUSE_KEY*n*_FAIL 0: Means no failure and that the data of KEY*n* is reliable; 1: Means that programming KEY*n* failed and the number of error bytes is over 5. (RO)

Register 11.30: EFUSE_RD_RS_ERR1_REG (0x01C4)



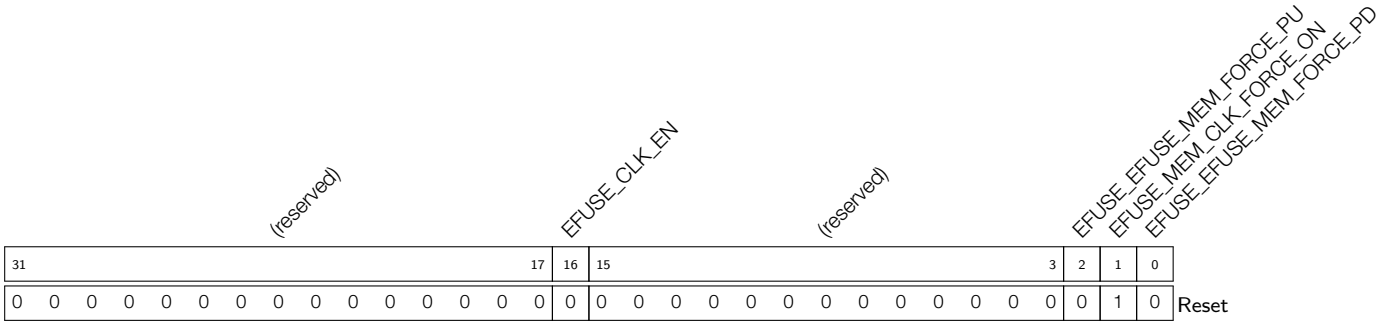
EFUSE_KEY5_ERR_NUM The value of this signal means the number of error bytes in KEY5. (RO)

EFUSE_KEY5_FAIL 0: Means no failure and that the data of KEY5 is reliable; 1: Means that programming user data failed and the number of error bytes is over 5. (RO)

EFUSE_SYS_PART2_ERR_NUM The value of this signal means the number of error bytes in BLOCK10. (RO)

EFUSE_SYS_PART2_FAIL 0: Means no failure and that the data of BLOCK10 is reliable; 1: Means that programming BLOCK10 data failed and the number of error bytes is over 5. (RO)

Register 11.31: EFUSE_CLK_REG (0x01C8)



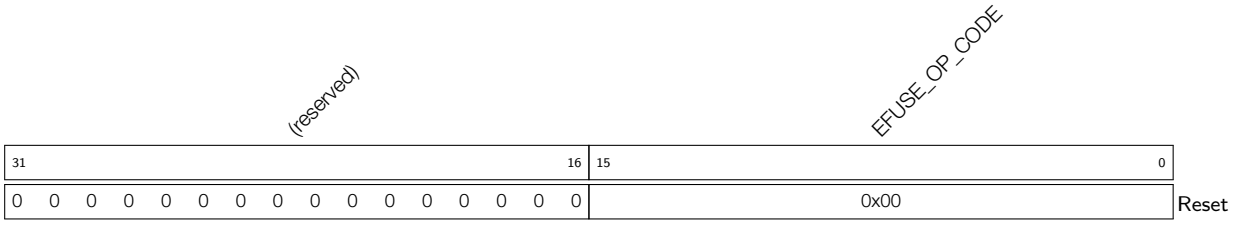
EFUSE_EFUSE_MEM_FORCE_PD If set, forces eFuse SRAM into power-saving mode. (R/W)

EFUSE_MEM_CLK_FORCE_ON If set, forces to activate clock signal of eFuse SRAM. (R/W)

EFUSE_EFUSE_MEM_FORCE_PU If set, forces eFuse SRAM into working mode. (R/W)

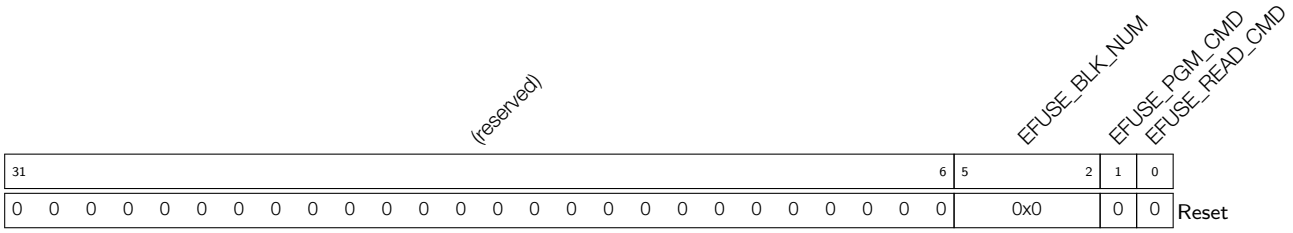
EFUSE_CLK_EN If set, forces to enable clock signal of eFuse memory. (R/W)

Register 11.32: EFUSE_CONF_REG (0x01CC)



EFUSE_OP_CODE 0x5A5A: Operate programming command; 0x5AA5: Operate read command. (R/W)

Register 11.33: EFUSE_CMD_REG (0x01D4)

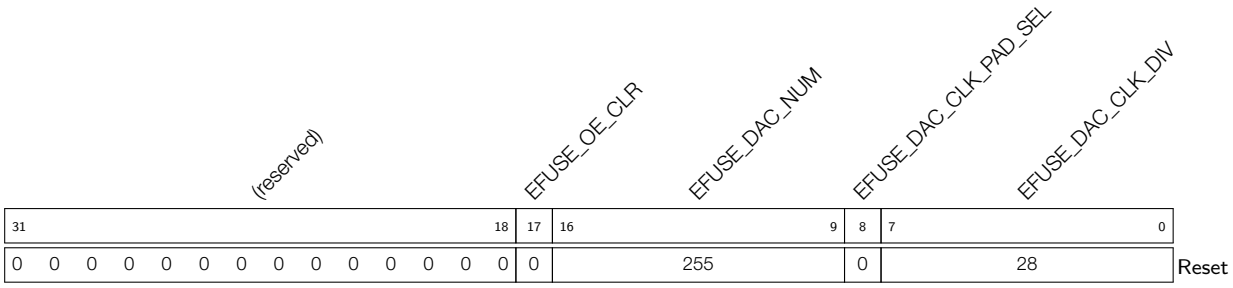


EFUSE_READ_CMD Set this bit to send read command. (R/W)

EFUSE_PGM_CMD Set this bit to send programming command. (R/W)

EFUSE_BLK_NUM The serial number of the block to be programmed. Value 0-10 corresponds to block number 0-10, respectively. (R/W)

Register 11.34: EFUSE_DAC_CONF_REG (0x01E8)



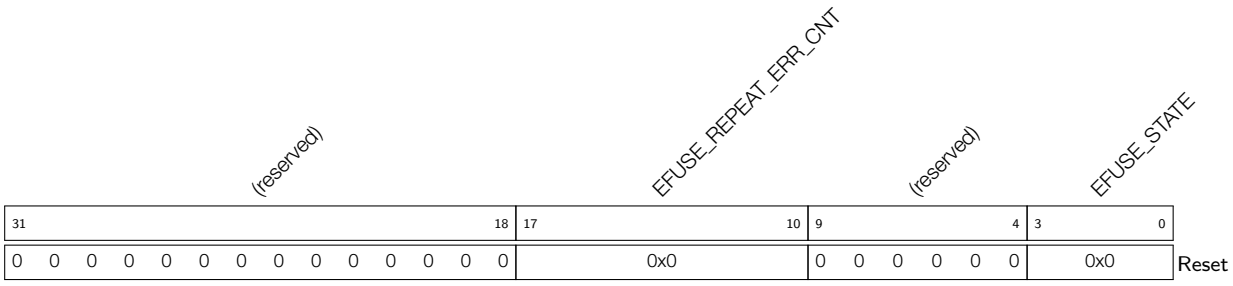
EFUSE_DAC_CLK_DIV Controls the division factor of the rising clock of the programming voltage. (R/W)

EFUSE_DAC_CLK_PAD_SEL Don't care. (R/W)

EFUSE_DAC_NUM Controls the rising period of the programming voltage. (R/W)

EFUSE_OE_CLR Reduces the power supply of the programming voltage. (R/W)

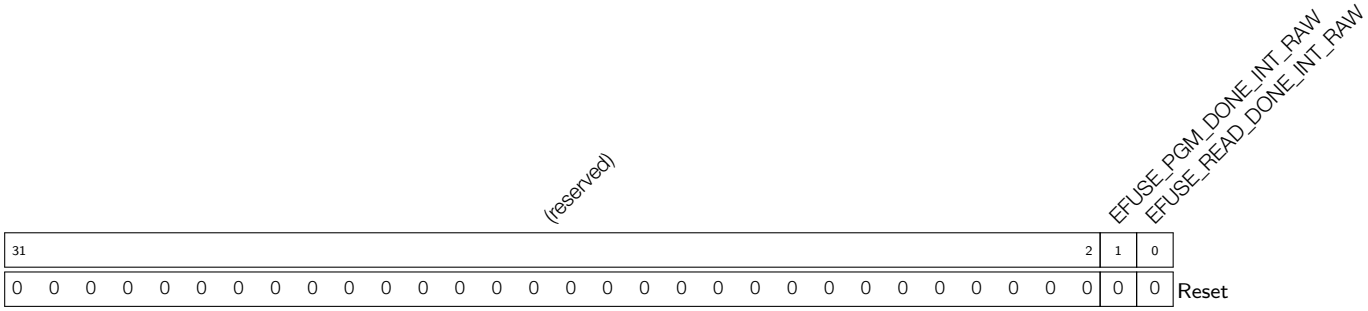
Register 11.35: EFUSE_STATUS_REG (0x01D0)



EFUSE_STATE Indicates the state of the eFuse state machine. (RO)

EFUSE_REPEAT_ERR_CNT Indicates the number of error bits during programming BLOCK0. (RO)

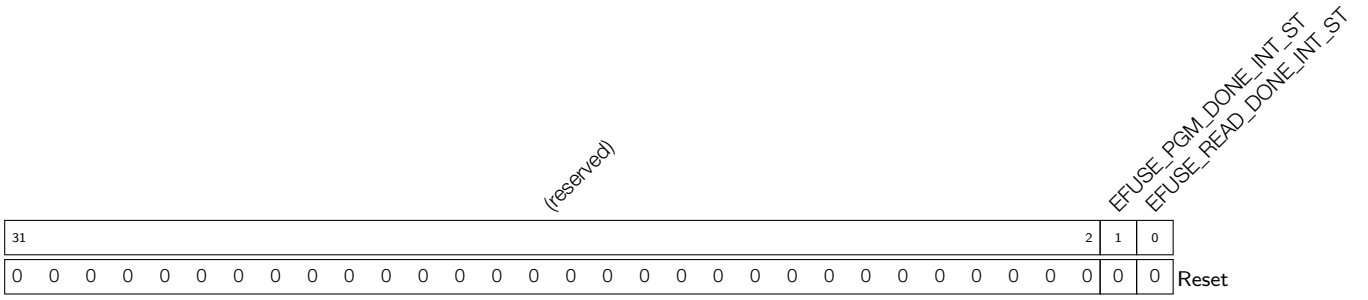
Register 11.36: EFUSE_INT_RAW_REG (0x01D8)



EFUSE_READ_DONE_INT_RAW The raw bit signal for read_done interrupt. (RO)

EFUSE_PGM_DONE_INT_RAW The raw bit signal for pgm_done interrupt. (RO)

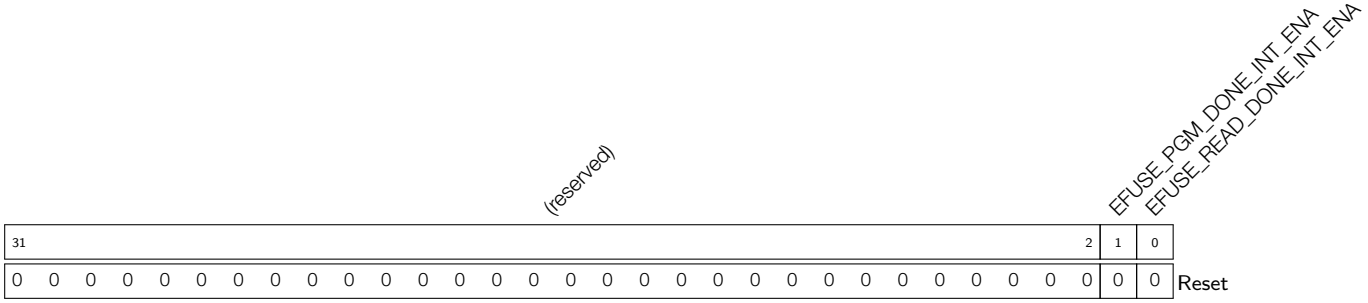
Register 11.37: EFUSE_INT_ST_REG (0x01DC)



EFUSE_READ_DONE_INT_ST The status signal for read_done interrupt. (RO)

EFUSE_PGM_DONE_INT_ST The status signal for pgm_done interrupt. (RO)

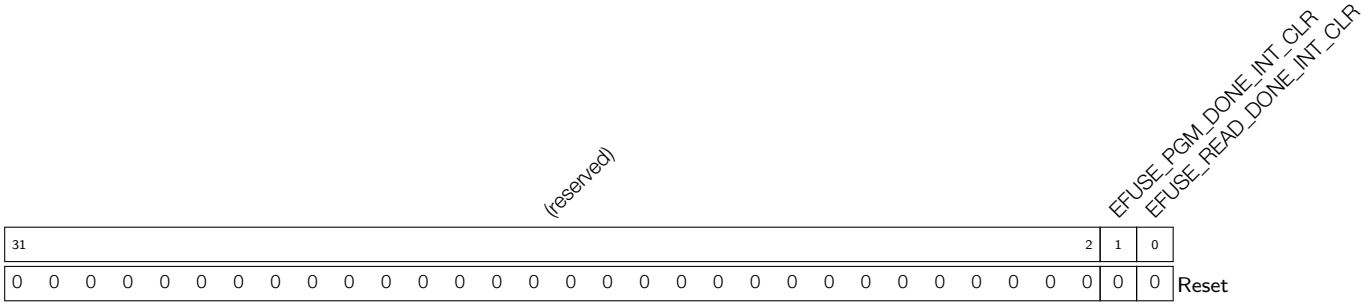
Register 11.38: EFUSE_INT_ENA_REG (0x01E0)



EFUSE_READ_DONE_INT_ENA The enable signal for read_done interrupt. (R/W)

EFUSE_PGM_DONE_INT_ENA The enable signal for pgm_done interrupt. (R/W)

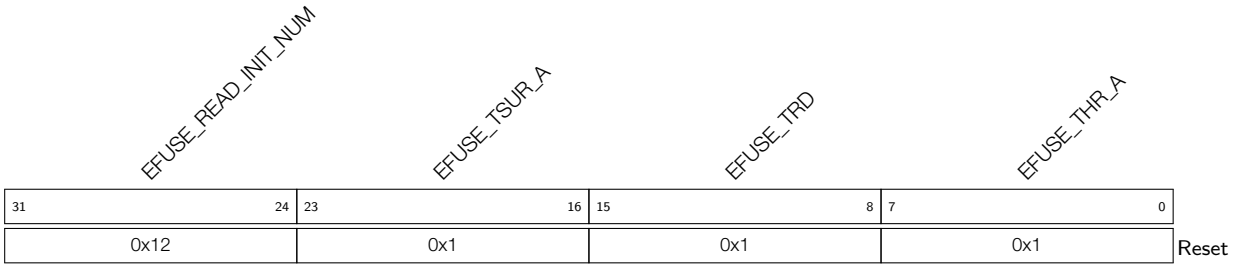
Register 11.39: EFUSE_INT_CLR_REG (0x01E4)



EFUSE_READ_DONE_INT_CLR The clear signal for read_done interrupt. (WO)

EFUSE_PGM_DONE_INT_CLR The clear signal for pgm_done interrupt. (WO)

Register 11.40: EFUSE_RD_TIM_CONF_REG (0x01EC)



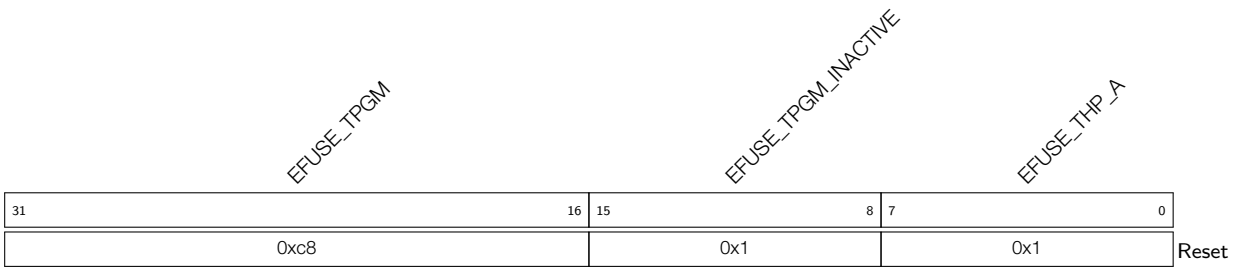
EFUSE_THR_A Configures the hold time of read operation. (R/W)

EFUSE_TRD Configures the length of pulse of read operation. (R/W)

EFUSE_TSUR_A Configures the setup time of read operation. (R/W)

EFUSE_READ_INIT_NUM Configures the initial read time of eFuse. (R/W)

Register 11.41: EFUSE_WR_TIM_CONF0_REG (0x01F0)

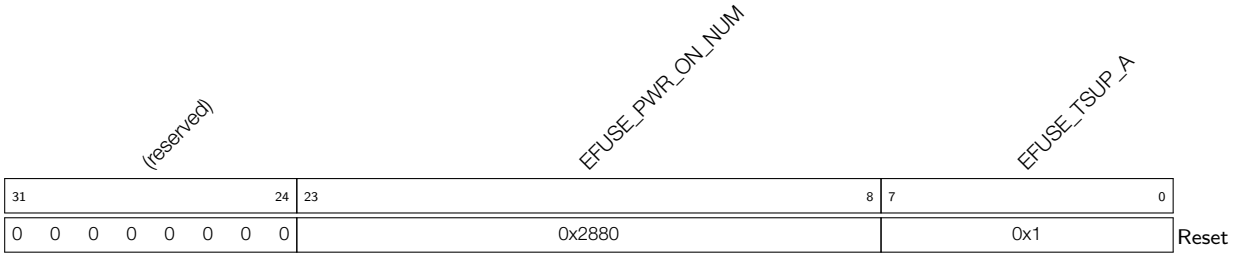


EFUSE_THP_A Configures the hold time of programming operation. (R/W)

EFUSE_TPGM_INACTIVE Configures the length of pulse during programming 0 to eFuse. (R/W)

EFUSE_TPGM Configures the length of pulse during programming 1 to eFuse. (R/W)

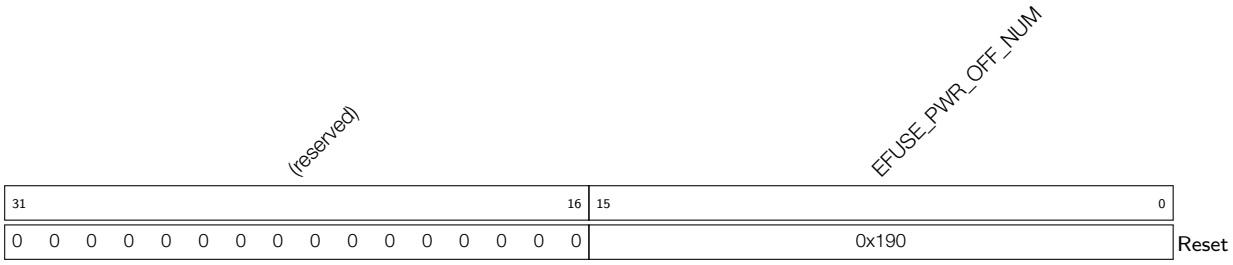
Register 11.42: EFUSE_WR_TIM_CONF1_REG (0x01F4)



EFUSE_TSUP_A Configures the setup time of programming operation. (R/W)

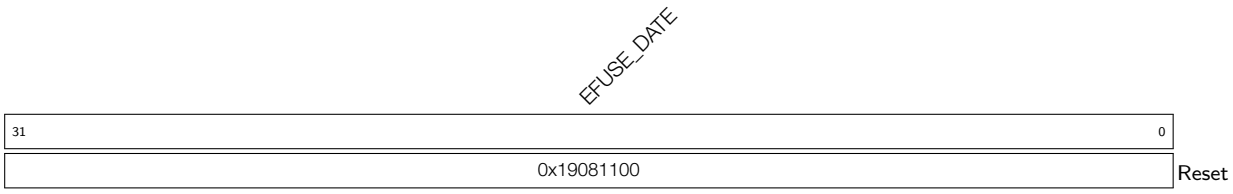
EFUSE_PWR_ON_NUM Configures the power up time for VDDQ. (R/W)

Register 11.43: EFUSE_WR_TIM_CONF2_REG (0x01F8)



EFUSE_PWR_OFF_NUM Configures the power outage time for VDDQ. (R/W)

Register 11.44: EFUSE_DATE_REG (0x01FC)



EFUSE_DATE Version control register. (R/W)

12. I²C Controller

12.1 Overview

The I²C (Inter-Integrated Circuit) controller allows ESP32-S2 to communicate with multiple peripheral devices. These peripheral devices can share one bus.

12.2 Features

The I²C controller has the following features:

- Master mode and slave mode
- Multi-master and multi-slave communication
- Standard mode (100 kbit/s)
- Fast mode (400 kbit/s)
- 7-bit addressing and 10-bit addressing
- Continuous data transfer in master mode achieved by pulling SCL low
- Programmable digital noise filtering
- Double addressing mode

12.3 I²C Functional Description

12.3.1 I²C Introduction

The I²C bus has two lines, namely a serial data line (SDA) and a serial clock line (SCL). Both SDA and SCL lines are open-drain. The I²C bus is connected to multiple devices, usually a single or multiple masters and a single or multiple slaves. However, only one master device can access a slave at a time.

The master initiates communication by generating a start condition: pulling the SDA line l ow while SCL is high, and sending nine clock pulses via SCL. The first eight pulses are used to transmits a byte, which consists of a 7-bit address followed by a read/write ($R/\overline{W}$) bit. If the address of a slave matches the 7-bit address transmitted, this matching slave can respond by pulling SDA low on the ninth clock pulse. The master and the slave can send or receive data according to the $R/\overline{W}$ bit. Whether to terminate the data transfer or not is determined by the logic level of the acknowledge (ACK) bit. During data transfer, SDA changes only when SCL is low. Once finishing communication, the master sends a STOP condition: pulling SDA up while SCL is high. If a master both reads and writes data in one transfer, then it should send a RESTART condition, a slave address and a $R/\overline{W}$ bit before changing its operation.

12.3.2 I²C Architecture

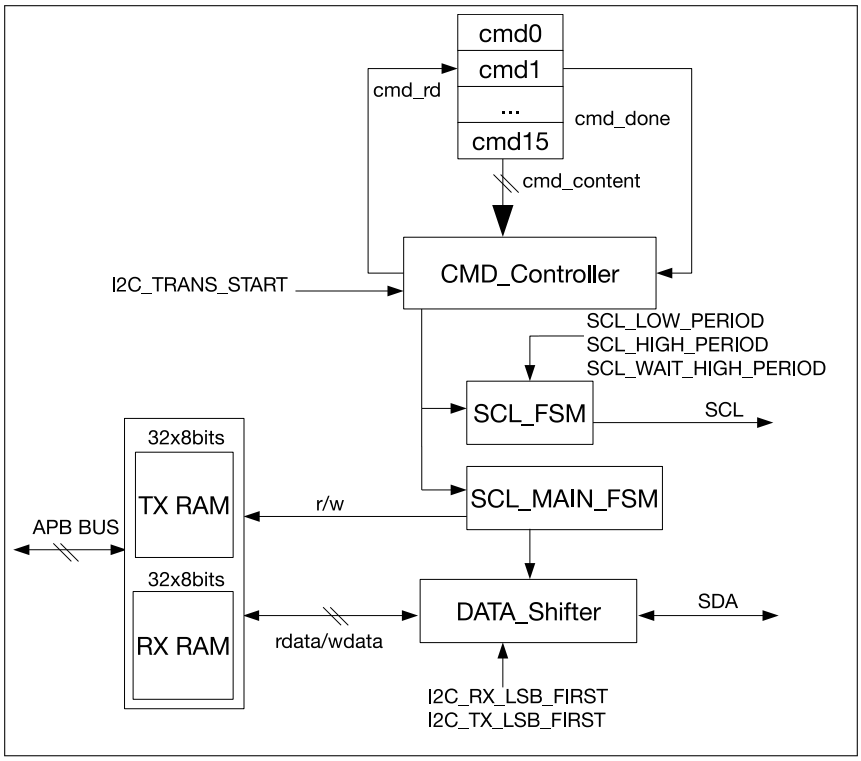


Figure 12-1. I²C Master Architecture

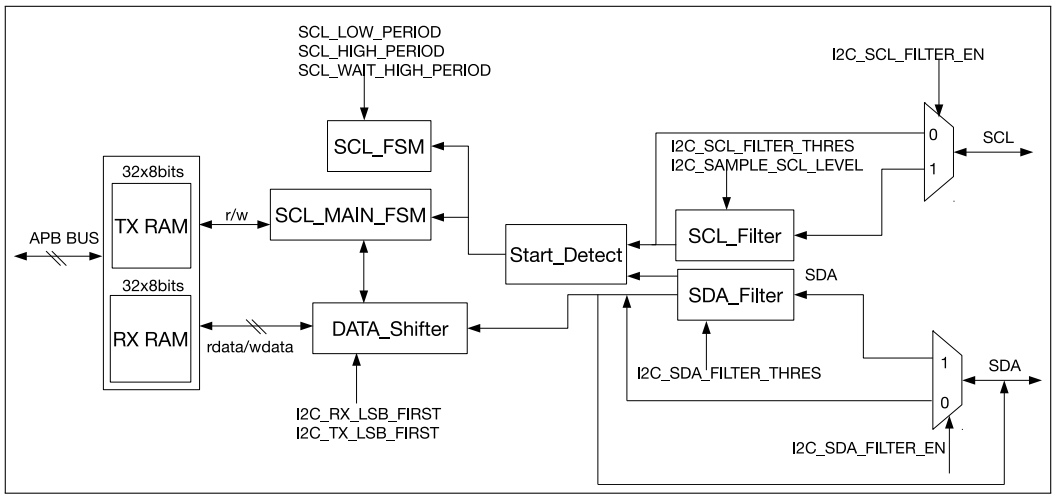


Figure 12-2. I²C Slave Architecture

The I²C controller runs either in master mode or slave mode, which is determined by I2C_MS_MODE bit. Figure 12-1 shows the architecture of a master, while Figure 12-2 shows that of a slave. The I²C controller has the following main parts: transmit and receive memory (TX/RX RAM), command controller (CMD_Controller), SCL clock controller (SCL_FSM), SDA data controller (SCL_MAIN_FSM), serial-to-parallel data converter (DATA_Shifter), filter for SCL (SCL_Filter) and filter for SDA (SDA_Filter).

12.3.2.1 TX/RX RAM

Both TX RAM and RX RAM are 32 × 8 bits. TX RAM stores data that the I²C controller needs to send. During communication, when the I²C controller needs to send data (except acknowledgement bits), it reads data from TX RAM and sends it sequentially via SDA. When the I²C controller works in master mode, all data must be stored in TX RAM in the order they will be sent to slaves. The data stored in TX RAM include slave addresses, read/write bits, register addresses (only in dual addressing mode) and data to be sent. When the I²C controller works in slave mode, TX RAM only stores data to be sent.

RX RAM stores data the I²C controller receives during communication. When the I²C controller works in slave mode, neither slave addresses sent by the master nor register addresses (only in double addressing mode) will be stored into RX RAM. Values of RX RAM can be read by software after I²C communication completes.

Both TX RAM and RX RAM can be accessed in FIFO or non-FIFO mode. The I2C_NONFIFO_EN bit sets FIFO or non-FIFO mode.

TX RAM can be read and written by the CPU. The CPU writes to TX RAM either in FIFO mode or in non-FIFO mode (direct address). In FIFO mode, the CPU writes to TX RAM via fixed address I2C_DATA_REG, with addresses for writing in TX RAM incremented automatically by hardware. In non-FIFO mode, the CPU accesses TX RAM directly via address fields (I²C Base Address + 0x100) ~(I²C Base Address + 0x17C). Each byte in TX RAM occupies an entire word in the address space. Therefore, the address of the first byte is I²C Base Address + 0x100, the second byte I²C Base Address + 0x104, the third byte I²C Base Address + 0x108, and so on. The CPU can only read TX RAM via direct addresses. Unlike addresses for writing, TX RAM must be read back from addresses starting at I²C Base Address + 0x80.

RX RAM can only be read by the CPU. The CPU reads RX RAM either in FIFO mode or in non-FIFO mode (direct address). In FIFO mode, the CPU reads RX RAM via fixed address I2C_DATA_REG, with addresses for reading RX RAM incremented automatically by hardware. In non-FIFO mode, the CPU accesses TX RAM directly via address fields (I²C Base Address + 0x100) ~(I²C Base Address + 0x17C). Each byte in RX RAM occupies an entire word in the address space. Therefore, the address of the first byte is I²C Base Address + 0x100, the second byte I²C Base Address + 0x104, the third byte I²C Base Address + 0x108 and so on.

Given that addresses for writing to TX RAM have an identical range with those for reading RX RAM, TX RAM and RX RAM can be seen as a 32 x 8 RAM. In following sections TX RAM and RX RAM are referred to as RAM.

12.3.2.2 CMD_Controller

When the I²C controller works in master mode, the integrated CMD_Controller module reads commands from 16 sequential command registers and controls SCL_FSM and SDA_FSM accordingly.

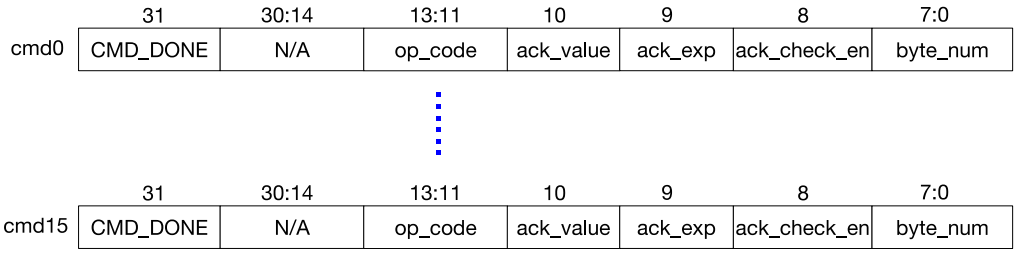


Figure 12-3. Structure of I²C Command Register

Command registers, whose structure is illustrated in Figure 12-3, are active only when the I²C controller works in master mode. Fields of command registers are:

1. CMD_DONE: Indicates that a command has been executed. After each command has been executed, the corresponding CMD_DONE bit is set to 1 by hardware. By reading this bit, software can tell if the command has been executed. When writing new commands, this bit must be cleared by software.
2. op_code: Indicates the command. The I²C controller supports five commands:
   - RSTART: op_code = 0: The I²C controller sends a START bit and a RESTART bit defined by the I²C protocol.
   - WRITE: op_code = 1: The I²C controller sends a slave address, a register address (only in double addressing mode) and data to the slave.
   - READ: op_code = 2: The I²C controller reads data from the slave.
   - STOP: op_code = 3: The I²C controller sends a STOP bit defined by the I²C protocol. This code also indicates that the command sequence has been executed, and the CMD_Controller stops reading commands. After restarted by software, the CMD_Controller resumes reading commands from command register 0.
   - END: op_code = 4: The I²C controller pulls the SCL pin down and suspends I²C communication. This code also indicates that the command sequence has completed, and the CMD_Controller stops executing commands. Once software refreshes data in command registers and the RAM, the CMD_Controller can be restarted to execute commands from command register 0 again.
3. ack_value: Used to configure the level of the ACK bit sent by the I²C controller during a read operation. This bit is ignored during RSTART, STOP, END and WRITE conditions.
4. ack_exp: Used to configure the level of the ACK bit expected by the I²C controller during a write operation. This bit is ignored during RSTART, STOP, END and READ conditions.
5. ack_check_en: Used to enable the I²C controller during a write operation to check whether ACK's level sent by the slave matches ack_exp in the command. If this bit is set and the level received does not match ack_exp in the WRITE command, the master will generate an I2C_NACK_INT interrupt and a STOP condition for data transfer. If this bit is cleared, the controller will not check the ACK level sent by the slave. This bit is ignored during RSTART, STOP, END and READ conditions.
6. byte_num: Specifies the length of data (in bytes) to be read or written. Can range from 1 to 255 bytes. This bit is ignored during RSTART, STOP and END conditions.

Each command sequence is executed starting from command register 0 and terminated by a STOP or an END. Therefore, all 16 command registers must have a STOP or an END command.

A complete data transfer on the I²C bus should be initiated by a START and terminated by a STOP. The transfer process may be completed using multiple sequences, separated by END commands. Each sequence may differ in the direction of data transfer, clock frequency, slave addresses, data length, data to be transmitted, etc. This allows efficient use of available peripheral RAM and also achieves more flexible I²C communication.

12.3.2.3 SCL_FSM

The integrated SCL_FSM module controls the SCL clock line. The frequency and duty cycle of SCL is configured using I2C_SCL_LOW_PERIOD_REG, I2C_SCL_HIGH_PERIOD_REG and I2C_SCL_WAIT_HIGH_PERIOD. After being in non-IDLE state for over I2C_SCL_ST_TO clock cycles, SCL_FSM triggers an I2C_SCL_ST_TO_INT interrupt and returns to IDLE state.

12.3.2.4 SCL_MAIN_FSM

The integrated SCL_MAIN_FSM module controls the SDA data line and data storage. After being in non-IDLE state for over I2C_SCL_MAIN_ST_TO clock cycles, SCL_MAIN_FSM triggers an I2C_SCL_MAIN_ST_TO_INT interrupt and returns to IDLE state.

12.3.2.5 DATA_Shifter

The integrated DATA_Shifter module is used for serial/parallel conversion, converting TX RAM byte data to an outgoing serial bitstream or an incoming serial bitstream to RX RAM byte data. I2C_RX_LSB_FIRST and I2C_TX_LSB_FIRST can be used to select LSB- or MSB-first storage and transmission of data.

12.3.2.6 SCL_Filter and SDA_Filter

The integrated SCL_Filter and SDA_Filter modules are identical and are used to filter signal noises on SCL and SDA, respectively. These filters can be enabled or disabled by configuring I2C_SCL_FILTER_EN and I2C_SDA_FILTER_EN.

SCL_Filter samples input signals on the SCL line continuously. These input signals are valid only if they remain unchanged for consecutive I2C_SCL_FILTER_THRES clock cycles. Given that only valid input signals can pass through the filter, SCL_Filter can remove glitches whose pulse width is lower than I2C_SCL_FILTER_THRES APB clock cycles.

SDA_Filter is identical to SCL_Filter, only applied to the SDA line. The threshold pulse width is provided in the I2C_SDA_FILTER_THRES register.

12.3.3 I²C Bus Timing

The I²C controller may use APB_CLK or REF_TICK as its clock source. When I2C_REF_ALWAYS_ON is 1, APB_CLK is used; when I2C_REF_ALWAYS_ON is 0, REF_TICK is used.

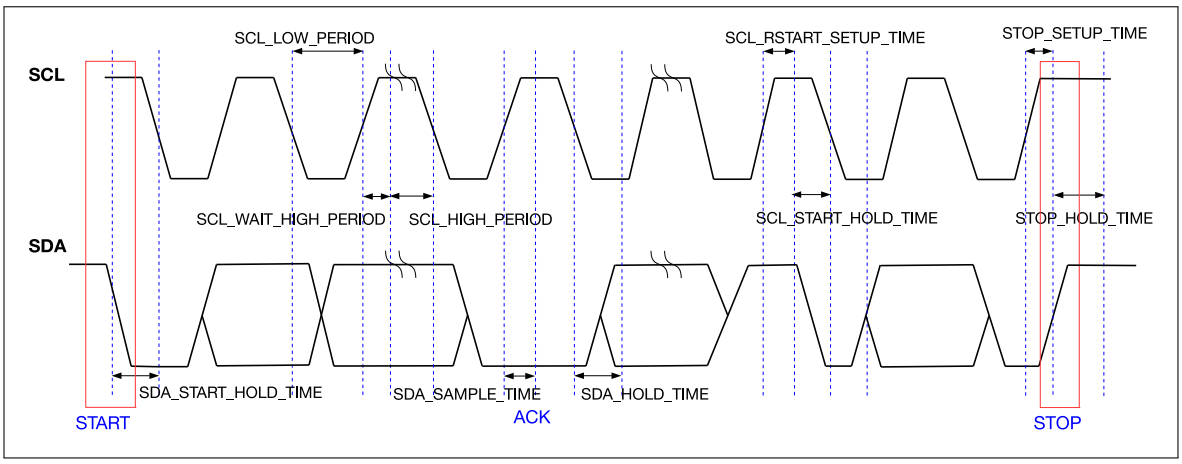


Figure 12-4. I²C Timing Diagram

Figure 12-4 shows the timing diagram of an I²C master. The unit of parameters in this figure is I2C_CLK (T_{I2C_CLK}). Specifically, when I2C_REF_ALWAYS_ON = 1, T_{I2C_CLK} is T_{APB_CLK}; when I2C_REF_ALWAYS_ON = 0, T_{I2C_CLK} is T_{REF_TICK}. Figure 12-4 also specifies registers used to configure the START bit, STOP bit, data hold time, data sample time, waiting time on the rising SCL edge, etc. Parameters in Figure 12-4 are described as the following:

1. I2C_SCL_START_HOLD_TIME: Specifies the interval between pulling SDA low and pulling SCL low when the master generates a START condition. This interval is (I2C_SCL_START_HOLD_TIME +1) × T_{I2C_CLK}. This register is active only when the I²C controller works in master mode.
2. I2C_SCL_LOW_PERIOD: Specifies the low period of SCL. This period lasts (I2C_SCL_START_HOLD_TIME +1) × T_{I2C_CLK}. However, it could be extended when SCL is pulled low by peripheral devices or by an END command executed by the I²C controller, or when the clock is stretched. This register is active only when the I²C controller works in master mode.
3. I2C_SCL_WAIT_HIGH_PERIOD: Specifies time for SCL to go high in T_{I2C_CLK}. Please make sure that SCL will be pulled high within this time period. Otherwise, the high period of SCL may be incorrect. This register is active only when the I²C controller works in master mode.
4. I2C_SCL_HIGH_PERIOD: Specifies the high period of SCL in T_{I2C_CLK}. This register is active only when the I²C controller works in master mode. When SCL goes high within (I2C_SCL_WAIT_HIGH_PERIOD + 1) × T_{I2C_CLK}, its frequency is:

$$f_{scl} = \frac{f_{\text{I2C\_CLK}}}{\text{I2C\_SCL\_LOW\_PERIOD} + 1 + \text{I2C\_SCL\_HIGH\_PERIOD} + \text{I2C\_SCL\_WAIT\_HIGH\_PERIOD}}$$

5. I2C_SDA_SAMPLE_TIME: Specifies the interval between the rising edge of SCL and the level sampling time of SDA. It is advised to set a value in the middle of SCL's high period. This register is active both in master mode and slave mode.
6. I2C_SDA_HOLD_TIME: Specifies the interval between changing the SDA output level and the falling edge of SCL. This register is active both in master mode and slave mode.

SCL and SDA output drivers must be configured as open drain. There are two ways to achieve this:

1. Set I2C_SCL_FORCE_OUT and I2C_SDA_FORCE_OUT, and configure GPIO_PIN*n*_PAD_DRIVER register for corresponding SCL and SDA pads as open-drain.

2. Clear I2C_SCL_FORCE_OUT and I2C_SDA_FORCE_OUT.

Because these lines are configured as open-drain, the low to high transition time of each line is determined together by the pull-up resistance and the capacitance on the line. The output frequency of I²C is limited by the SDA and SCL line's pull-up speed, mainly SCL's speed.

In addition, when I2C_SCL_FORCE_OUT and I2C_SCL_PD_EN are set to 1, SCL can be forced low; when I2C_SDA_FORCE_OUT and I2C_SDA_PD_EN are set to 1, SDA can be forced low.

12.4 Typical Applications

For the convenience of description, I²C masters and slaves in all subsequent figures refer to ESP32-S2 I²C peripheral controllers. However, these controllers can communicate with any other I²C devices.

12.4.1 An I²C Master Writes to an I²C Slave with a 7-bit Address in One Command Sequence

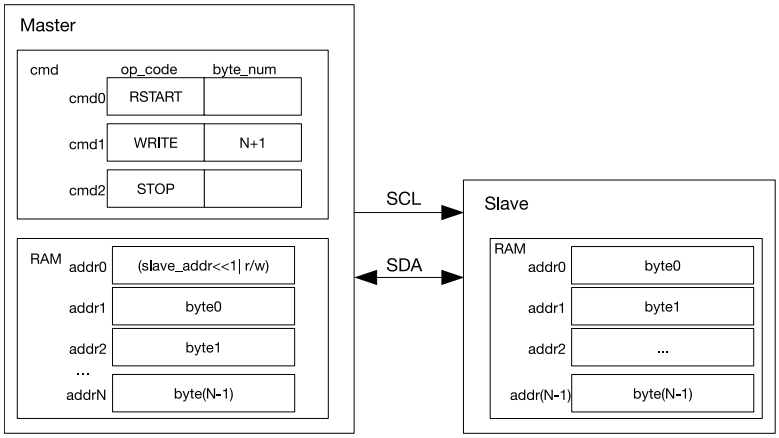


Figure 12-5. An I²C Master Writing to an I²C Slave with a 7-bit Address

Figure 12-5 shows how an I²C master writes N bytes of data using 7-bit addressing. As shown in figure 12-5, the first byte in the master's RAM is a 7-bit slave address followed by a $R/\overline{W}$ bit. When the $R/\overline{W}$ bit is zero, it indicates a WRITE operation. The remaining bytes are used to store data ready for transfer. The cmd box contains related command sequences.

After the command sequence is configured and data in RAM is ready, the master enables the controller and initiates data transfer by setting I2C_TRANS_START bit. The controller has four steps to take:

1. Wait for SCL to go high, to avoid SCL used by other masters or slaves.
2. Execute a RSTART command and send a START bit.
3. Execute a WRITE command by taking N+1 bytes from the RAM in order and send them to the slave in the same order. The first byte is the address of the slave.
4. Send a STOP. Once the I²C master transfers a STOP bit, an I2C_TRANS_COMPLETE_INT interrupt is generated.

If data to be transferred is larger than 32 bytes, the RAM access will wrap around. While the controller sends data, software must replace data already sent in RAM. To assist with this process, the master will generate an I2C_TXFIFO_WM_INT interrupt when less than I2C_TXFIFO_WM_THRHD bytes of data remain to be sent.

After detecting this interrupt, software can refresh data in RAM. When the RAM is accessed in non-FIFO mode, in order to overwrite existing data in the RAM with new ones, software needs to first configure I2C_TX_UPDATE bit to latch the start address and the end address of data sent in the RAM, and then read I2C_TXFIFO_START_ADDR and I2C_TXFIFO_END_ADDR field in I2C_FIFO_ST_REG register to obtain these addresses. When the RAM is accessed in FIFO mode, new data can be written to the RAM directly via I2C_DATA_REG register.

The controller stops executing the command sequence after a STOP command, or when one of the following two events occurs:

1. When ack_check_en is set to 1, the I²C master checks the ACK value each time it sends a data byte. If the ACK value received does not match ack_exp (the expected ACK value) in the WRITE command, the master generates an I2C_NACK_INT interrupt and stops the transmission.
2. During the high period of SCL, if the input value and the output value of SDA do not match, the I²C master generates an I2C_ARBITRATION_LOST_INT interrupt, stops executing the command sequence, returns to IDLE state and releases SCL and SDA.

Once detecting a START bit sent by the I²C master, the I²C slave receives the address and compares it with its own address. If the received address does not match I2C_SLAVE_ADDR[6:0], the slave stops receiving data. If the received address does match I2C_SLAVE_ADDR[6:0], the slave receives data and stores them into the RAM in order.

If data to be transferred is larger than 32 bytes, the RAM may wrap around. While the controller receives data, software reclaim data already received by the slave. To assist with this process, the master will generate an I2C_RXFIFO_WM_INT after I2C_RXFIFO_WM_THRHD bytes are received in RAM.

Once detecting this interrupt, software can read data from the RAM registers. When the RAM is accessed in non-FIFO mode, in order to read data, software needs to configure I2C_RX_UPDATE bit to latch the start address and the end address of data to be reclaimed, and read I2C_RXFIFO_START_ADDR and I2C_RXFIFO_END_ADDR fields in I2C_FIFO_ST_REG register to obtain these addresses. When the RAM is accessed in FIFO mode, data can be read directly via I2C_DATA_REG register.

12.4.2 An I²C Master Writes to an I²C Slave with a 10-bit Address in One Command Sequence

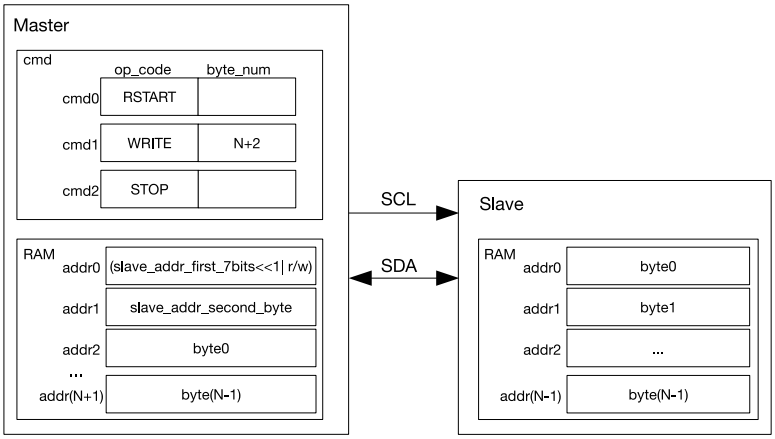


Figure 12-6. A Master Writing to a Slave with a 10-bit Address

Besides 7-bit addressing (SLV_ADDR[6:0]), the ESP32-S2 I²C controller also supports 10-bit addressing (SLV_ADDR[9:0]). In the following text, the slave address is referred to as SLV_ADDR.

Figure 12-6 shows how an I²C master writes N-bytes of data using 10-bit addressing. Unlike a 7-bit address, a 10-bit slave address is formed from two bytes. In master mode, the first byte of the slave address, which comprises slave_addr_first_7bits and a $R/\overline{W}$ bit, is stored into addr0 in the RAM. slave_addr_first_7bits should be configured as (0x78 | SLV_ADDR[9:8]). The second byte slave_addr_second_byte is stored into addr1 in the RAM. slave_addr_second_byte should be configured as SLV_ADDR[7:0].

In the slave, the 10-bit addressing mode is enabled by configuring I2C_ADDR_10BIT_EN bit. The address of the I²C slave is configured using I2C_SLAVE_ADDR. I2C_SLAVE_ADDR[14:7] should be configured as SLV_ADDR[7:0], and I2C_SLAVE_ADDR[6:0] should be configured as (0x78 | SLV_ADDR[9:8]). Since a 10-bit slave address has one more byte than a 7-bit address, byte_num of the WRITE command and the number of bytes in the RAM increase by one.

12.4.3 An I²C Master Writes to an I²C Slave with Two 7-bit Addresses in One Command Sequence

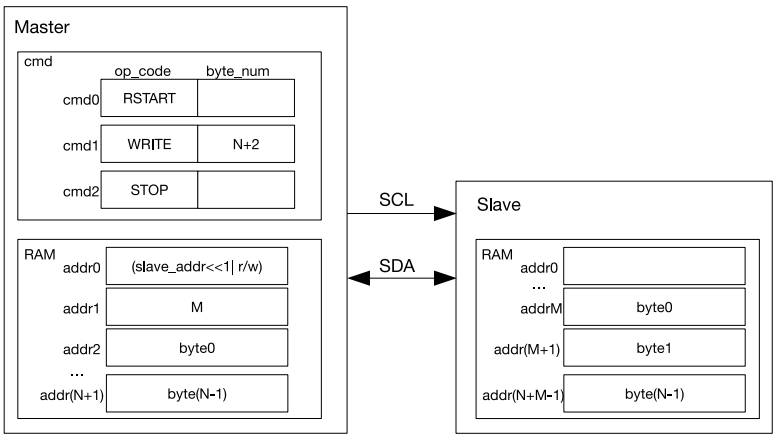


Figure 12-7. An I²C Master Writing Address M in the RAM to an I²C Slave with a 7-bit Address

When working in slave mode, the controller supports double addressing, where the first address is the address of an I²C slave, and the second one is the slave's memory address. Double addressing is enabled by configuring I2C_FIFO_ADDR_CFG_EN. When using double addressing, RAM must be accessed in non-FIFO mode. As figure 12-7 illustrates, the I²C slave put received byte0 ~ byte(N-1) into its RAM in an order staring from addrM. The RAM is overwritten every 32 bytes.

12.4.4 An I²C Master Writes to an I²C Slave with a 7-bit Address in Multiple Command Sequences

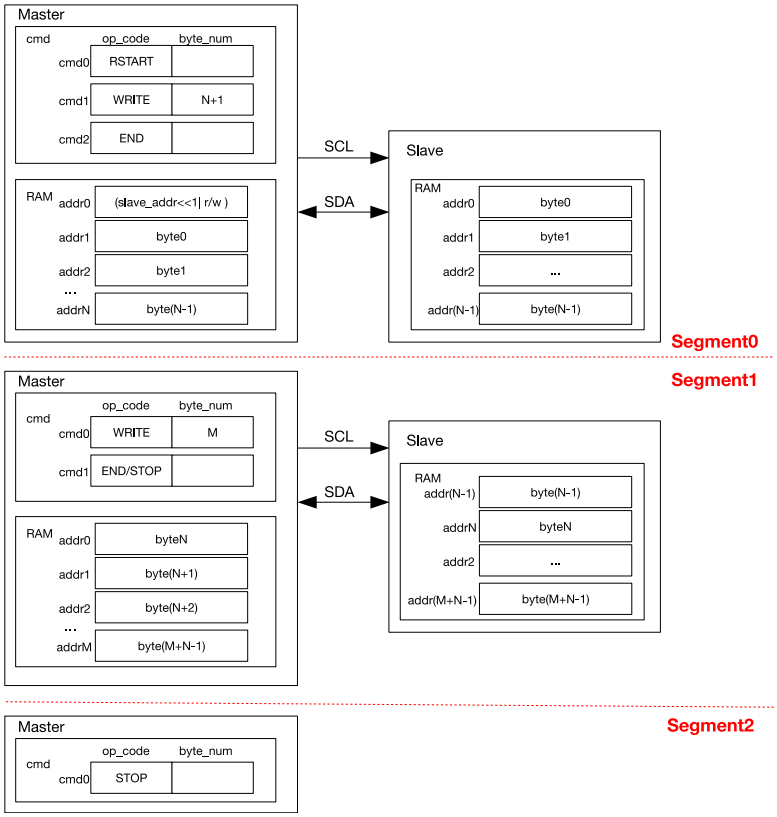


Figure 12-8. An I²C Master Writing to an I²C Slave with a 7-bit Address in Multiple Sequences

Given that the I²C Controller RAM holds only 32 bytes, when data are too large to be processed even by the wrapped RAM, it is advised to transmit them in multiple command sequences. At the end of every command sequence is an END command. When the controller executes this END command to pull SCL low, software refreshes command sequence registers and the RAM for next the transmission.

Figure 12-8 shows how an I²C master writes to an I²C slave in two or three segments. For the first segment, the CMD_Controller registers are configured as shown in Segment0. Once data in the master's RAM is ready and I2C_TRANS_START is set, the master initiates data transfer. After executing the END command, the master turns off the SCL clock and pulls the SCL low to reserve the bus. Meanwhile, the controller generates an I2C_END_DETECT_INT interrupt.

For the second segment, after detecting the I2C_END_DETECT_INT interrupt, software refreshes the CMD_Controller registers and reloads the RAM and clears this interrupt, as shown in Segment1. If cmd1 in the second segment is a STOP, then data is transmitted to the slave in two segments. The master resumes data transfer after I2C_TRANS_START is set, and terminates the transfer by sending a STOP bit.

For the third segment, after the second data transfer finishes and an I2C_END_DETECT_INT is detected, the CMD_Controller registers of the master are configured as shown in Segment2. Once I2C_TRANS_START is set, the master generates a STOP bit and terminates the transfer.

Please note that other I²C masters will not transact on the bus between two segments. The bus is only released after a STOP signal is sent. To interrupt the transfer, the I²C controller can be reset by setting I2C_FSM_RST at any time. This register will later be cleared automatically by hardware.

When the master is in IDLE state and I2C_SCL_RST_SLV_EN is set, hardware sends I2C_SCL_RST_SLV_NUM SCL pulses. I2C_SCL_RST_SLV_EN will later be cleared automatically by hardware.

Note that the operation of other I²C masters and I²C slaves may differ from that of the ESP32-S2 I²C devices. Please refer to datasheets of specific I²C devices.

12.4.5 An I²C Master Reads an I²C Slave with a 7-bit Address in One Command Sequence

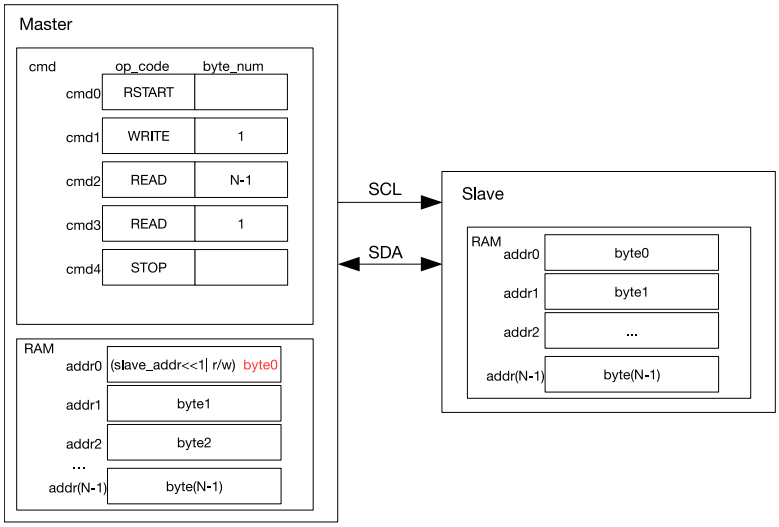


Figure 12-9. An I²C Master Reading an I²C Slave with a 7-bit Address

Figure 12-9 shows how an I²C master reads N bytes of data from an I²C slave using 7-bit addressing. cmd1 is a WRITE command, and when this command is executed the master sends a slave address. The byte sent comprises a 7-bit slave address and a $R/\overline{W}$ bit. When the $R/\overline{W}$ bit is 1, it indicates a READ operation. If the address of an I²C slave matches the sent address, this matching slave starts sending data to the master. The master generates acknowledgements according to ack_value defined in the READ command upon receiving every byte.

As illustrated in Figure 12-9, the master executes two READ commands: it generates ACKs for (N-1) bytes of data in cmd2, and a NACK for the last byte of data in cmd 3. This configuration may be changed as required. The master writes received data into the controller RAM from addr0, whose original content (a slave address and a $R/\overline{W}$ bit) is overwritten by byte0 marked red in Figure 12-9.

12.4.6 An I²C Master Reads an I²C Slave with a 10-bit Address in One Command Sequence

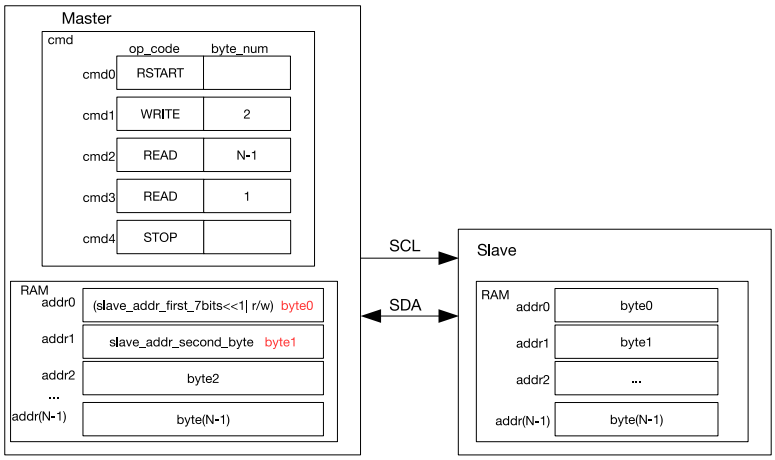


Figure 12-10. An I²C Master Reading an I²C Slave with a 10-bit Address

Figure 12-10 shows how an I²C master reads data from an I²C slave using 10-bit addressing. Unlike 7-bit addressing, in 10-bit addressing the WRITE command of the I²C master is formed from two bytes, and correspondingly the RAM of this master stores a 10-bit address of two bytes. I2C_ADDR_10BIT_EN and I2C_SLAVE_ADDR[14:0] should be set. Please refer to 12.4.2 for how to set this register.

12.4.7 An I²C Master Reads an I²C Slave with Two 7-bit Addresses in One Command Sequence

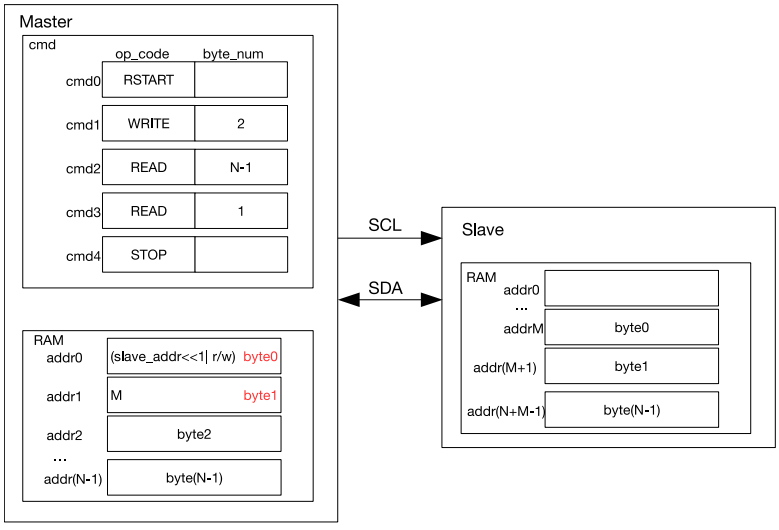


Figure 12-11. An I²C Master Reading N Bytes of Data from addrM of an I²C Slave with a 7-bit Address

Figure 12-11 shows how an I²C master reads data from specified addresses in an I²C slave. This procedure is as follows:

1. Set I2C_FIFO_ADDR_CFG_EN and prepare data to be sent in the RAM of the slave.
2. Prepare the slave address and register address M in the master.

3. Set I2C_TRANS_START and start transferring N bytes of data in the slave's RAM starting from address M to the master.

12.4.8 An I²C Master Reads an I²C Slave with a 7-bit Address in Multiple Command Sequences

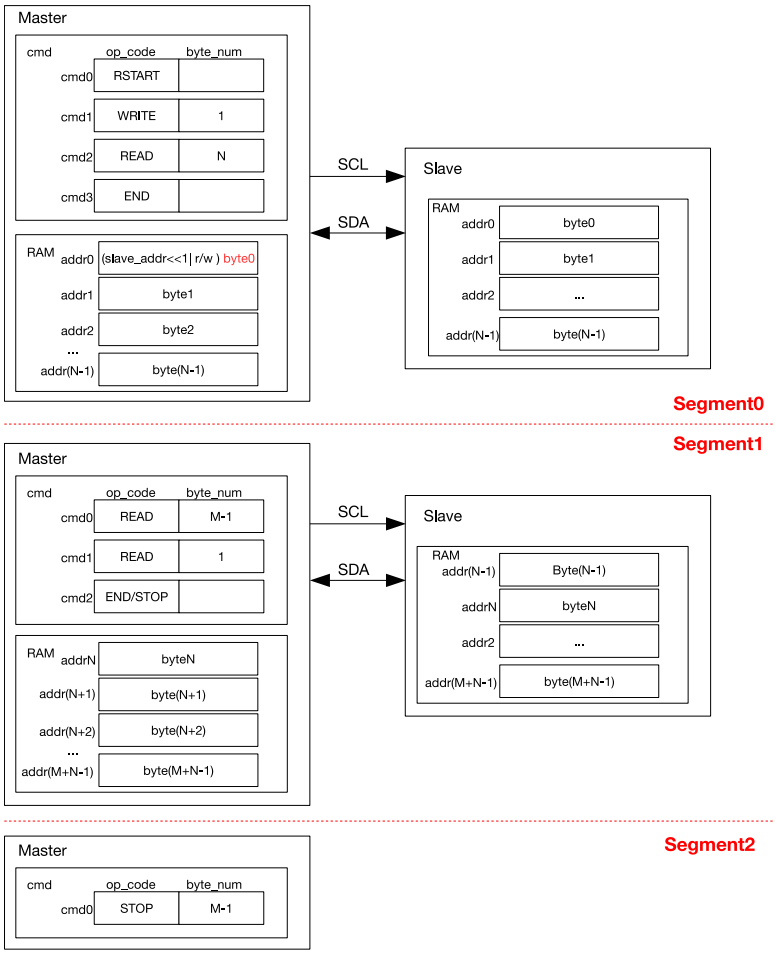


Figure 12-12. An I²C Master Reading an I²C Slave with a 7-bit Address in Segments

Figure 12-12 shows how an I²C master reads (N+M) bytes of data from an I²C slave in two/three segments separated by END commands. Configuration procedures are described as follows:

1. Configure the command register and the RAM, as shown in Segment0.
2. Prepare data in the RAM of the slave, and set I2C_TRANS_START to start data transfer. After executing the END command, the master refreshes command registers and the RAM as shown in Segment1, and clears the corresponding I2C_END_DETECT_INT interrupt. If cmd2 in the second segment is a STOP, then data is read from the slave in two segments. The master resumes data transfer by setting I2C_TRANS_START and terminates the transfer by sending a STOP bit.
3. If cmd2 in Segment1 is an END, then data is read from the slave in three segments. After the second data transfer finishes and an I2C_END_DETECT_INT interrupt is detected, the cmd box is configured as shown in Segment2. Once I2C_TRANS_START is set, the master terminates the transfer by sending a STOP bit.

12.5 Clock Stretching

In slave mode, the I²C controller can hold the SCL line low in exchange for more processing time. This function called clock stretching is enabled by setting I2C_SLAVE_SCL_STRETCH_EN bit. The time period of clock stretching is configured by setting I2C_STRETCH_PROTECT_NUM bit. The SCL line will be held low when one of the following three events occurs:

1. Finding a match: In slave mode, the address of the I²C controller matches the address sent via the SDA line.
2. RAM being full: In slave mode, RX RAM of the I²C controller is full.
3. Data all sent: In slave mode, TX RAM of the I²C controller is empty.

After SCL has been stretched low, the cause of stretching can be read from I2C_STRETCH_CAUSE bit. Clock stretching is disabled by setting I2C_SLAVE_SCL_STRETCH_CLR bit.

12.6 Interrupts

- I2C_SLAVE_STRETCH_INT: Generated when a slave holds SCL low.
- I2C_DET_START_INT: Triggered when a START bit is detected.
- I2C_SCL_MAIN_ST_TO_INT: Triggered when main state machine SCL_MAIN_FSM remains unchanged for over I2C_SCL_MAIN_ST_TO[23:0] clock cycles.
- I2C_SCL_ST_TO_INT: Triggered when state machine SCL_FSM remains unchanged for over I2C_SCL_ST_TO[23:0] clock cycles.
- I2C_RXFIFO_UDF_INT: Triggered when the I²C controller receives I2C_NONFIFO_RX_THRES bytes of data in non-FIFO mode.
- I2C_TXFIFO_OVF_INT: Triggered when the I²C controller sends I2C_NONFIFO_TX_THRES bytes of data.
- I2C_NACK_INT: Triggered when the ACK value received by the master is not as expected, or when the ACK value received by the slave is 1.
- I2C_TRANS_START_INT: Triggered when the I²C controller sends a START bit.
- I2C_TIME_OUT_INT: Triggered when SCL stays high or low for more than I2C_TIME_OUT clock cycles during data transfer.
- I2C_TRANS_COMPLETE_INT: Triggered when the I²C controller detects a STOP bit.
- I2C_MST_TXFIFO_UDF_INT: Triggered when TX FIFO of the master underflows.
- I2C_ARBITRATION_LOST_INT: Triggered when the SDA's output value does not match its input value while the master's SCL is high.
- I2C_BYTE_TRANS_DONE_INT: Triggered when the I²C controller sends or receives a byte.
- I2C_END_DETECT_INT: Triggered when op_code of the master indicates an END command and an END condition is detected.
- I2C_RXFIFO_OVF_INT: Triggered when Rx FIFO of the I²C controller overflows.
- I2C_TXFIFO_WM_INT: I²C TX FIFO watermark interrupt. Triggered when I2C_FIFO_PRT_EN is 1 and the pointers of TX FIFO are less than I2C_TXFIFO_WM_THRHD[4:0].

- I2C_RXFIFO_WM_INT: I²C Rx FIFO watermark interrupt. Triggered when I2C_FIFO_PRT_EN is 1 and the pointers of Rx FIFO are greater than I2C_RXFIFO_WM_THRHD[4:0].

12.7 Base Address

Users can access the I²C Controller with base addresses in Table 12-1. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 12-1. I²C Controller Base Address

	Bus to Access Peripheral	Base Address
I²C0	PeriBUS1	0x3F413000
	PeriBUS2	0x60013000
I²C1	PeriBUS1	0x3F427000
	PeriBUS2	0x60027000

12.8 Register Summary

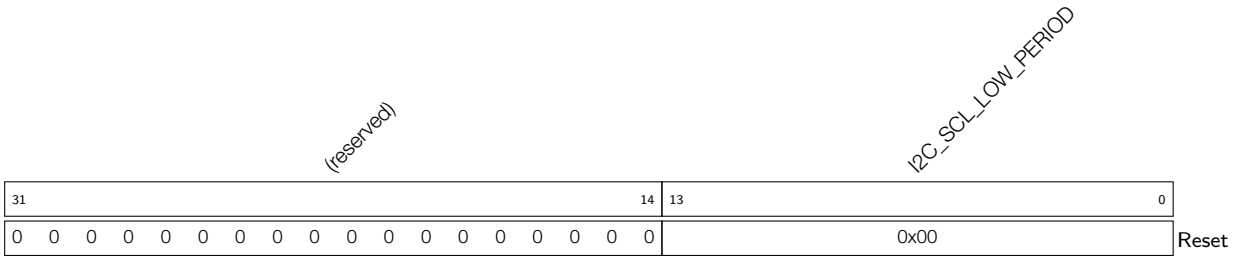
Addresses in the following table are relative to I²C base addresses provided in Section 12.7.

Name	Description	Address	Access
Timing Register			
I2C_SCL_LOW_PERIOD_REG	Configures the low level width of the SCL clock	0x0000	R/W
I2C_SDA_HOLD_REG	Configures the hold time after a negative SCL edge	0x0030	R/W
I2C_SDA_SAMPLE_REG	Configures the sample time after a positive SCL edge	0x0034	R/W
I2C_SCL_HIGH_PERIOD_REG	Configures the high level width of the SCL clock	0x0038	R/W
I2C_SCL_START_HOLD_REG	Configures the interval between pulling SDA low and pulling SCL low when the master generates a START condition	0x0040	R/W
I2C_SCL_RSTART_SETUP_REG	Configures the interval between the positive edge of SCL and the negative edge of SDA	0x0044	R/W
I2C_SCL_STOP_HOLD_REG	Configures the delay after the SCL clock edge for a stop condition	0x0048	R/W
I2C_SCL_STOP_SETUP_REG	Configures the delay between the SDA and SCL positive edge for a stop condition	0x004C	R/W
I2C_SCL_ST_TIME_OUT_REG	SCL status time out register	0x0098	R/W
I2C_SCL_MAIN_ST_TIME_OUT_REG	SCL main status time out register	0x009C	R/W
Configuration Register			
I2C_CTR_REG	Transmission setting	0x0004	R/W
I2C_TO_REG	Setting time out control for receiving data	0x000C	R/W
I2C_SLAVE_ADDR_REG	Local slave address setting	0x0010	R/W
I2C_FIFO_CONF_REG	FIFO configuration register	0x0018	R/W
I2C_SCL_SP_CONF_REG	Power configuration register	0x00A0	R/W

Name	Description	Address	Access
I2C_SCL_STRETCH_CONF_REG	Set SCL stretch of I2C slave	0x00A4	varies
Status Register			
I2C_SR_REG	Describe I2C work status	0x0008	RO
I2C_FIFO_ST_REG	FIFO status register	0x0014	varies
I2C_DATA_REG	RX FIFO read data	0x001C	RO
Interrupt Register			
I2C_INT_RAW_REG	Raw interrupt status	0x0020	RO
I2C_INT_CLR_REG	Interrupt clear bits	0x0024	WO
I2C_INT_ENA_REG	Interrupt enable bits	0x0028	R/W
I2C_INT_STATUS_REG	Status of captured I2C communication events	0x002C	RO
Filter Register			
I2C_SCL_FILTER_CFG_REG	SCL filter configuration register	0x0050	R/W
I2C_SDA_FILTER_CFG_REG	SDA filter configuration register	0x0054	R/W
Command Register			
I2C_COMD0_REG	I2C command register 0	0x0058	R/W
I2C_COMD1_REG	I2C command register 1	0x005C	R/W
I2C_COMD2_REG	I2C command register 2	0x0060	R/W
I2C_COMD3_REG	I2C command register 3	0x0064	R/W
I2C_COMD4_REG	I2C command register 4	0x0068	R/W
I2C_COMD5_REG	I2C command register 5	0x006C	R/W
I2C_COMD6_REG	I2C command register 6	0x0070	R/W
I2C_COMD7_REG	I2C command register 7	0x0074	R/W
I2C_COMD8_REG	I2C command register 8	0x0078	R/W
I2C_COMD9_REG	I2C command register 9	0x007C	R/W
I2C_COMD10_REG	I2C command register 10	0x0080	R/W
I2C_COMD11_REG	I2C command register 11	0x0084	R/W
I2C_COMD12_REG	I2C command register 12	0x0088	R/W
I2C_COMD13_REG	I2C command register 13	0x008C	R/W
I2C_COMD14_REG	I2C command register 14	0x0090	R/W
I2C_COMD15_REG	I2C command register 15	0x0094	R/W
Version Register			
I2C_DATE_REG	Version control register	0x00F8	R/W

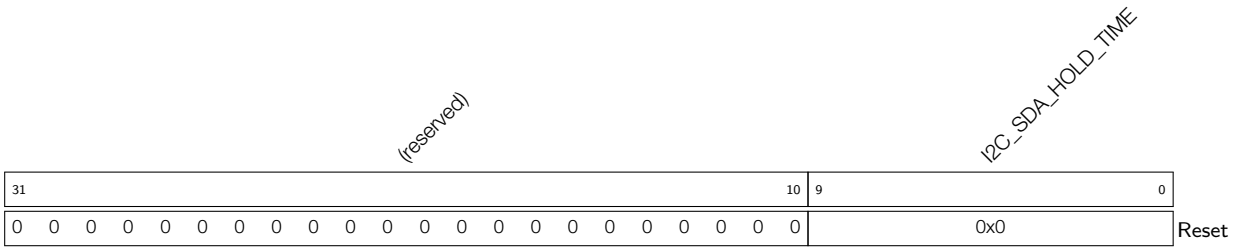
12.9 Registers

Register 12.1: I2C_SCL_LOW_PERIOD_REG (0x0000)



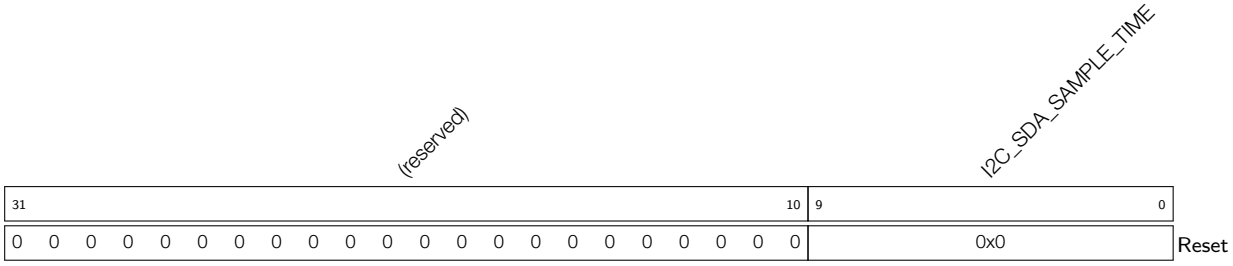
I2C_SCL_LOW_PERIOD This register is used to configure for how long SCL remains low in master mode, in I2C module clock cycles. (R/W)

Register 12.2: I2C_SDA_HOLD_REG (0x0030)



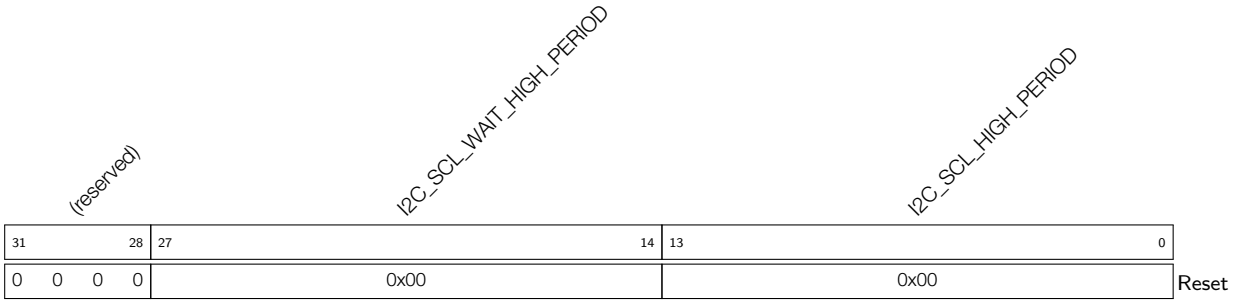
I2C_SDA_HOLD_TIME This register is used to configure the interval between changing the SDA output level and the falling edge of SCL, in I2C module clock cycles. (R/W)

Register 12.3: I2C_SDA_SAMPLE_REG (0x0034)



I2C_SDA_SAMPLE_TIME This register is used to configure the interval between the rising edge of SCL and the level sampling time of SDA, in I2C module clock cycles. (R/W)

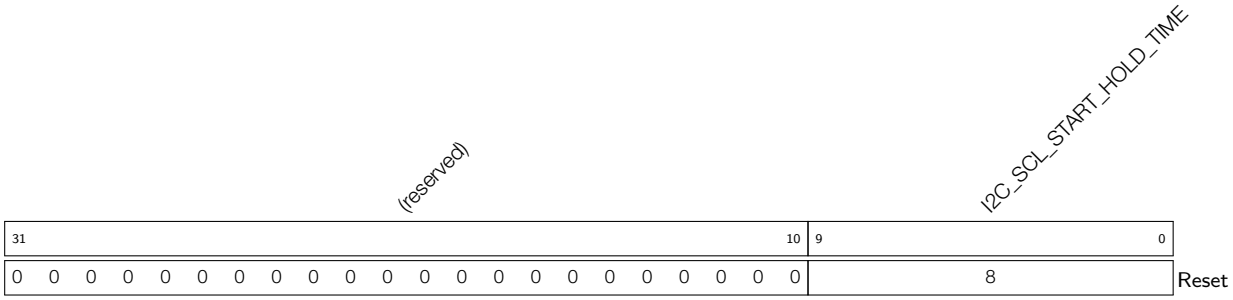
Register 12.4: I2C_SCL_HIGH_PERIOD_REG (0x0038)



I2C_SCL_HIGH_PERIOD This register is used to configure for how long SCL remains high in master mode, in I2C module clock cycles. (R/W)

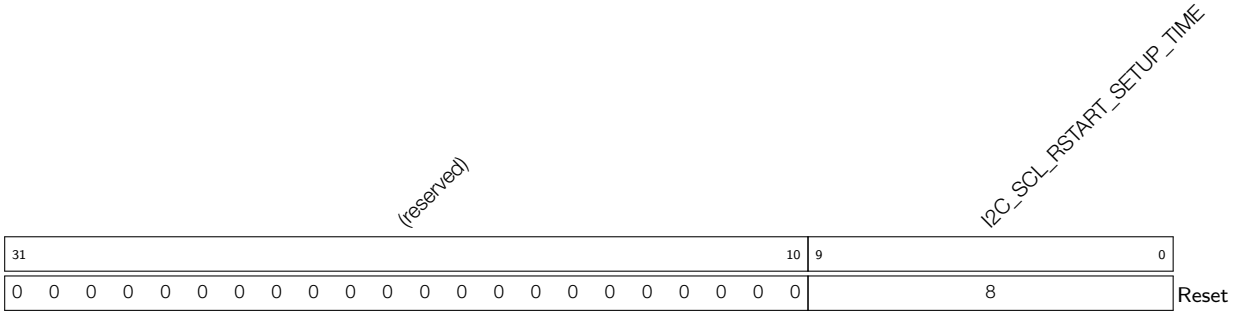
I2C_SCL_WAIT_HIGH_PERIOD This register is used to configure for the SCL_FSM's waiting period for SCL to go high in master mode, in I2C module clock cycles. (R/W)

Register 12.5: I2C_SCL_START_HOLD_REG (0x0040)



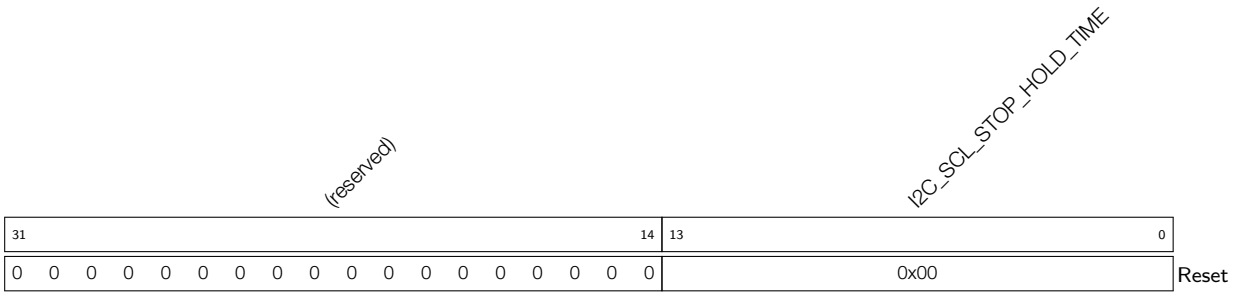
I2C_SCL_START_HOLD_TIME This register is used to configure interval between pulling SDA low and pulling SCL low when the master generates a START condition, in I2C module clock cycles. (R/W)

Register 12.6: I2C_SCL_RSTART_SETUP_REG (0x0044)



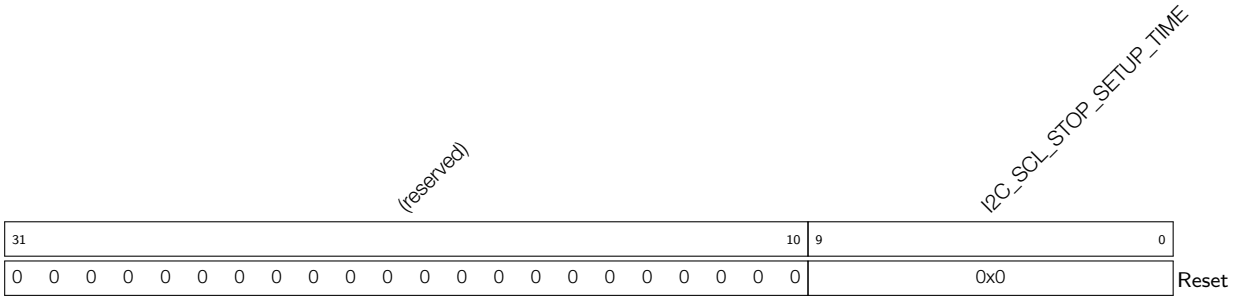
I2C_SCL_RSTART_SETUP_TIME This register is used to configure the interval between the positive edge of SCL and the negative edge of SDA for a RESTART condition, in I2C module clock cycles. (R/W)

Register 12.7: I2C_SCL_STOP_HOLD_REG (0x0048)



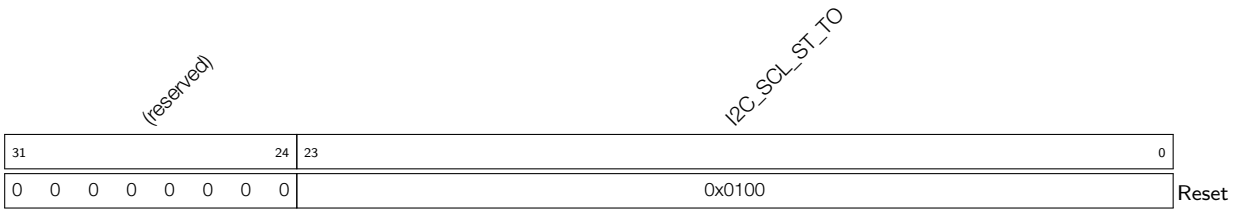
I2C_SCL_STOP_HOLD_TIME This register is used to configure the delay after the STOP condition, in I2C module clock cycles. (R/W)

Register 12.8: I2C_SCL_STOP_SETUP_REG (0x004C)



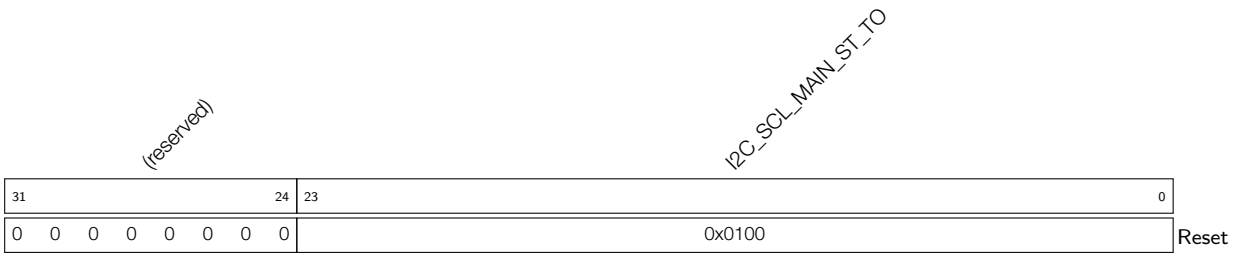
I2C_SCL_STOP_SETUP_TIME This register is used to configure the time between the positive edge of SCL and the positive edge of SDA, in I2C module clock cycles. (R/W)

Register 12.9: I2C_SCL_ST_TIME_OUT_REG (0x0098)



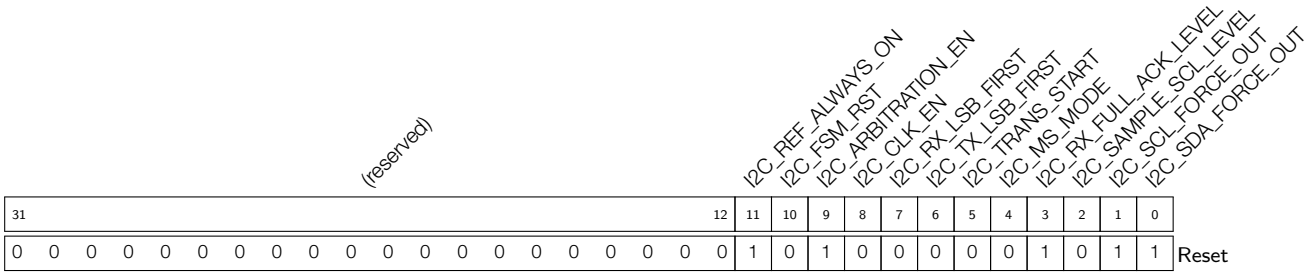
I2C_SCL_ST_TO The threshold value of SCL_FSM state unchanged period. (R/W)

Register 12.10: I2C_SCL_MAIN_ST_TIME_OUT_REG (0x009C)



I2C_SCL_MAIN_ST_TO The threshold value of SCL_MAIN_FSM state unchanged period. (R/W)

Register 12.11: I2C_CTR_REG (0x0004)



I2C_SDA_FORCE_OUT 0: direct output; 1: open drain output. (R/W)

I2C_SCL_FORCE_OUT 0: direct output; 1: open drain output. (R/W)

I2C_SAMPLE_SCL_LEVEL This register is used to select the sample mode. 1: sample SDA data on the SCL low level. 0: sample SDA data on the SCL high level. (R/W)

I2C_RX_FULL_ACK_LEVEL This register is used to configure the ACK value that need to sent by master when the rx_fifo_cnt has reached the threshold. (R/W)

I2C_MS_MODE Set this bit to configure the module as an I2C Master. Clear this bit to configure the module as an I2C Slave. (R/W)

I2C_TRANS_START Set this bit to start sending the data in TX FIFO. (R/W)

I2C_TX_LSB_FIRST This bit is used to control the sending mode for data needing to be sent. 1: send data from the least significant bit; 0: send data from the most significant bit. (R/W)

I2C_RX_LSB_FIRST This bit is used to control the storage mode for received data. 1: receive data from the least significant bit; 0: receive data from the most significant bit. (R/W)

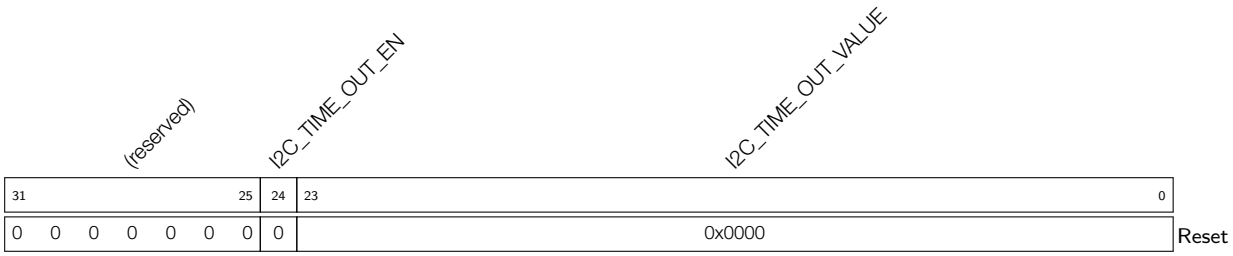
I2C_CLK_EN Reserved (R/W)

I2C_ARBITRATION_EN This is the enable bit for I2C bus arbitration function. (R/W)

I2C_FSM_RST This register is used to reset the SCL_FSM. (R/W)

I2C_REF_ALWAYS_ON This register is used to control the REF_TICK. (R/W)

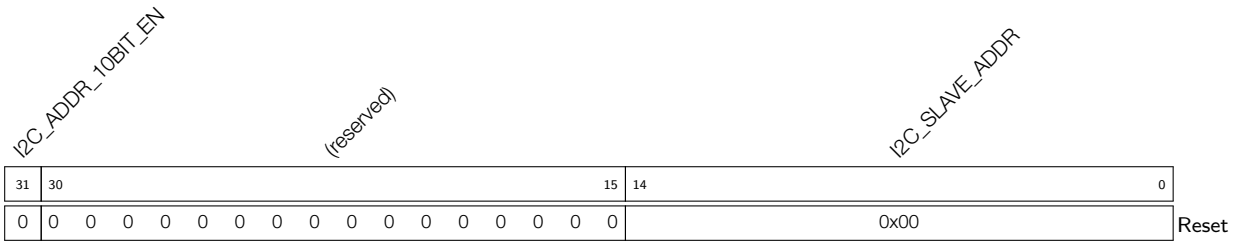
Register 12.12: I2C_TO_REG (0x000C)



I2C_TIME_OUT_VALUE This register is used to configure the timeout for receiving a data bit in APB clock cycles. (R/W)

I2C_TIME_OUT_EN This is the enable bit for time out control. (R/W)

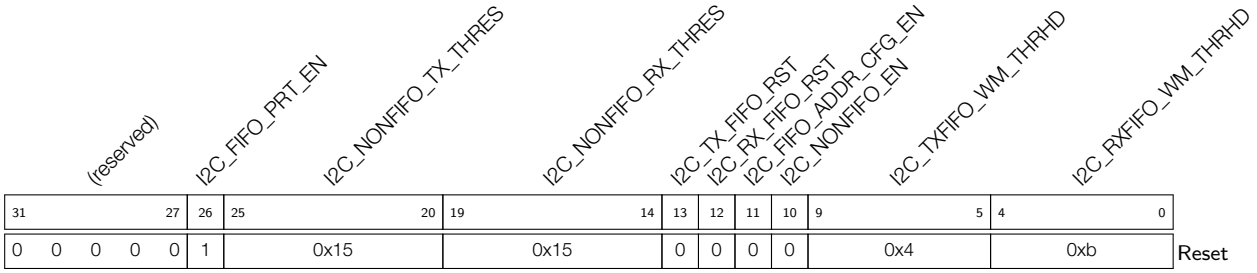
Register 12.13: I2C_SLAVE_ADDR_REG (0x0010)



I2C_SLAVE_ADDR When configured as an I2C Slave, this field is used to configure the slave address. (R/W)

I2C_ADDR_10BIT_EN This field is used to enable the slave 10-bit addressing mode in master mode. (R/W)

Register 12.14: I2C_FIFO_CONF_REG (0x0018)



I2C_RXFIFO_WM_THRHD The water mark threshold of RX FIFO in non-FIFO mode. When I2C_FIFO_PRT_EN is 1 and RX FIFO counter is bigger than I2C_RXFIFO_WM_THRHD[4:0], I2C_RXFIFO_WM_INT_RAW bit will be valid. (R/W)

I2C_TXFIFO_WM_THRHD The water mark threshold of TX FIFO in non-FIFO mode. When I2C_FIFO_PRT_EN is 1 and TX FIFO counter is smaller than I2C_TXFIFO_WM_THRHD[4:0], I2C_TXFIFO_WM_INT_RAW bit will be valid. (R/W)

I2C_NONFIFO_EN Set this bit to enable APB non-FIFO mode. (R/W)

I2C_FIFO_ADDR_CFG_EN When this bit is set to 1, the byte received after the I2C address byte represents the offset address in the I2C Slave RAM. (R/W)

I2C_RX_FIFO_RST Set this bit to reset RX FIFO. (R/W)

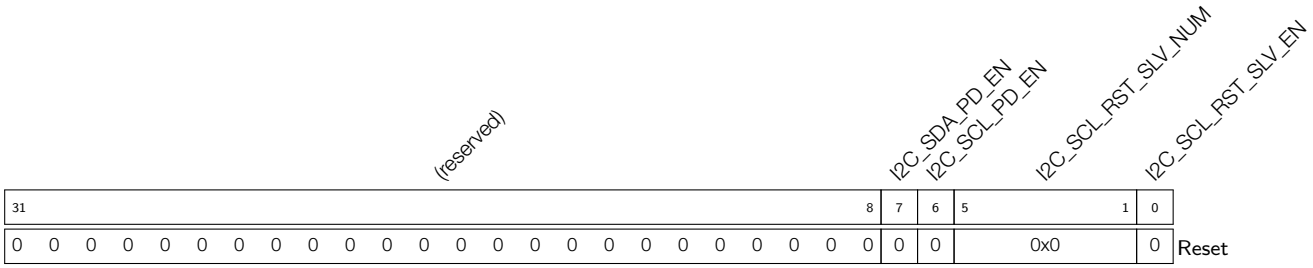
I2C_TX_FIFO_RST Set this bit to reset TX FIFO. (R/W)

I2C_NONFIFO_RX_THRES When I2C receives more than I2C_NONFIFO_RX_THRES bytes of data, it will generate an I2C_RXFIFO_UDF_INT interrupt and update the current offset address of the received data. (R/W)

I2C_NONFIFO_TX_THRES When I2C sends more than I2C_NONFIFO_TX_THRES bytes of data, it will generate an I2C_TXFIFO_OVF_INT interrupt and update the current offset address of the sent data. (R/W)

I2C_FIFO_PRT_EN The control enable bit of FIFO pointer in non-FIFO mode. This bit controls the valid bits and the interrupts of TX/RX FIFO overflow, underflow, full and empty. (R/W)

Register 12.15: I2C_SCL_SP_CONF_REG (0x00A0)



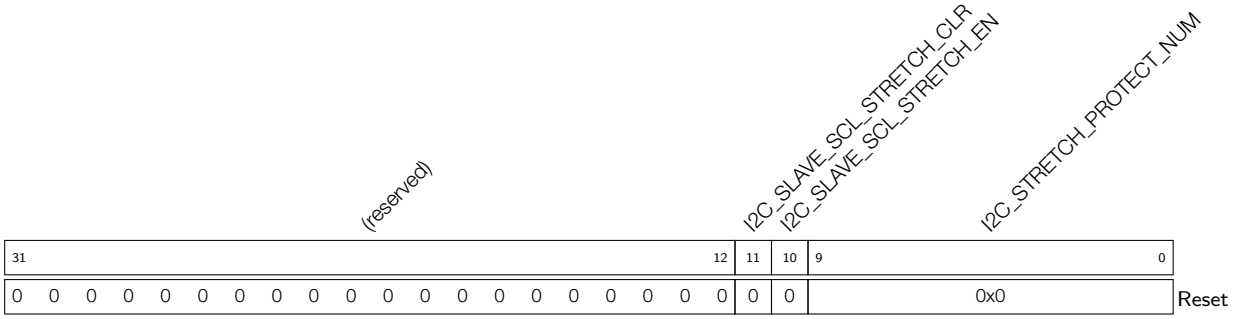
I2C_SCL_RST_SLV_EN When I2C master is IDLE, set this bit to send out SCL pulses. The number of pulses equals to I2C_SCL_RST_SLV_NUM[4:0]. (R/W)

I2C_SCL_RST_SLV_NUM Configure the pulses of SCL generated in I2C master mode. Valid when I2C_SCL_RST_SLV_EN is 1. (R/W)

I2C_SCL_PD_EN The power down enable bit for the I2C output SCL line. 1: Power down. 0: Not power down. Set I2C_SCL_FORCE_OUT and I2C_SCL_PD_EN to 1 to stretch SCL low. (R/W)

I2C_SDA_PD_EN The power down enable bit for the I2C output SDA line. 1: Power down. 0: Not power down. Set I2C_SDA_FORCE_OUT and I2C_SDA_PD_EN to 1 to stretch SDA low. (R/W)

Register 12.16: I2C_SCL_STRETCH_CONF_REG (0x00A4)

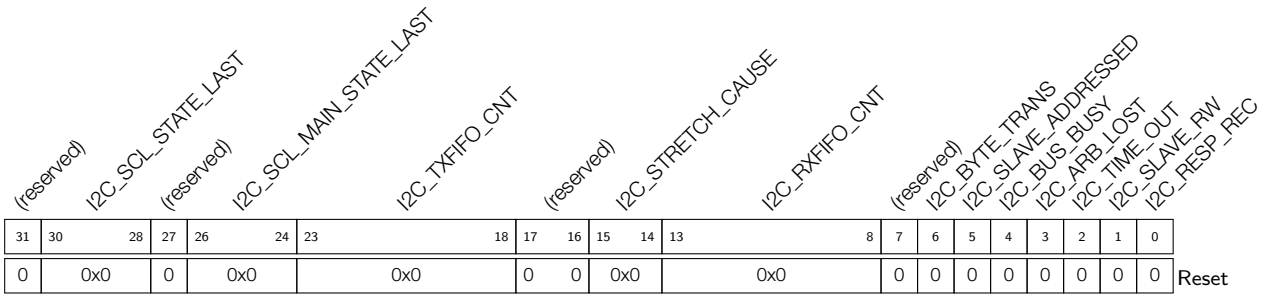


I2C_STRETCH_PROTECT_NUM Configure the period of I2C slave stretching SCL line. (R/W)

I2C_SLAVE_SCL_STRETCH_EN The enable bit for slave SCL stretch function. 1: Enable. 0: Disable. The SCL output line will be stretched low when I2C_SLAVE_SCL_STRETCH_EN is 1 and stretch event happens. The stretch cause can be seen in I2C_STRETCH_CAUSE. (R/W)

I2C_SLAVE_SCL_STRETCH_CLR Set this bit to clear the I2C slave SCL stretch function. (WO)

Register 12.17: I2C_SR_REG (0x0008)



I2C_RESP_REC The received ACK value in master mode or slave mode. 0: ACK; 1: NACK. (RO)

I2C_SLAVE_RW When in slave mode, 1: master reads from slave; 0: master writes to slave. (RO)

I2C_TIME_OUT When the I2C controller takes more than I2C_TIME_OUT clocks to receive a data bit, this field changes to 1. (RO)

I2C_ARB_LOST When the I2C controller loses control of SCL line, this register changes to 1. (RO)

I2C_BUS_BUSY 1: the I2C bus is busy transferring data; 0: the I2C bus is in idle state. (RO)

I2C_SLAVE_ADDRESSED When configured as an I2C Slave, and the address sent by the master is equal to the address of the slave, then this bit will be of high level. (RO)

I2C_BYTE_TRANS This field changes to 1 when one byte is transferred. (RO)

I2C_RXFIFO_CNT This field represents the amount of data needed to be sent. (RO)

I2C_STRETCH_CAUSE The cause of stretching SCL low in slave mode. 0: stretching SCL low at the beginning of I2C read data state. 1: stretching SCL low when I2C TX FIFO is empty in slave mode. 2: stretching SCL low when I2C RX FIFO is full in slave mode. (RO)

I2C_TXFIFO_CNT This field stores the amount of received data in RAM. (RO)

I2C_SCL_MAIN_STATE_LAST This field indicates the states of the I2C module state machine. 0: Idle; 1: Address shift; 2: ACK address; 3: RX data; 4: TX data; 5: Send ACK; 6: Wait ACK (RO)

I2C_SCL_STATE_LAST This field indicates the states of the state machine used to produce SCL. 0: Idle; 1: Start; 2: Negative edge; 3: Low; 4: Positive edge; 5: High; 6: Stop (RO)

Register 12.18: I2C_FIFO_ST_REG (0x0014)

(reserved)	I2C_SLAVE_RW_POINT	I2C_TX_UPDATE	I2C_RX_UPDATE	I2C_TXFIFO_END_ADDR	I2C_TXFIFO_START_ADDR	I2C_RXFIFO_END_ADDR	I2C_RXFIFO_START_ADDR	
31 30	29 22	21	20	19 15	14 10	9 5	4 0	
0 0	0x0	0	0	0x0	0x0	0x0	0x0	Reset

I2C_RXFIFO_START_ADDR This is the offset address of the last received data, as described in I2C_NONFIFO_RX_THRES. (RO)

I2C_RXFIFO_END_ADDR This is the offset address of the last received data, as described in I2C_NONFIFO_RX_THRES. This value refreshes when an I2C_RXFIFO_UDF_INT or I2C_TRANS_COMPLETE_INT interrupt is generated. (RO)

I2C_TXFIFO_START_ADDR This is the offset address of the first sent data, as described in I2C_NONFIFO_TX_THRES. (RO)

I2C_TXFIFO_END_ADDR This is the offset address of the last sent data, as described in I2C_NONFIFO_TX_THRES. The value refreshes when an I2C_TXFIFO_OVF_INT or I2C_TRANS_COMPLETE_INT interrupt is generated. (RO)

I2C_RX_UPDATE Write 0 or 1 to I2C_RX_UPDATE to update the value of I2C_RXFIFO_END_ADDR and I2C_RXFIFO_START_ADDR. (WO)

I2C_TX_UPDATE Write 0 or 1 to I2C_TX_UPDATE to update the value of I2C_TXFIFO_END_ADDR and I2C_TXFIFO_START_ADDR. (WO)

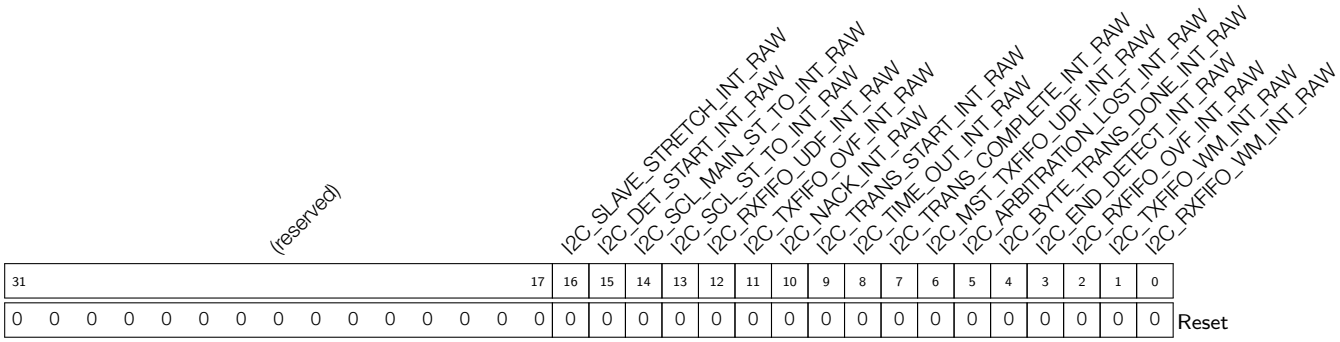
I2C_SLAVE_RW_POINT The received data in I2C slave mode. (RO)

Register 12.19: I2C_DATA_REG (0x001C)

(reserved)	I2C_FIFO_RDATA	
31 8	7 0	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0x0	Reset

I2C_FIFO_RDATA The value of RX FIFO read data. (RO)

Register 12.20: I2C_INT_RAW_REG (0x0020)



I2C_RXFIFO_WM_INT_RAW The raw interrupt bit for I2C_RXFIFO_WM_INT interrupt. (RO)

I2C_TXFIFO_WM_INT_RAW The raw interrupt bit for I2C_TXFIFO_WM_INT interrupt. (RO)

I2C_RXFIFO_OVF_INT_RAW The raw interrupt bit for I2C_RXFIFO_OVF_INT interrupt. (RO)

I2C_END_DETECT_INT_RAW The raw interrupt bit for the I2C_END_DETECT_INT interrupt. (RO)

I2C_BYTE_TRANS_DONE_INT_RAW The raw interrupt bit for the I2C_END_DETECT_INT interrupt. (RO)

I2C_ARBITRATION_LOST_INT_RAW The raw interrupt bit for the I2C_ARBITRATION_LOST_INT interrupt. (RO)

I2C_MST_TXFIFO_UDF_INT_RAW The raw interrupt bit for I2C_TRANS_COMPLETE_INT interrupt. (RO)

I2C_TRANS_COMPLETE_INT_RAW The raw interrupt bit for the I2C_TRANS_COMPLETE_INT interrupt. (RO)

I2C_TIME_OUT_INT_RAW The raw interrupt bit for the I2C_TIME_OUT_INT interrupt. (RO)

I2C_TRANS_START_INT_RAW The raw interrupt bit for the I2C_TRANS_START_INT interrupt. (RO)

I2C_NACK_INT_RAW The raw interrupt bit for I2C_SLAVE_STRETCH_INT interrupt. (RO)

I2C_TXFIFO_OVF_INT_RAW The raw interrupt bit for I2C_TXFIFO_OVF_INT interrupt. (RO)

I2C_RXFIFO_UDF_INT_RAW The raw interrupt bit for I2C_RXFIFO_UDF_INT interrupt. (RO)

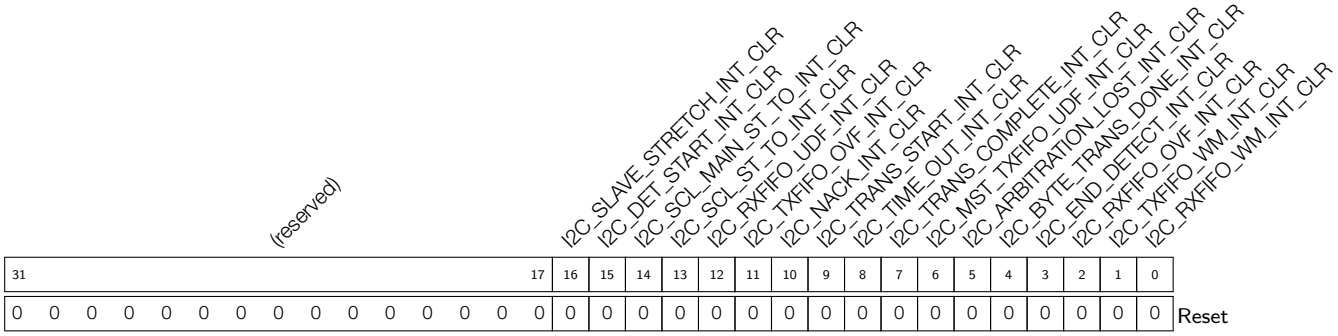
I2C_SCL_ST_TO_INT_RAW The raw interrupt bit for I2C_SCL_ST_TO_INT interrupt. (RO)

I2C_SCL_MAIN_ST_TO_INT_RAW The raw interrupt bit for I2C_SCL_MAIN_ST_TO_INT interrupt. (RO)

I2C_DET_START_INT_RAW The raw interrupt bit for I2C_DET_START_INT interrupt. (RO)

I2C_SLAVE_STRETCH_INT_RAW The raw interrupt bit for I2C_SLAVE_STRETCH_INT interrupt. (RO)

Register 12.21: I2C_INT_CLR_REG (0x0024)



I2C_RXFIFO_WM_INT_CLR Set this bit to clear I2C_RXFIFO_WM_INT interrupt. (WO)

I2C_TXFIFO_WM_INT_CLR Set this bit to clear I2C_TXFIFO_WM_INT interrupt. (WO)

I2C_RXFIFO_OVF_INT_CLR Set this bit to clear I2C_RXFIFO_OVF_INT interrupt. (WO)

I2C_END_DETECT_INT_CLR Set this bit to clear the I2C_END_DETECT_INT interrupt. (WO)

I2C_BYTE_TRANS_DONE_INT_CLR Set this bit to clear the I2C_END_DETECT_INT interrupt. (WO)

I2C_ARBITRATION_LOST_INT_CLR Set this bit to clear the I2C_ARBITRATION_LOST_INT interrupt. (WO)

I2C_MST_TXFIFO_UDF_INT_CLR Set this bit to clear I2C_TRANS_COMPLETE_INT interrupt. (WO)

I2C_TRANS_COMPLETE_INT_CLR Set this bit to clear the I2C_TRANS_COMPLETE_INT interrupt. (WO)

I2C_TIME_OUT_INT_CLR Set this bit to clear the I2C_TIME_OUT_INT interrupt. (WO)

I2C_TRANS_START_INT_CLR Set this bit to clear the I2C_TRANS_START_INT interrupt. (WO)

I2C_NACK_INT_CLR Set this bit to clear I2C_SLAVE_STRETCH_INT interrupt. (WO)

I2C_TXFIFO_OVF_INT_CLR Set this bit to clear I2C_TXFIFO_OVF_INT interrupt. (WO)

I2C_RXFIFO_UDF_INT_CLR Set this bit to clear I2C_RXFIFO_UDF_INT interrupt. (WO)

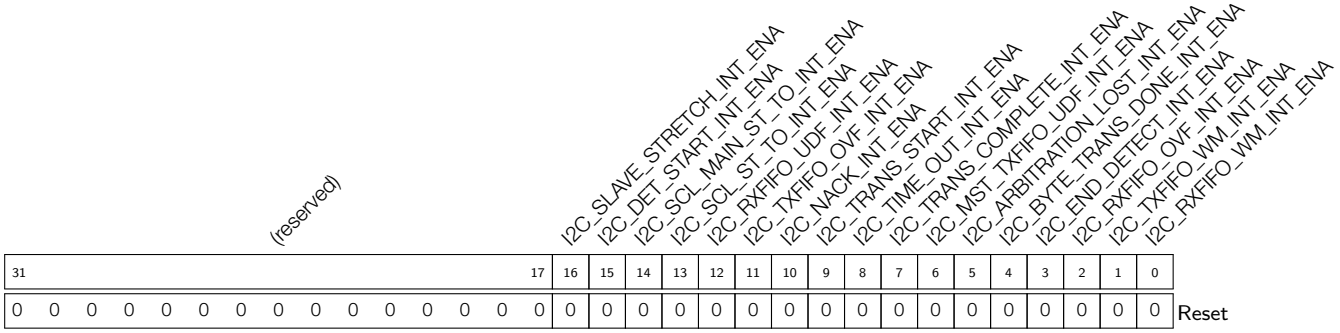
I2C_SCL_ST_TO_INT_CLR Set this bit to clear I2C_SCL_ST_TO_INT interrupt. (WO)

I2C_SCL_MAIN_ST_TO_INT_CLR Set this bit to clear I2C_SCL_MAIN_ST_TO_INT interrupt. (WO)

I2C_DET_START_INT_CLR Set this bit to clear I2C_DET_START_INT interrupt. (WO)

I2C_SLAVE_STRETCH_INT_CLR Set this bit to clear I2C_SLAVE_STRETCH_INT interrupt. (WO)

Register 12.22: I2C_INT_ENA_REG (0x0028)



I2C_RXFIFO_WM_INT_ENA The raw interrupt bit for I2C_RXFIFO_WM_INT interrupt. (R/W)

I2C_TXFIFO_WM_INT_ENA The raw interrupt bit for I2C_TXFIFO_WM_INT interrupt. (R/W)

I2C_RXFIFO_OVF_INT_ENA The raw interrupt bit for I2C_RXFIFO_OVF_INT interrupt. (R/W)

I2C_END_DETECT_INT_ENA The raw interrupt bit for the I2C_END_DETECT_INT interrupt. (R/W)

I2C_BYTE_TRANS_DONE_INT_ENA The raw interrupt bit for the I2C_END_DETECT_INT interrupt. (R/W)

I2C_ARBITRATION_LOST_INT_ENA The raw interrupt bit for the I2C_ARBITRATION_LOST_INT interrupt. (R/W)

I2C_MST_TXFIFO_UDF_INT_ENA The raw interrupt bit for I2C_TRANS_COMPLETE_INT interrupt. (R/W)

I2C_TRANS_COMPLETE_INT_ENA The raw interrupt bit for the I2C_TRANS_COMPLETE_INT interrupt. (R/W)

I2C_TIME_OUT_INT_ENA The raw interrupt bit for the I2C_TIME_OUT_INT interrupt. (R/W)

I2C_TRANS_START_INT_ENA The raw interrupt bit for the I2C_TRANS_START_INT interrupt. (R/W)

I2C_NACK_INT_ENA The raw interrupt bit for I2C_SLAVE_STRETCH_INT interrupt. (R/W)

I2C_TXFIFO_OVF_INT_ENA The raw interrupt bit for I2C_TXFIFO_OVF_INT interrupt. (R/W)

I2C_RXFIFO_UDF_INT_ENA The raw interrupt bit for I2C_RXFIFO_UDF_INT interrupt. (R/W)

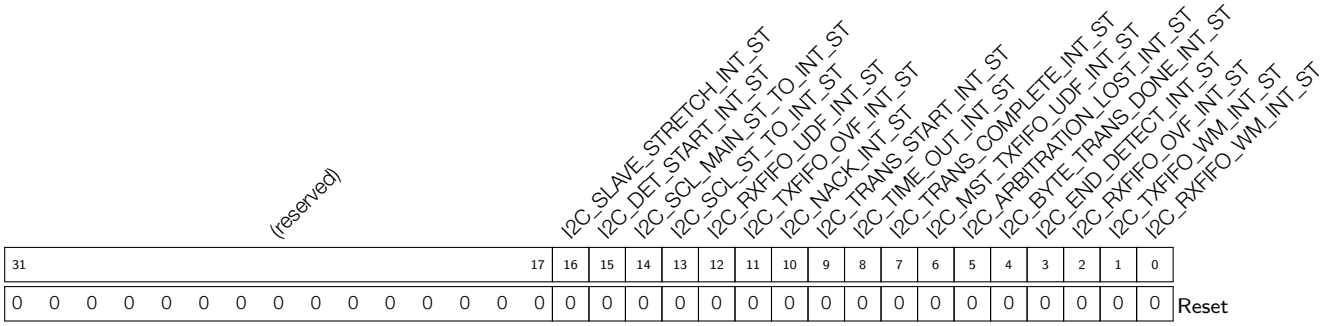
I2C_SCL_ST_TO_INT_ENA The raw interrupt bit for I2C_SCL_ST_TO_INT interrupt. (R/W)

I2C_SCL_MAIN_ST_TO_INT_ENA The raw interrupt bit for I2C_SCL_MAIN_ST_TO_INT interrupt. (R/W)

I2C_DET_START_INT_ENA The raw interrupt bit for I2C_DET_START_INT interrupt. (R/W)

I2C_SLAVE_STRETCH_INT_ENA The raw interrupt bit for I2C_SLAVE_STRETCH_INT interrupt. (R/W)

Register 12.23: I2C_INT_STATUS_REG (0x002C)



I2C_RXFIFO_WM_INT_ST The masked interrupt status bit for I2C_RXFIFO_WM_INT interrupt. (RO)

I2C_TXFIFO_WM_INT_ST The masked interrupt status bit for I2C_TXFIFO_WM_INT interrupt. (RO)

I2C_RXFIFO_OVF_INT_ST The masked interrupt status bit for I2C_RXFIFO_OVF_INT interrupt. (RO)

I2C_END_DETECT_INT_ST The masked interrupt status bit for the I2C_END_DETECT_INT interrupt. (RO)

I2C_BYTE_TRANS_DONE_INT_ST The masked interrupt status bit for the I2C_END_DETECT_INT interrupt. (RO)

I2C_ARBITRATION_LOST_INT_ST The masked interrupt status bit for the I2C_ARBITRATION_LOST_INT interrupt. (RO)

I2C_MST_TXFIFO_UDF_INT_ST The masked interrupt status bit for I2C_TRANS_COMPLETE_INT interrupt. (RO)

I2C_TRANS_COMPLETE_INT_ST The masked interrupt status bit for the I2C_TRANS_COMPLETE_INT interrupt. (RO)

I2C_TIME_OUT_INT_ST The masked interrupt status bit for the I2C_TIME_OUT_INT interrupt. (RO)

I2C_TRANS_START_INT_ST The masked interrupt status bit for the I2C_TRANS_START_INT interrupt. (RO)

I2C_NACK_INT_ST The masked interrupt status bit for I2C_SLAVE_STRETCH_INT interrupt. (RO)

I2C_TXFIFO_OVF_INT_ST The masked interrupt status bit for I2C_TXFIFO_OVF_INT interrupt. (RO)

I2C_RXFIFO_UDF_INT_ST The masked interrupt status bit for I2C_RXFIFO_UDF_INT interrupt. (RO)

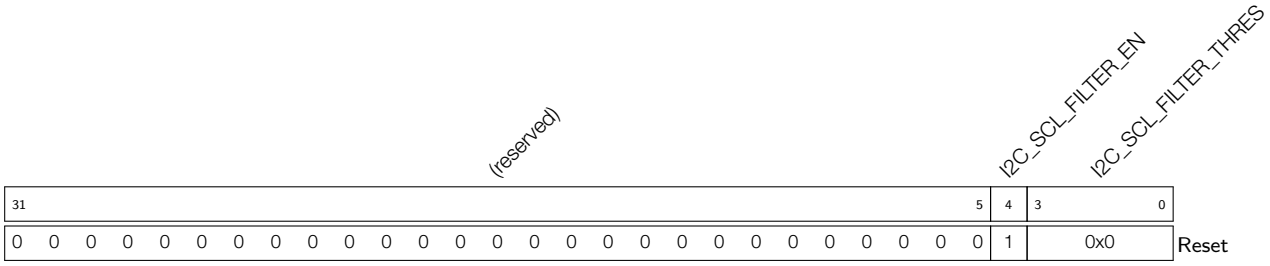
I2C_SCL_ST_TO_INT_ST The masked interrupt status bit for I2C_SCL_ST_TO_INT interrupt. (RO)

I2C_SCL_MAIN_ST_TO_INT_ST The masked interrupt status bit for I2C_SCL_MAIN_ST_TO_INT interrupt. (RO)

I2C_DET_START_INT_ST The masked interrupt status bit for I2C_DET_START_INT interrupt. (RO)

I2C_SLAVE_STRETCH_INT_ST The masked interrupt status bit for I2C_SLAVE_STRETCH_INT interrupt. (RO)

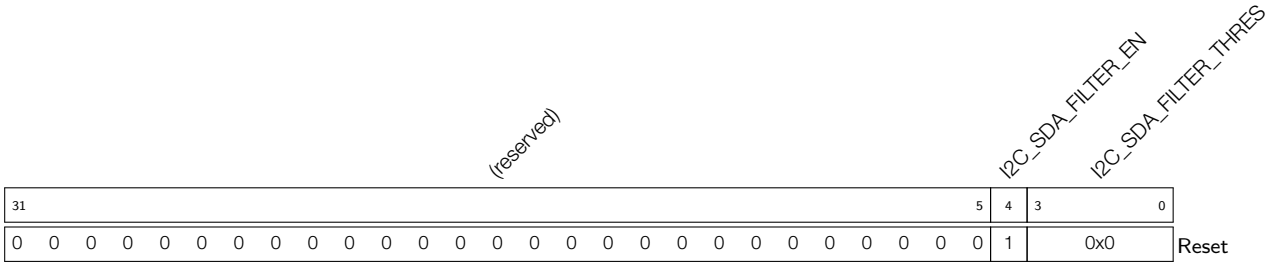
Register 12.24: I2C_SCL_FILTER_CFG_REG (0x0050)



I2C_SCL_FILTER_THRES When a pulse on the SCL input has smaller width than this register value in I2C module clock cycles, the I2C controller will ignore that pulse. (R/W)

I2C_SCL_FILTER_EN This is the filter enable bit for SCL. (R/W)

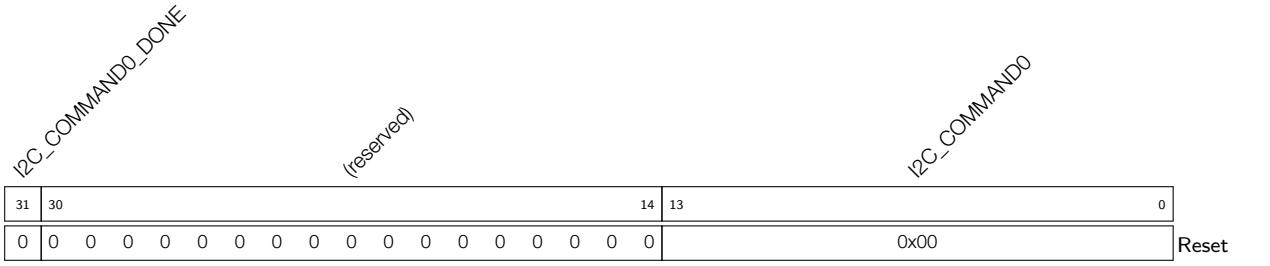
Register 12.25: I2C_SDA_FILTER_CFG_REG (0x0054)



I2C_SDA_FILTER_THRES When a pulse on the SDA input has smaller width than this register value in I2C module clock cycles, the I2C controller will ignore that pulse. (R/W)

I2C_SDA_FILTER_EN This is the filter enable bit for SDA. (R/W)

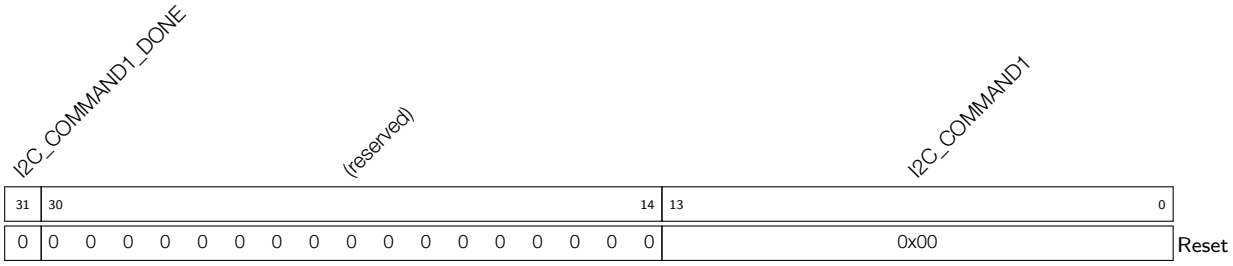
Register 12.26: I2C_COMD0_REG (0x0058)



I2C_COMMAND0 This is the content of command 0. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND0_DONE When command 0 is done in I2C Master mode, this bit changes to high level. (R/W)

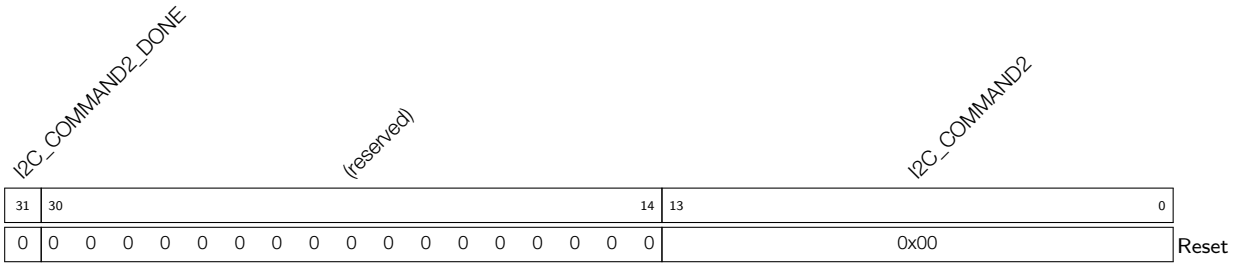
Register 12.27: I2C_COMD1_REG (0x005C)



I2C_COMMAND1 This is the content of command 1. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND1_DONE When command 1 is done in I2C Master mode, this bit changes to high level. (R/W)

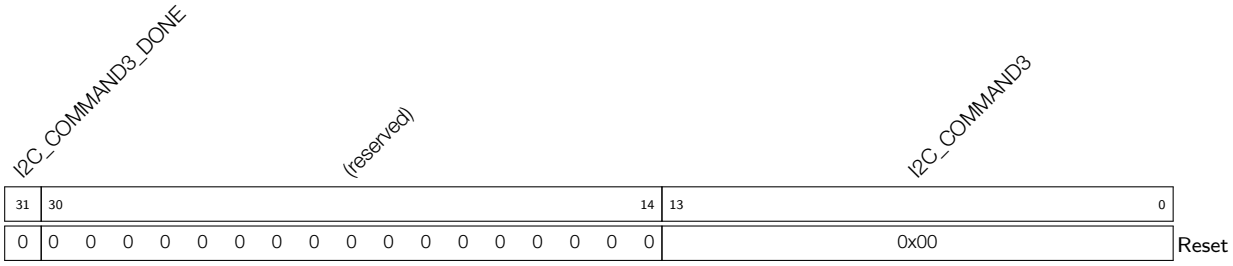
Register 12.28: I2C_COMD2_REG (0x0060)



I2C_COMMAND2 This is the content of command 2. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND2_DONE When command 2 is done in I2C Master mode, this bit changes to high Level. (R/W)

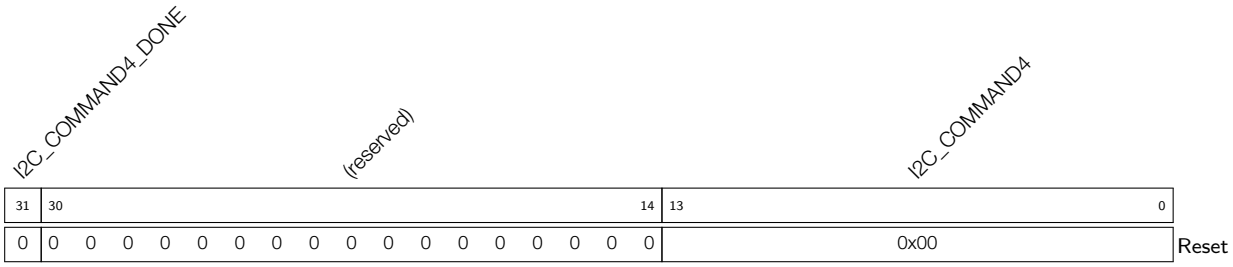
Register 12.29: I2C_COMD3_REG (0x0064)



I2C_COMMAND3 This is the content of command 3. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND3_DONE When command 3 is done in I2C Master mode, this bit changes to high level. (R/W)

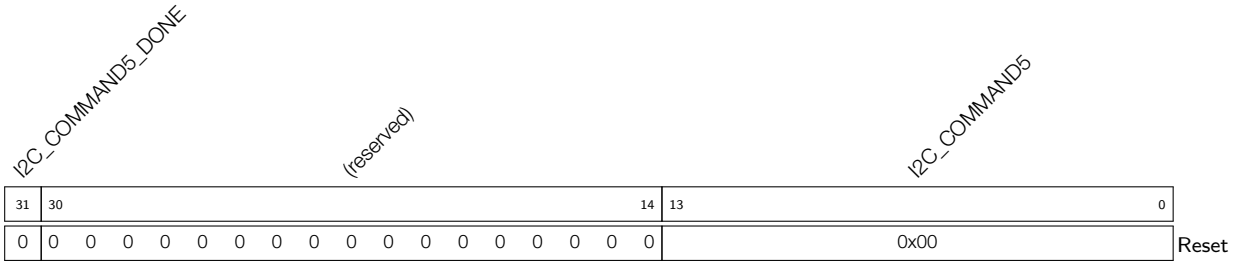
Register 12.30: I2C_COMD4_REG (0x0068)



I2C_COMMAND4 This is the content of command 4. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND4_DONE When command 4 is done in I2C Master mode, this bit changes to high level. (R/W)

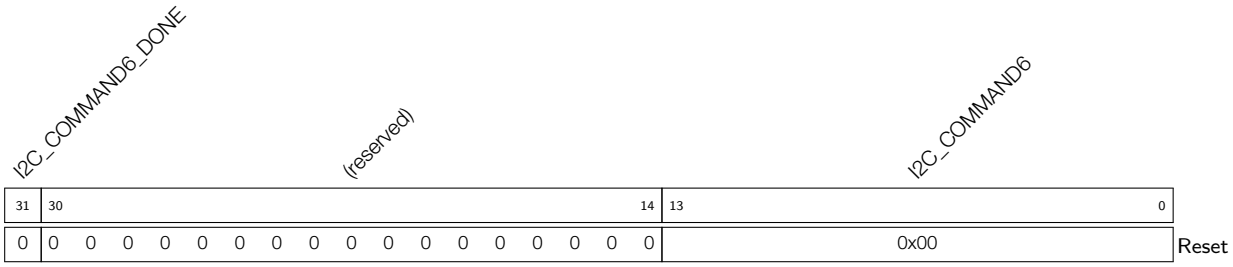
Register 12.31: I2C_COMD5_REG (0x006C)



I2C_COMMAND5 This is the content of command 5. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND5_DONE When command 5 is done in I2C Master mode, this bit changes to high level. (R/W)

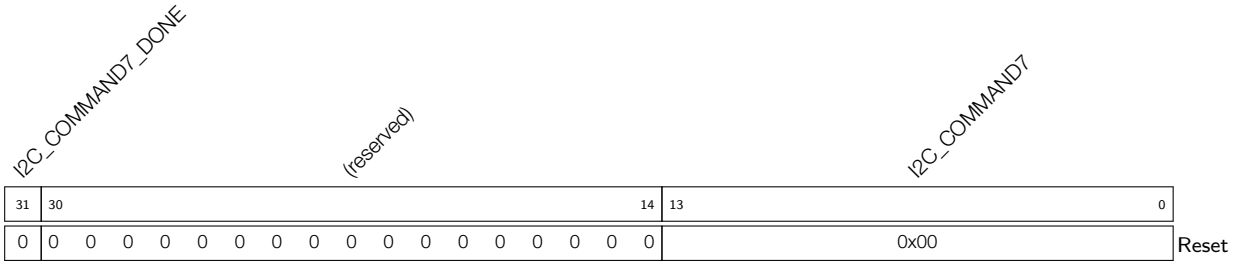
Register 12.32: I2C_COMD6_REG (0x0070)



I2C_COMMAND6 This is the content of command 6. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND6_DONE When command 6 is done in I2C Master mode, this bit changes to high level. (R/W)

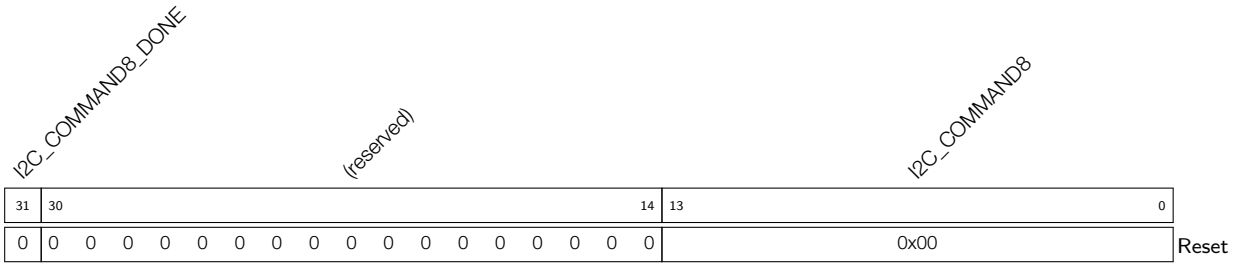
Register 12.33: I2C_COMD7_REG (0x0074)



I2C_COMMAND7 This is the content of command 7. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND7_DONE When command 7 is done in I2C Master mode, this bit changes to high level. (R/W)

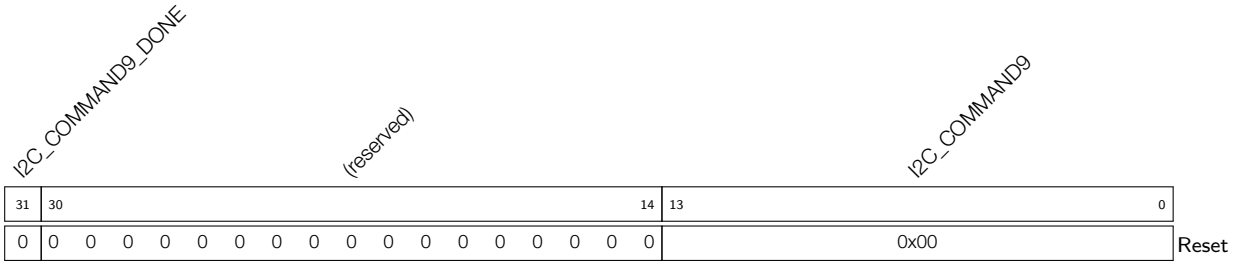
Register 12.34: I2C_COMD8_REG (0x0078)



I2C_COMMAND8 This is the content of command 8. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND8_DONE When command 8 is done in I2C Master mode, this bit changes to high level. (R/W)

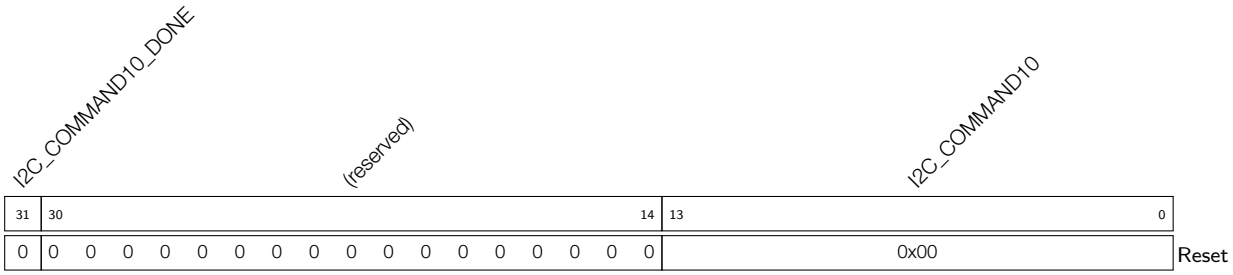
Register 12.35: I2C_COMD9_REG (0x007C)



I2C_COMMAND9 This is the content of command 9. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND9_DONE When command 9 is done in I2C Master mode, this bit changes to high level. (R/W)

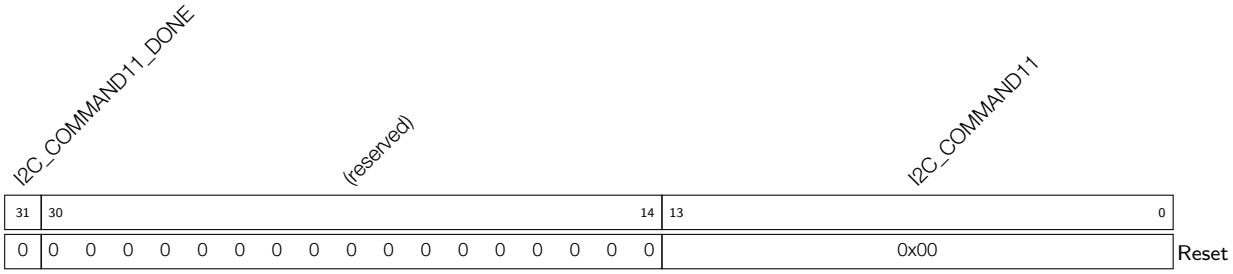
Register 12.36: I2C_COMD10_REG (0x0080)



I2C_COMMAND10 This is the content of command 10. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND10_DONE When command 10 is done in I2C Master mode, this bit changes to high level. (R/W)

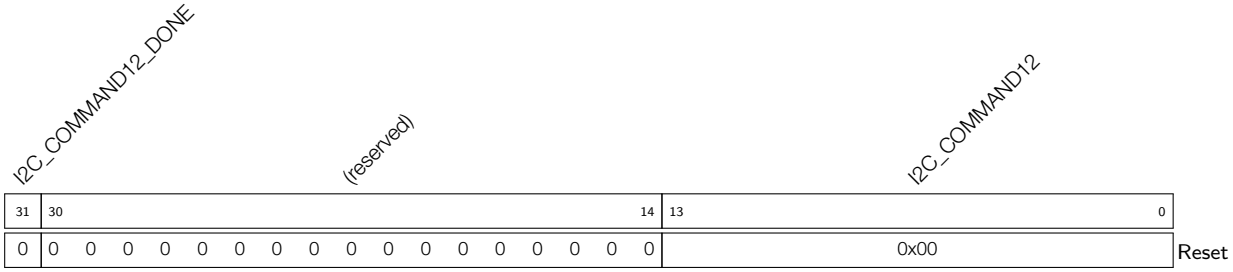
Register 12.37: I2C_COMD11_REG (0x0084)



I2C_COMMAND11 This is the content of command 11. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND11_DONE When command 11 is done in I2C Master mode, this bit changes to high level. (R/W)

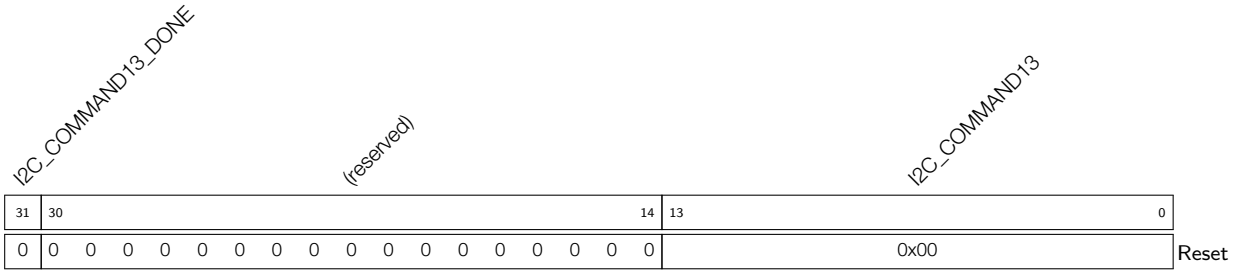
Register 12.38: I2C_COMD12_REG (0x0088)



I2C_COMMAND12 This is the content of command 12. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND12_DONE When command 12 is done in I2C Master mode, this bit changes to high level. (R/W)

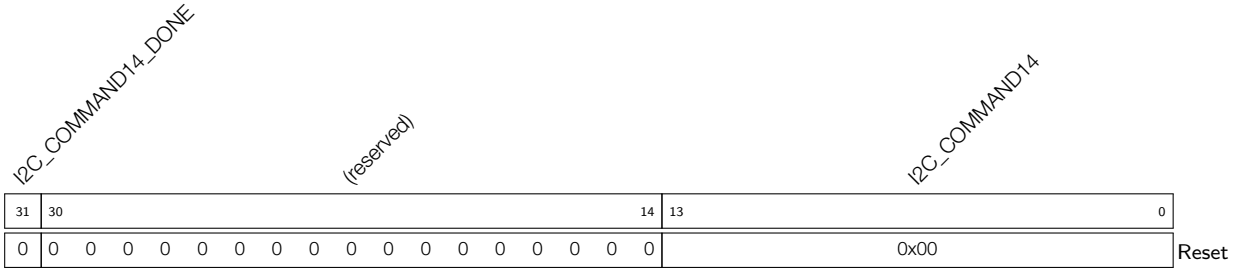
Register 12.39: I2C_COMD13_REG (0x008C)



I2C_COMMAND13 This is the content of command 13. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND13_DONE When command 13 is done in I2C Master mode, this bit changes to high level. (R/W)

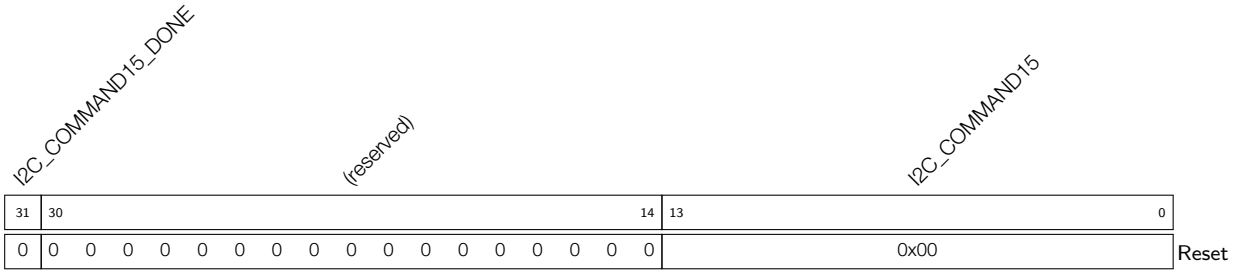
Register 12.40: I2C_COMD14_REG (0x0090)



I2C_COMMAND14 This is the content of command 14. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND14_DONE When command 14 is done in I2C Master mode, this bit changes to high level. (R/W)

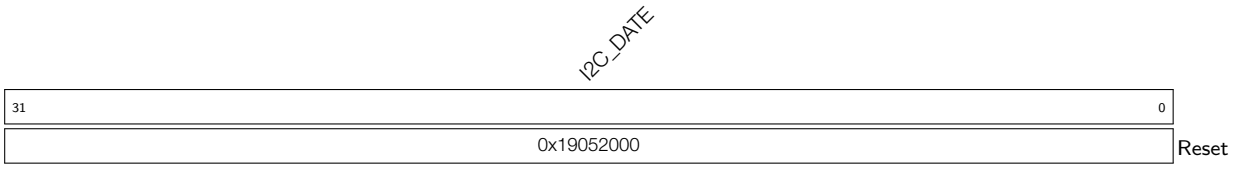
Register 12.41: I2C_COMD15_REG (0x0094)



I2C_COMMAND15 This is the content of command 15. It consists of three parts: op_code is the command, 0: RSTART; 1: WRITE; 2: READ; 3: STOP; 4: END. byte_num represents the number of bytes that need to be sent or received. ack_check_en, ack_exp and ack are used to control the ACK bit. See I2C cmd structure for more Information. (R/W)

I2C_COMMAND15_DONE When command 15 is done in I2C Master mode, this bit changes to high level. (R/W)

Register 12.42: I2C_DATE_REG (0x00F8)



I2C_DATE This is the the version control register. (R/W)

13. AES Accelerator

13.1 Introduction

ESP32-S2 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-S2 has two working modes, which are Typical AES and DMA-AES.

13.2 Features

The following functionality is supported:

- Typical AES working mode
  - AES-128/AES-192/AES-256 encryption and decryption
  - Four variations of key endianness and four variations of text endianness
- DMA-AES working mode
  - Block mode
    - ECB (Electronic Codebook)
    - CBC (Cipher Block Chaining)
    - OFB (Output Feedback)
    - CTR (Counter)
    - CFB8 (8-bit Cipher Feedback)
    - CFB128 (128-bit Cipher Feedback)
  - GCM (Galois/Counter Mode)
  - Interrupt on completion of computation

13.3 Working Modes

The AES Accelerator integrated in ESP32-S2 has two working modes, which are Typical AES and DMA-AES.

- Typical AES Working Mode: supports AES-128/AES-192/AES-256 encryption and decryption under NIST FIPS 197. In this working mode, the plaintext and ciphertext is written and read via CPU directly.
- DMA-AES Working Mode: supports block cipher algorithms ECB/CBC/OFB/CTR/CFB8/CFB128 under NIST SP 800-38A, and GCM mode of operation under NIST SP 800-38D. In this working mode, the plaintext and ciphertext is written and read via crypto 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 13-1 below.

Table 13-1. AES Accelerator Working Mode

AES_DMA_ENABLE_REG	Working Mode
0	Typical AES
1	DMA-AES

For detailed introduction on these two working modes, please refer to Section 13.4 and Section 13.5 below.

Notice:

ESP32-S2's Digital Signature and External Memory Encryption and Decryption modules also call the AES accelerator. Therefore, users cannot access the SHA accelerator when these modules are working.

13.4 Typical AES Working Mode

In the Typical AES working mode, the AES accelerator is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data, i.e. AES-128/AES-192/AES-256 encryption and decryption. Users can choose the operation type for AES accelerator working in Typical AES working mode by configuring the AES_MODE_REG register according to Table 13-2 below.

Table 13-2. Operation Type under Typical AES Working Mode

AES_MODE_REG[2:0]	Operation Type
0	AES-128 encryption
1	AES-192 encryption
2	AES-256 encryption
4	AES-128 decryption
5	AES-192 decryption
6	AES-256 decryption

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 13-3 below.

Table 13-3. 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.

In the Typical AES working mode, the AES accelerator requires 11 ~ 15 clock cycles to encrypt a message block, and 21 or 22 clock cycles to decrypt a message block.

13.4.1 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-192 encryption/decryption, the 192-bit key is stored in AES_KEY_0_REG ~ AES_KEY_5_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-192/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-192/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.

13.4.2 Endianness

Text Endianness

In Typical AES working mode, the AES Accelerator uses cryptograohic keys to encrypt and decrypt data in blocks of 128 bits. The Bit 2 and Bit 3 of the AES_ENDIAN_REG register define the endianness of input text, while the Bit 4 and Bit 5 define the endianness of output text. To be more specific, Bit 2 and Bit 4 control how the four bytes are stored in each word, and Bit 3 and Bit 5 control how the four words are stored in each message block.

Users can choose one of the four text endianness types provided by the AES Accelerator by configuring the AES_ENDIAN_REG register. Details can been seen in Table 13-4.

Table 13-4. Text Endianness Types for Typical AES

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

Note:

1. The definition of "State" is described in Section 3.4 The State in NIST FIPS 197.
2. Where,
   - When *x* = IN, the Word Endian and Byte Endian controlling bits of AES_TEXT_IN_*m*_REG are the Bit 2 and Bit 3 of AES_ENDIAN_REG, respectively;
   - When *x* = OUT, the Word Endian and Byte Endian controlling bits of AES_TEXT_OUT_*m*_REG are the Bit 4 and Bit 5 of AES_ENDIAN_REG, respectively.

Key Endianness

In Typical AES working mode, Bit 0 and bit 1 in AES_ENDIAN_REG define the key endianness.

Users can choose one of the four key endianness types provided by the AES accelerator by configuring the AES_ENDIAN_REG register. Details can been seen in Table 13-5, Table 13-6, and Table 13-7.

Table 13-5. Key Endianness Types for AES-128 Encryption and Decryption

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

Note:

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

Table 13-6. Key Endianness Types for AES-192 Encryption and Decryption

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

Note:

1. w[0] ~ w[5] are “the first Nk words of the expanded key” as specified in Chapter 5.2 Key Expansion in NIST FIPS 197.
2. “Column Bit” specifies the bytes of each word stored in w[0] ~ w[5].

Table 13-7. Key Endianness Types for AES-256 Encryption and Decryption

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

Note:

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

13.4.3 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, and AES_ENDIAN_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, and AES_ENDIAN_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, AES_KEY_*n*_REG, and AES_ENDIAN_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.

13.5 DMA-AES Working Mode

In the DMA-AES working mode, the AES accelerator supports six block operation including ECB/CBC/OFB/CTR /CFB8/CFB128 as well as GCM operation. Users can choose the operation type for AES accelerator working in the DMA-AES working mode by configuring the AES_BLOCK_MODE_REG register according to Table 13-8 below.

Table 13-8. Operation Type under DMA-AES Working Mode

AES_BLOCK_MODE_REG[2:0]	Operation Type
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	GCM (Galois/Counter 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 13-9 below.

Table 13-9. Working Status under DMA-AES Working mode

AES_STATE_REG	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 enabled. Also, note that the interrupt should be cleared by software after use.

13.5.1 Key, Plaintext, and Cipertext

Block Operation

During the block operations, the AES Accelerator reads source data (in_stream) from DMA, and write result data (out_stream) to DMA as well, 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 13-10 below.

Table 13-10. 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**($X$) = $X_1 \| \| X_2 \| \| \cdots \| \| X_{n-1} \| \| X_n^* = X \| \| 0^{128-t}$

13.5.2 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 13-11 below.

Table 13-11. 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

DMA can access both internal memory and PSRAM outside ESP32-S2. When you use DMA to access external

PSRAM, please use base addresses that meet the requirements for DMA. When you use DMA to access internal memory, base addresses do not have such requirements.

13.5.3 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.

13.5.4 Block Number

Register AES_BLOCK_NUM_REG stores the Block Number of plaintext P or cipertext 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.

13.5.5 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 $\cdots$ Byte15 (from left to right). AES_IV_MEM stores data following the Endianness pattern presented in Table 13-11, 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.

13.5.6 Block Operation Process

1. Write 0 to the CRYPTO_DMA_AES_SHA_SELECT_REG register.
2. Configure Crypto DMA chained list and start DMA.
3. 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, AES_KEY_*n*_REG, and AES_ENDIAN_REG.
    - Select operation type by configuring the AES_BLOCK_MODE_REG register. For details, see Table 13-8.
    - Initialize the AES_BLOCK_NUM_REG register. For details, see Section 13.5.4.
    - Initialize the AES_INC_SEL_REG register (only needed when AES Accelerator is working under CTR block operation).
    - Initialize the AES_IV_MEM memory (only needed when AES Accelerator is working under ECB block operation).

4. Start operation by writing 1 to the AES_TRIGGER_REG register.
5. Wait for the completion of computation, which happens when the content of AES_STATE_REG becomes 2 or the AES interrupt occurs.
6. 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.
7. Clear interrupt by writing 1 to the AES_INT_CLR_REG register, if any AES interrupt occured during the computation.
8. 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 5 is completed.

13.5.7 GCM Operation Process

1. Write 0 to the CRYPTO_DMA_AES_SHA_SELECT_REG register.
2. Configure Crypto DMA chained list and start DMA.
3. 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 AES_ENDIAN_REG.
   - Write 6 to the AES_BLOCK_MODE_REG register.
   - Initialize the AES_BLOCK_NUM_REG register. Details about this register are described in Section 13.5.4.
   - Initialize the AES_AAD_BLOCK_NUM_REG register. Details about this register are described in Section 13.6.4.
   - Initialize the AES_REMAINDER_BIT_NUM_REG register. Details about this register is described in Section 13.6.5.
4. Start operation by writing 1 to the AES_TRIGGER_REG register.
5. Wait for the completion of computation, which happens when the content of AES_STATE_REG becomes 2. For details on the working status of AES Accelerator, please refer to Table 13-9. At this step, no interrupt occurs.
6. Obtain the H value from the AES_H_MEM memory.
7. Generate J_0 and write it to the AES_J0_MEM memory.
8. Continue operating by writing 1 to the AES_CONTINUE_REG register.
9. Wait for the completion of computation, which happens when the content of AES_STATE_REG becomes 2 or the AES interrupt occurs. For details on the working status of AES Accelerator, please refer to Table 13-9.
10. Obtain T_0 by reading AES_T0_MEM, which is already ready at this step.
11. 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.

12. Clear interrupt by writting 1 to the AES_INT_CLR_REG register, if any AES interrupt occured during the computation.
13. Exit DMA by writing 1 to the AES_DMA_EXIT_REG register. After this, the content of the AES_STATE_REG becomes 0. Note that, you can exit DMA earlier, but only after Step 9 is completed.

13.6 GCM Algorithm

ESP32-S2's AES accelerator fully supports GCM Algorithm. In reality, the AAD, C and P that are longer than 2^{32}-1 bits are seldom used. Therefore, we specify that the length of AAD, C and P should be no longer than 2^{32}-1 here. Figure 13-1 below demonstrates how GCM encryption is implemented in the AES Accelerator of ESP32-S2.

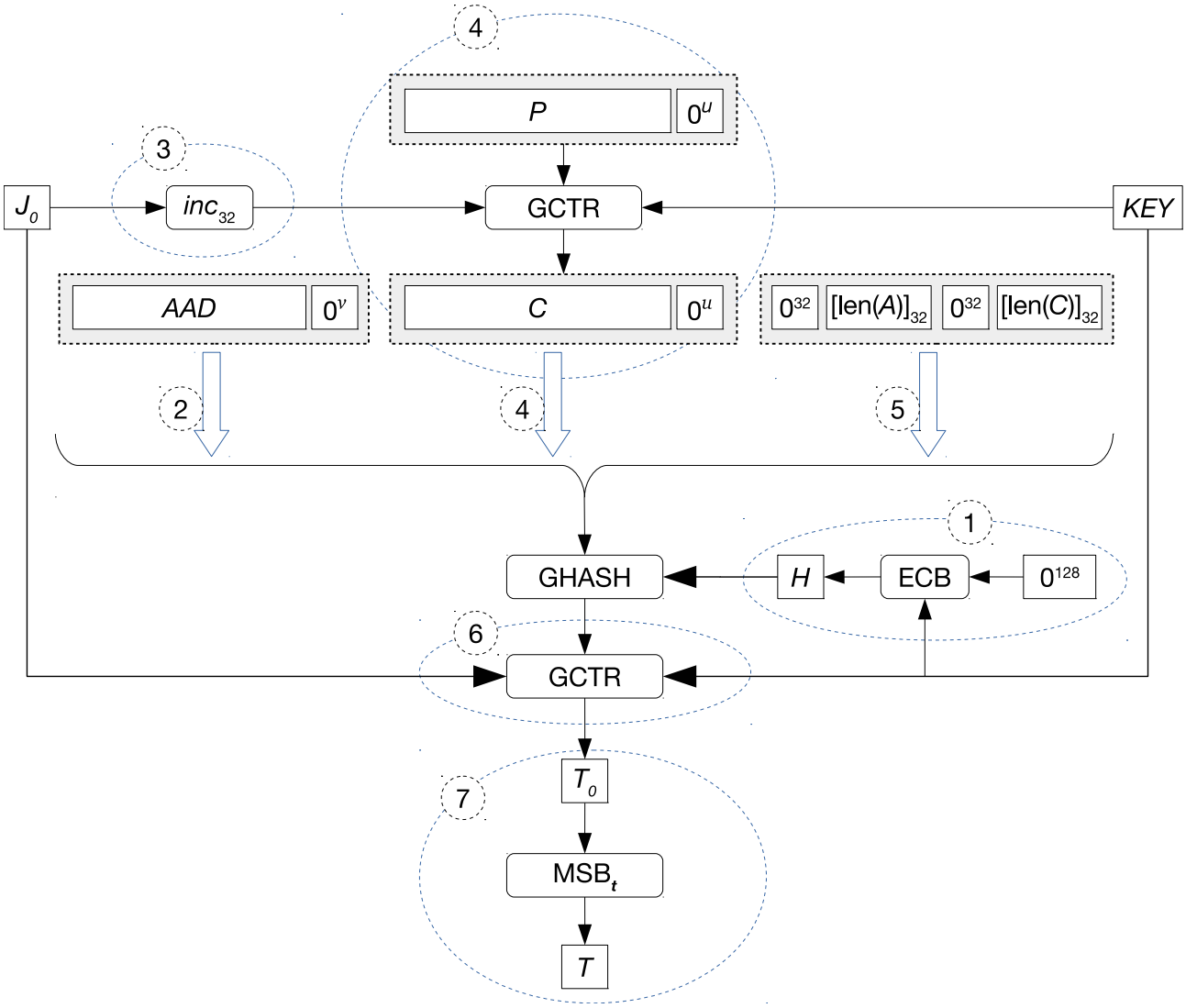


Figure 13-1. GCM Encryption Process

GCM encryption can be implemented as follows:

1. Hardware executes the ECB Algorithm to obtain the Hash subkey H, which is needed in the Hash computation.
2. Hardware executes the GHASH Algorithm to perform Hash computation with the padded AAD.
3. Hardware gets ready for CTR encryption by obtaining the result of applying Standard Incrementing Function INC_{32} to J_0.
4. Hardware executes the GCTR Algorithm to encrypt the padded plaintext P, then executes the GHASH Algorithm to perform Hash computation on the padded cipertext C.

5. Hardware executes the GHASH Algorithm to perform Hash computation on AAD Blocks, obtaining a 128-bit Hash result.
6. Hardware executes the GCTR Algorithm to encrypt J_0, obtaining T_0.
7. Software obtains the result T_0 from hardware, and execute MSB_t Algorithm to obtain the final result Authenticated Tag T.

The only difference between GCM decryption and GCM encryption lies in Step 4 in Figure 13-1. To be more specific, instead of executing GCTR Algorithm to encrypt the padded plaintext, the AES Accelerator executes the same Algorithm to decrypt the padded cipertext in GCM decryption. For details, please see NIST SP 800-38D.

13.6.1 Hash Subkey

During GCM operation, the Hash subkey H is a 128-bit value computed by hardware, which is demonstrated in Step 1 in Figure 13-1. Also you can find more information about Hash subkey at "Step 1. Let $H = \text{CIPH}_K(0^{128})$" in Chapter 7 GCM Specification of NIST SP 800-38D.
Just like all other Endianness, the Hash subkey H is stored in the AES_H_MEM memory with its most significant (i.e., left-most) byte Byte0 stored at the lowest address and least significant (i.e., right-most) byte Byte15 at the highest address. For details, see Table 13-11.

13.6.2 J_0

The J_0 is a 128-bit value computed by hardware, which is required during Step 3 and Step 6 of the GCM process in Figure 13-1. For details on the generation of J_0, please see Chapter 7 GCM Specification in NIST SP 800-38D Specification.
The J_0 is stored in the AES_J0_MEM memory as Endianness. Just like all other Endianness, its most significant (i.e., left-most) byte Byte0 is stored at the lowest address in the memory while least significant (i.e., right-most) byte Byte15 at the highest address. For details, see Table 13-11.

13.6.3 Authenticated Tag

Authenticated Tag (Tag for short) is one of the key results of GCM computation, which is demonstrated in Step 7 of Figure 13-1. The value of Tag is determined by the length of Authenticated Tag t ($1 <= t <= 128$):

- When $t = 128$, the value of Tag equals to T_0, a 128-bit string that is stored in the AES_T0_MEM as Endianness. Just like all other Endianness, its most significant (i.e., left-most) byte Byte0 is stored at the lowest address in the memory while least significant (i.e., right-most) byte Byte15 at the highest address. For details, see Table 13-11.
- When $1 <= t < 128$, the value of Tag equals to the t most significant (i.e., left-most) bits of T_0. In this case, Tag is represented as $\text{MSB}_t(T_0)$, which returns the t most significant bits of T_0. For example, $\text{MSB}_4(111011010) = 1110$ and $\text{MSB}_5(11010011010) = 11010$. For details on the $\text{MSB}_t()$ function, please refer to Chapter 6 Mathematical Components of GCM in the NIST SP 800-38D specification.

13.6.4 AAD Block Number

Register AES_AAD_BLOCK_NUM_REG stores the Block Number of Additional Authenticated Data (AAD). The length of this register equals to length(**TEXT-PADDING**(AAD))/128. AES Accelerator only uses this register when

working in the DMA-AES mode.

13.6.5 Remainder Bit Number

Register AES_REMAINDER_BIT_NUM_REG stores the Remainder Bit Number, which indicates the number of effective bits of incomplete blocks in plaintext/cipertext. The value stored in this register equals to length(P)%128 or length(C)%128. AES Accelerator only uses this register when working in the DMA-AES mode.

Register AES_REMAINDER_BIT_NUM_REG does not affect the results of plaintext or cipertext, but does impact the value of T_0, therefore the Tag value too.

The GCM Algorithm can be viewed as the combination of GCTR operation and GHASH operation, among which, the GCTR performs the encryption and decryption, while the GHASH sloves the Tag.

Note that the AES_REMAINDER_BIT_NUM_REG register is only effective for GCM encryption. To be more specific:

- For GCM encryption, the Hardware firstly computes C, then passes it in the form of **TEXT-PADDING**(C) as the input of GHASH operation. In this case, hardware determines how many trailing "0" should be added based on the content of AES_REMAINDER_BIT_NUM_REG.
- For GCM decrpytion, the padding is completed with the **TEXT-PADDING**(C) function. In this case, the AES_REMAINDER_BIT_NUM_REG register is not effective.

13.7 Base Address

Users can access AES with two base addresses, which can be seen in Table 13-12. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 13-12. AES Accelerator Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F43A000
PeriBUS2	0x6003A000

13.8 Memory Summary

Both the starting address and ending address in the following table are relative to AES base addresses provided in Section 13.7.

Name	Description	Length	Starting Address	Ending Address	Access
AES_IV_MEM	Memory IV	16 bytes	0x0050	0x005F	R/W
AES_H_MEM	Memory H	16 bytes	0x0060	0x006F	RO
AES_J0_MEM	Memory J0	16 bytes	0x0070	0x007F	R/W
AES_T0_MEM	Memory T0	16 bytes	0x0080	0x008F	RO

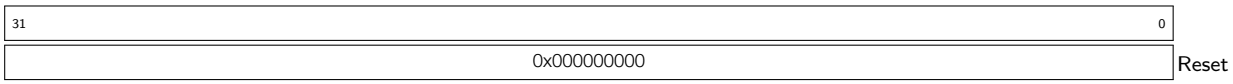
13.9 Register Summary

The addresses in the following table are relative to AES base addresses provided in Section 13.7.

Name	Description	Address	Access
Key Registers			
AES_KEY_0_REG	AES key register 0	0x0000	R/W
AES_KEY_1_REG	AES key register 1	0x0004	R/W
AES_KEY_2_REG	AES key register 2	0x0008	R/W
AES_KEY_3_REG	AES key register 3	0x000C	R/W
AES_KEY_4_REG	AES key register 4	0x0010	R/W
AES_KEY_5_REG	AES key register 5	0x0014	R/W
AES_KEY_6_REG	AES key register 6	0x0018	R/W
AES_KEY_7_REG	AES key register 7	0x001C	R/W
TEXT_IN Registers			
AES_TEXT_IN_0_REG	Source data register 0	0x0020	R/W
AES_TEXT_IN_1_REG	Source data register 1	0x0024	R/W
AES_TEXT_IN_2_REG	Source data register 2	0x0028	R/W
AES_TEXT_IN_3_REG	Source data register 3	0x002C	R/W
TEXT_OUT Registers			
AES_TEXT_OUT_0_REG	Result data register 0	0x0030	RO
AES_TEXT_OUT_1_REG	Result data register 1	0x0034	RO
AES_TEXT_OUT_2_REG	Result data register 2	0x0038	RO
AES_TEXT_OUT_3_REG	Result data register 3	0x003C	RO
Configuration Registers			
AES_MODE_REG	AES working mode configuration register	0x0040	R/W
AES_ENDIAN_REG	Endian configuration register	0x0044	R/W
AES_DMA_ENABLE_REG	DMA enable register	0x0090	R/W
AES_BLOCK_MODE_REG	Block operation type register	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
AES_AAD_BLOCK_NUM_REG	AAD block number configuration register	0x00A0	R/W
AES_REMAINDER_BIT_NUM_REG	Remainder bit number of plaintext/ciphertext	0x00A4	R/W
Controlling / Status Registers			
AES_TRIGGER_REG	Operation start controlling register	0x0048	WO
AES_STATE_REG	Operation status register	0x004C	RO
AES_CONTINUE_REG	Operation continue controlling register	0x00A8	WO
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

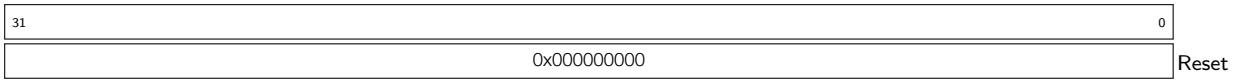
13.10 Registers

Register 13.1: AES_KEY_*n*_REG (*n*: 0-7) (0x0000+4n*)**



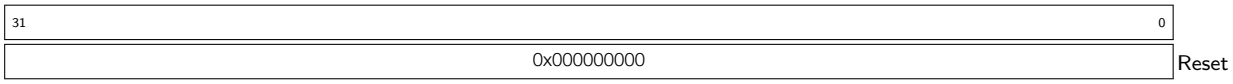
AES_KEY_*n*_REG (*n*: 0-7) Stores AES keys. (R/W)

Register 13.2: AES_TEXT_IN_*m*_REG (*m*: 0-3) (0x0020+4m*)**



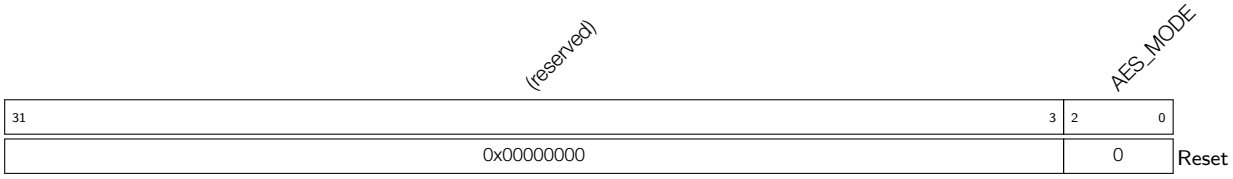
AES_TEXT_IN_*m*_REG (*m*: 0-3) Stores the source data when the AES Accelerator operates in the Typical AES working mode. (R/W)

Register 13.3: AES_TEXT_OUT_*m*_REG (*m*: 0-3) (0x0030+4m*)**



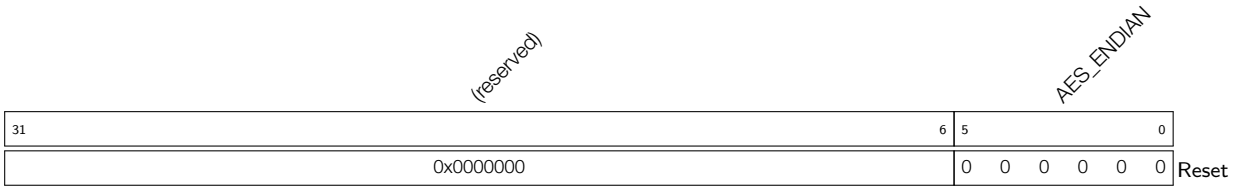
AES_TEXT_OUT_*m*_REG (*m*: 0-3) Stores the result data when the AES Accelerator operates in the Typical AES working mode. (RO)

Register 13.4: AES_MODE_REG (0x0040)



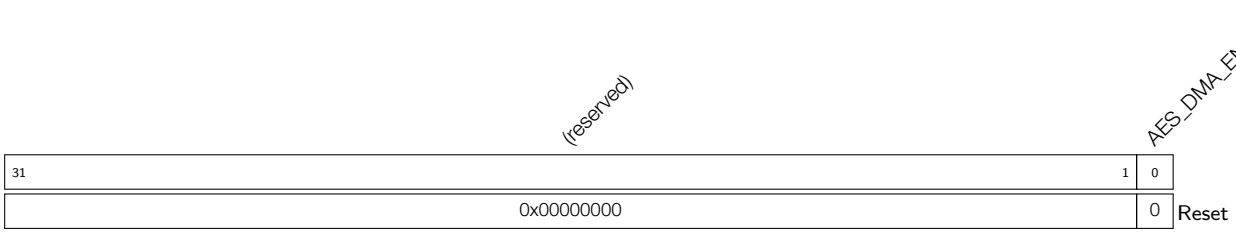
AES_MODE Defines the operation type of the AES Accelerator operating under the Typical AES working mode. For details, see Table 13-2. (R/W)

Register 13.5: AES_ENDIAN_REG (0x0044)



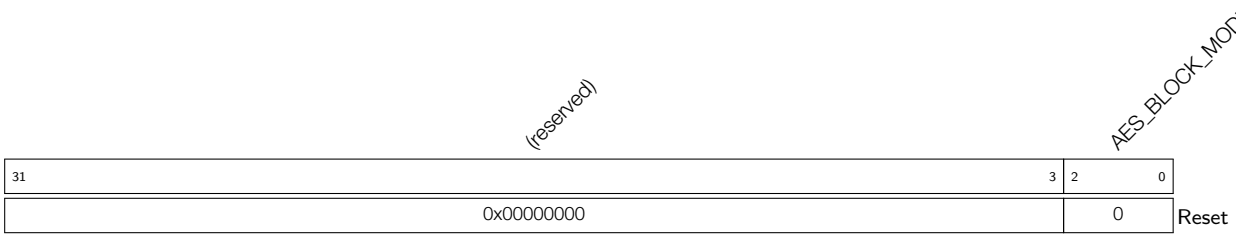
AES_ENDIAN Defines the endianness of input and output texts. For details, please see Table 13-4. (R/W)

Register 13.6: AES_DMA_ENABLE_REG (0x0090)



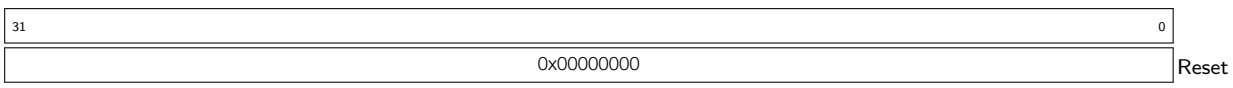
AES_DMA_ENABLE Defines the working mode of the AES Accelerator. For details, see Table 13-1. (R/W)

Register 13.7: AES_BLOCK_MODE_REG (0x0094)



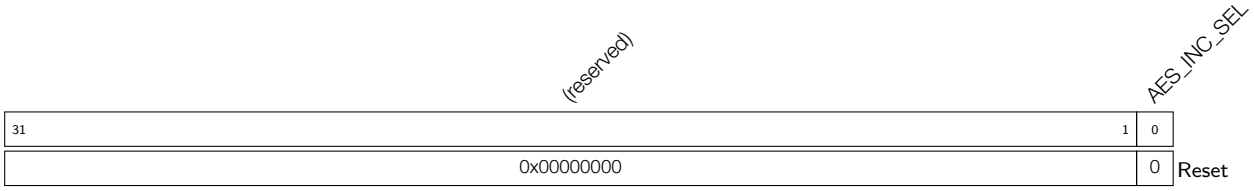
AES_BLOCK_MODE Defines the operation type of the AES Accelerator operating under the DMA-AES working mode. For details, see Table 13-8. (R/W)

Register 13.8: AES_BLOCK_NUM_REG (0x0098)



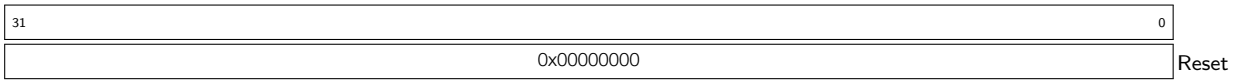
AES_BLOCK_NUM Stores the Block Number of plaintext or cipertext when the AES Accelerator operates under the DMA-AES working mode. For details, see Section 13.5.4. (R/W)

Register 13.9: AES_INC_SEL_REG (0x009C)



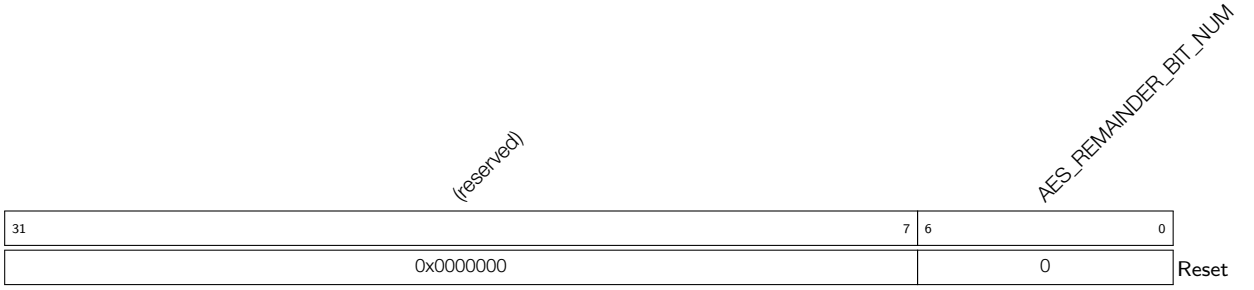
AES_INC_SEL Defines the Standard Incrementing Function for CTR block operation. Set this bit to 0 or 1 to choose INC_{32} or INC_{128}. (R/W)

Register 13.10: AES_AAD_BLOCK_NUM_REG (0x00A0)



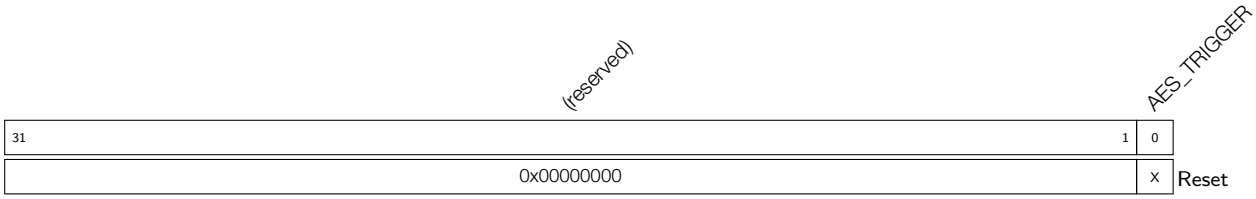
AES_AAD_BLOCK_NUM Stores the ADD Block Number for the GCM operation. (R/W) For details, see Section 13.6.4.

Register 13.11: AES_REMAINDER_BIT_NUM_REG (0x00A4)



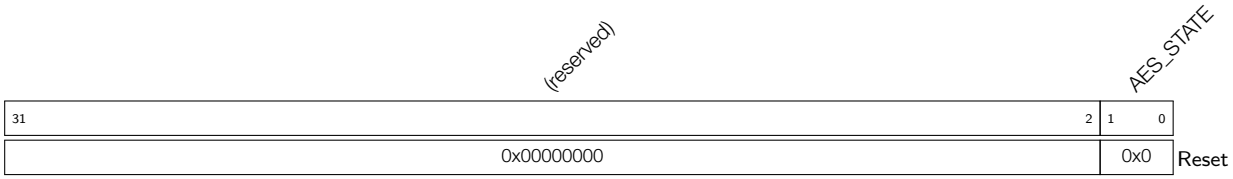
AES_REMAINDER_BIT_NUM Stores the Remainder Bit Number for the GCM operation. For details, see Section 13.6.5. (R/W)

Register 13.12: AES_TRIGGER_REG (0x0048)



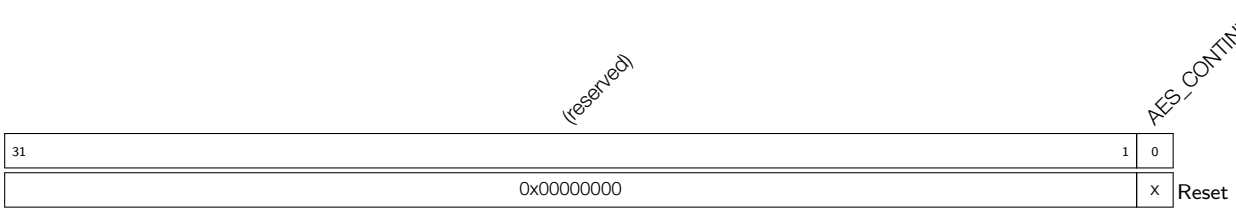
AES_TRIGGER Set this bit to 1 to start AES operation. (WO)

Register 13.13: AES_STATE_REG (0x004C)



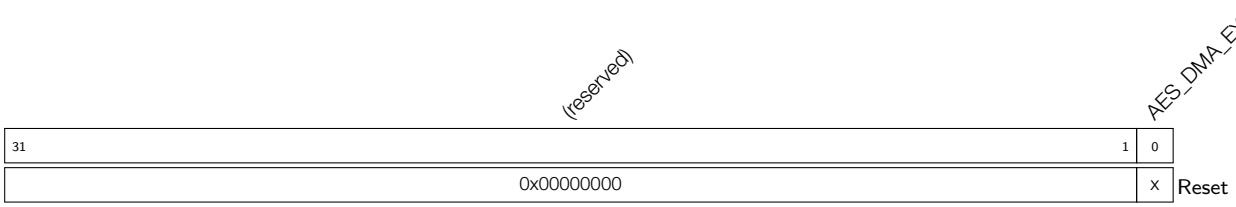
AES_STATE Stores the working status of the AES Accelerator. For details, see Table 13-3 for Typical AES working mode and Table 13-9 for DMA AES working mode. (RO)

Register 13.14: AES_CONTINUE_REG (0x00A8)



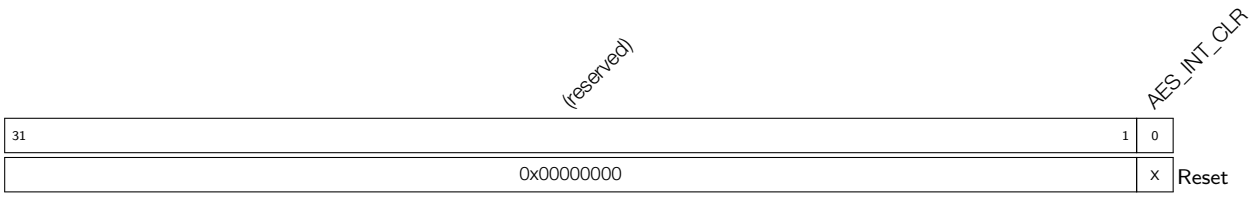
AES_CONTINUE Set this bit to 1 to continue AES operation. (WO)

Register 13.15: AES_DMA_EXIT_REG (0x00B8)



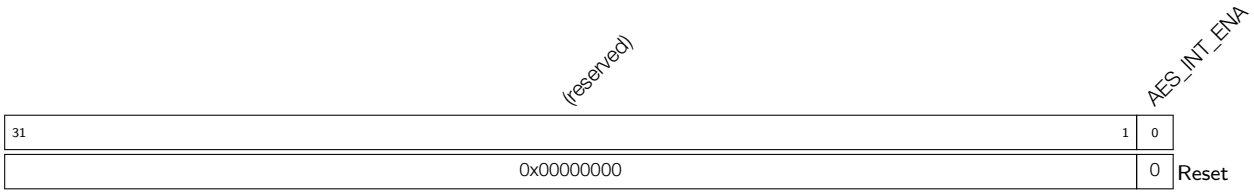
AES_DMA_EXIT Set this bit to 1 to exit AES operation. This register is only effective for DMA-AES operation. (WO)

Register 13.16: AES_INT_CLR_REG (0x00AC)



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

Register 13.17: AES_INT_ENA_REG (0x00B0)



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

14. SHA Accelerator

14.1 Introduction

ESP32-S2 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-S2 has two working modes, which are Typical SHA and DMA-SHA.

14.2 Features

The following functionality is supported:

- All the hash algorithms introduced in FIPS PUB 180-4 Spec.
  - SHA-1
  - SHA-224
  - SHA-256
  - SHA-384
  - SHA-512
  - SHA-512/224
  - SHA-512/256
  - SHA-512/*t*
- Two working modes
  - Typical SHA
  - DMA-SHA
- Interleave function when working in Typical SHA working mode
- Interrupt function when working in DMA-SHA working mode

14.3 Working Modes

The SHA accelerator integrated in ESP32-S2 has two working modes: Typical SHA and DMA-SHA.

- Typical SHA Working Mode: all the data is written and read via CPU directly.
- DMA-SHA Working Mode: all the data is read via crypto 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 14-1.

Table 14-1. 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 14-2.

Table 14-2. SHA Hash Algorithm

Hash Algorithm	SHA_MODE_REG Configuration
SHA-1	0
SHA-224	1
SHA-256	2
SHA-384	3
SHA-512	4
SHA-512/224	5
SHA-512/256	6
SHA-512/t	7

Notice:

ESP32-S2's Digital Signature and HMAC modules also call the SHA accelerator. Therefore, users cannot access the SHA accelerator when these modules are working.

14.4 Function Description

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

14.4.1 Preprocessing

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

14.4.1.1 Padding the Message

The SHA accelerator can only process message blocks of 512 or 1024 bits, depending on the algorithm. Thus, all the messages should be padded to a multiple of 512 or 1024 bits before the hash computation.

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

- **SHA-1**, **SHA-224** and **SHA-256**
  1. First, append the bit "1" to the end of the message;

2. Second, append k zero bits, 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 that is equal to the number m expressed using a binary representation.

- **SHA-384**, **SHA-512**, **SHA-512/224**, **SHA-512/256** and **SHA-512/**t
  1. First, append the bit "1" to the end of the message;
  2. Second, append k zero bits, where k is the smallest, non-negative solution to the equation $m + 1 + k \equiv 896 \ mod \ 1024$;
  3. Last, append the 128-bit block that is 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.

14.4.1.2 Parsing the Message

The message and its padding must be parsed into N 512-bit or 1024-bit blocks.

- For **SHA-1**, **SHA-224** and **SHA-256**: the message and its padding are 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 $M_{15}^{(i)}$.
- For **SHA-384**, **SHA-512**, **SHA-512/224**, **SHA-512/256** and **SHA-512/**t: the message and its padding are parsed into N 1024-bit blocks. Since the 1024 bits of the input block may be expressed as sixteen 64-bit words, the first 64 bits of message block i are denoted $M_0^{(i)}$, the next 64 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

In Typical SHA working mode, all the message blocks are written into the SHA_M_n_REG, following the rules below:

- For **SHA-1, SHA-224 and SHA-256**: $M_0^{(i)}$ 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.
- For **SHA-384, SHA-512, SHA-512/224 and SHA-512/256**: the most significant 32 bits and the least significant 32 bits of $M_0^{(i)}$ are stored in SHA_M_0_REG and SHA_M_1_REG, respectively, ..., the most significant 32 bits and the least significant 32 bits of $M_{15}^{(i)}$ are stored in SHA_M_30_REG and SHA_M_31_REG, respectively.

Note:

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

In DMA-SHA working mode, please complete the following configuration:

1. Create an external linked list;
2. Configure this linked list based on the instruction described in Chapter *DMA Controller*, including but not limited to assigning the starting address of the input message to the buffer address pointer of the linked list;
3. Configure the CRYPTO_DMA_OUTLINK_ADDR to the first out-link linked list;
4. Write 1 to register CRYPTO_DMA_OUTLINK_START, so the DMA starts to move data;

5. Write 1 to register CRYPTO_DMA_AES_SHA_SELECT_REG, so the SHA accelerator gets to use the DMA resource shared by AES and SHA accelerators.

14.4.1.3 Initial Hash Value

Before hash computation begins for each of the secure hash algorithms, the initial Hash value $H^{(0)}$ must be set based on different algorithms, among which the SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 algorithms use the initial Hash values (constant C) stored in the hardware.

However, SHA-512/t requires a distinct initial hash value for each operation for a given value of t. Simply put, SHA-512/t is the generic name for a t-bit hash function based on SHA-512 whose output is truncated to t bits. t is any positive integer without a leading zero such that t<512, and t is not 384. The initial hash value for SHA-512/t for a given value of t can be calculated by performing SHA-512 from hexadecimal representation of the string "SHA-512/t". It's not hard to observe that when determining the initial hash values for SHA-512/t algorithms with different t, the only difference lies in the value of t.

Therefore, we have specially developed the following simplified method to calculate the initial hash value for SHA-512/t:

1. **Generate t_string and t_length**: t_string is a 32-bit data that stores the input message of t. t_length is a 7-bit data that stores the length of the input message. The t_string and t_length are generated in methods described below, depending on the value of t:

   - If $1 <= t <= 9$, then $t\_length = 7'h48$ and t_string is padded in the following format:

$8'h3t_0$	$1'b1$	$23'b0$

     where $t_0 = t$.

     For example, if t = 8, then t_0 = 8 and t_string = $32'h38800000$.

   - If $10 <= t <= 99$, then $t\_length = 7'h50$ and t_string is padded in the following format:

$8'h3t_1$	$8'h3t_0$	$1'b1$	$15'b0$

     where, $t_0 = t\%10$ and $t_1 = t/10$.

     For example, if t = 56, then t_0 = 6, t_1 = 5, and t_string = $32'h35368000$.

   - If $100 <= t < 512$, then $t\_length = 7'h58$ and t_string is padded in the following format:

$8'h3t_2$	$8'h3t_1$	$8'h3t_0$	$1'b1$	$7'b0$

     where, $t_0 = t\%10$, $t_1 = (t/10)\%10$, and $t_2 = t/100$.

     For example, if t = 231, then t_0 = 1, t_1 = 3, t_2 = 2, and t_string = $32'h32333180$.

2. **Initialize relevant registers**: Initialize SHA_T_STRING_REG and SHA_T_LENGTH_REG with the generated t_string and t_length in the previous step.

3. **Obtain initial hash value**: Set the SHA_MODE_REG register to 7. Set the SHA_START_REG register to 1 to start the SHA accelerator. Then poll register SHA_BUSY_REG until the content of this register becomes 0, indicating the calculation of initial hash value is completed.

Please note that the initial value for SHA-512/t can be also calculated according to the Section "5.3.6 SHA-512/t" in FIPS PUB 180-4 Spec, that is performing SHA-512 operation (with its initial hash value set to the result of 8-bitwise XOR operation of C and 0xa5) from the hexadecimal representation of the string "SHA-512/t".

14.4.2 Hash Computation Process

After the preprocessing, the ESP32-S2 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-S2 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.

14.4.2.1 Typical SHA Process

ESP32-S2 SHA accelerator supports "interleave" functionality when working under Typical SHA mode:

- **Type "alone"**: Users do not insert any new computation before the SHA accelerator completes all the message blocks.
- **Type "interleave"**: Users can insert new computations (both Typical SHA task and DMA-SHA task) every time the SHA accelerator completes one message block. To be more specific, users can store the message digest in registers SHA_H_*n*_REG after completing each message block, and assign the accelerator with other higher priority tasks. After the inserted task completes, users can put the message digest stored back to registers SHA_H_*n*_REG, and resume the accelerator with the previously paused computation.

Typical SHA Process (except for SHA-512/t)

1. Select a hash algorithm.
   - Select a hash algorithm by configuring the SHA_MODE_REG register. For details, please refer to Table 14-2.
2. Process the current message block.
   (a) Write the current message block in registers SHA_M_*n*_REG;
   (b) Start the SHA accelerator [1]:
      - 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 computation;
      - If this is not the first time to execute this step, 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 computation.

   (c) Poll register SHA_BUSY_REG until the content of this register becomes 0, indicating the accelerator has completed the computation for the current message block and now is in the "idle" status. Then, go to step 3.
3. Decide if you want to insert other computations.
   - If yes, please get ready for handing over the SHA accelerator for the new task:

(a) Read and store the hash algorithm selected for the current computation stored in the SHA_MODE_REG register;

(b) Read and store the message digests stored in registers SHA_H_*n*_REG;

(c) Last, please go to perform the inserted computation. For the detailed process of the inserted computation, please refer to Typical SHA or DMA-SHA, depending on the working mode.

- Otherwise, please continue to execute Step 4.

4. Decide if you have more message blocks following the previous computation:
   - If yes, please go back to 2.
   - Otherwise, go to Step 5.
5. Decide if you need to return the SHA accelerator to a previous computation (i.e., decide whether the current task is an inserted task or not):
   - If yes, please get ready to return the SHA accelerator for the previous computation:
     (a) Write the previously stored hash algorithm back to register SHA_MODE_REG;
     (b) Write the previously stored message digests back to registers SHA_H_*n*_REG;
     (c) Then, go to Step 2.
   - Otherwise, there is no need to return SHA Control. Therefore, please go to Step 6 directly.
6. Obtain the message digests:
   - Read the message digests from registers SHA_H_*n*_REG.

Typical SHA Process (SHA-512/*t*)

1. Select a hash algorithm.
   - Select SHA-512/*t* algorithm by configuring the SHA_MODE_REG register to 7.
2. Calculate the initial hash value.
   (a) Calculate t_stiring and t_length and initialize SHA_T_STRING_REG and SHA_T_LENGTH_REG with the generated t_string and t_length. For details, please refer to Section 14.4.1.3.
   (b) Set the SHA_START_REG register to 1 to start the SHA accelerator.
   (c) Poll register SHA_BUSY_REG until the content of this register becomes 0, indicating the calculation of initial hash value is completed.
3. Process the current message block.
   (a) Write the current message block in registers SHA_M_*n*_REG;
   (b) Start the SHA accelerator [1]:
   - 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 computation.

(c) Poll register SHA_BUSY_REG until the content of this register becomes 0, indicating the accelerator has completed the computation for the current message block and now is in the "idle" status. Then, go to step 4.

4. Decide if you want to insert other computations.
   - If yes, please get ready for handing over the SHA accelerator for the new task:
     (a) Read and store the hash algorithm selected for the current computation stored in the SHA_MODE_REG register;
     (b) Read and store the message digests stored in registers SHA_H_*n*_REG;
     (c) Last, please go to perform the inserted computation. For the detailed process of the inserted computation, please refer to Typical SHA or DMA-SHA, depending on the working mode.
   - Otherwise, please continue to execute Step 5.
5. Decide if you have more message blocks following the previous computation:
   - If yes, please go back to 3.
   - Otherwise, go to Step 6.
6. Decide if you need to return the SHA accelerator to a previous computation (i.e., decide whether the current task is an inserted task or not):
   - If yes, please get ready to return the SHA accelerator for the previous computation:
     (a) Write the previously stored hash algorithm back to register SHA_MODE_REG;
     (b) Write the previously stored message digests back to registers SHA_H_*n*_REG;
     (c) Then, go to Step 3.
   - Otherwise, there is no need to return SHA Control. Therefore, please go to Step 7 directly.
7. Obtain the message digests:
   - Read the message digests from registers SHA_H_*n*_REG.

Note:

1. In Step 2b, 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 computation, to save time.

14.4.2.2 DMA-SHA Process

ESP32-S2 SHA accelerator does not support type "interleave" computation, which means you cannot insert new computation before the whole DMA-SHA process completes. In this mode, users who need task insertion are recommended to divide your message blocks and perform several DMA-SHA computations, instead of trying to compute all the messages in one go.

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 crypto DMA.

Therefore, users are required to configure the DMA controller as instructed in Subsection 14.4.1.2.

DMA-SHA process (except SHA-512/t)

1. Select a hash algorithm.
   - Select a hash algorithm by configuring the SHA_MODE_REG register. For details, please refer to Table 14-2.
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 computation.
   - If the current DMA-SHA computation follows a previous computation, firstly write the message digest from the previous computation 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 computation, 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 digests:
   - Read the message digests from registers SHA_H_n_REG.

DMA-SHA process for SHA-512/t

1. Select a hash algorithm.
   - Select SHA-512/t algorithm by configuring the SHA_MODE_REG register to 7.
2. Configure the SHA_INT_ENA_REG register to enable or disable interrupt (Set 1 to enable).
3. Calculate the initial hash value.
   (a) Calculate t_stiring and t_length and initialize SHA_T_STRING_REG and SHA_T_LENGTH_REG with the generated t_string and t_length. For details, please refer to Section 14.4.1.3.
   (b) Set the SHA_START_REG register to 1 to start the SHA accelerator.
   (c) Poll register SHA_BUSY_REG until the content of this register becomes 0, indicating the calculation of initial hash value is completed.
4. Configure the number of message blocks.
   - Write the number of message blocks M to the SHA_DMA_BLOCK_NUM_REG register.
5. Start the DMA-SHA computation.
   - Write 1 to register SHA_DMA_CONTINUE_REG to start the accelerator.
6. Wait till the completion of the DMA-SHA computation, 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.

7. Obtain the message digests:
   - Read the message digests from registers SHA_H_*n*_REG.

14.4.3 Message Digest

After the hash computation completes, the SHA accelerator writes the message digests from the computation to registers SHA_H_*n*_REG(*n*: 0~15). The lengths of the generated message digest are different depending on different hash algorithms. For details, see Table 14-6 below:

Table 14-6. The Storage and Length of Message Digests 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
SHA-384	384	SHA_H_0_REG ~ SHA_H_11_REG
SHA-512	512	SHA_H_0_REG ~ SHA_H_15_REG
SHA-512/224	224	SHA_H_0_REG ~ SHA_H_6_REG
SHA-512/256	256	SHA_H_0_REG ~ SHA_H_7_REG
SHA-512/*t* [2]	*t*	SHA_H_0_REG ~ SHA_H_*x*_REG

Note:

1. The message digests are 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 14.4.1.2.
2. The registers used for SHA-512/*t* algorithm depend on the value of t. *x*+1 indicates the number of 32-bit registers used to store *t* bits of message digest, so that *x* = $roundup(t/32)$-1. For example:
   - When *t* = 8, then *x* = 0, indicating that the 8-bit long message digest is stored in the most significant 8 bits of register SHA_H_0_REG;
   - When *t* = 32, then *x* = 0, indicating that the 32-bit long message digest is stored in register SHA_H_0_REG;
   - When *t* = 132, then *x* = 4, indicating that the 132-bit long message digest is stored in registers SHA_H_0_REG, SHA_H_1_REG, SHA_H_2_REG, SHA_H_3_REG, and SHA_H_4_REG.

14.4.4 Interrupt

SHA accelerator supports interrupt on the completion of computation 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.

14.5 Base Address

Users can access SHA with two base addresses, which can be seen in Table 14-7. For more information about accessing peripherals from different buses, please see Chapter 1 *System and Memory*.

Table 14-7. Base Address

Bus	Base Address
PeriBUS1	0x3F43B000
PeriBUS2	0x6003B000

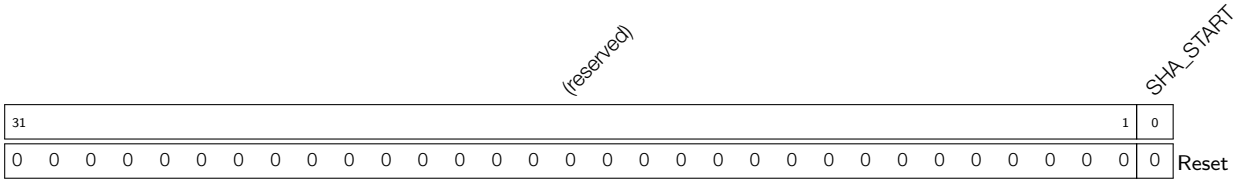
14.6 Register Summary

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
SHA_T_STRING_REG	String content register for calculating initial Hash Value (only effective for SHA-512/t)	0x0004	R/W
SHA_T_LENGTH_REG	String length register for calculating initial Hash Value (only effective for SHA-512/t)	0x0008	R/W
Memories			
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

Name	Description	Address	Access
SHA_H_8_REG	Hash value	0x0060	R/W
SHA_H_9_REG	Hash value	0x0064	R/W
SHA_H_10_REG	Hash value	0x0068	R/W
SHA_H_11_REG	Hash value	0x006C	R/W
SHA_H_12_REG	Hash value	0x0070	R/W
SHA_H_13_REG	Hash value	0x0074	R/W
SHA_H_14_REG	Hash value	0x0078	R/W
SHA_H_15_REG	Hash value	0x007C	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
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
SHA_M_16_REG	Message	0x00C0	R/W
SHA_M_17_REG	Message	0x00C4	R/W
SHA_M_18_REG	Message	0x00C8	R/W
SHA_M_19_REG	Message	0x00CC	R/W
SHA_M_20_REG	Message	0x00D0	R/W
SHA_M_21_REG	Message	0x00D4	R/W
SHA_M_22_REG	Message	0x00D8	R/W
SHA_M_23_REG	Message	0x00DC	R/W
SHA_M_24_REG	Message	0x00E0	R/W
SHA_M_25_REG	Message	0x00E4	R/W
SHA_M_26_REG	Message	0x00E8	R/W
SHA_M_27_REG	Message	0x00EC	R/W
SHA_M_28_REG	Message	0x00F0	R/W
SHA_M_29_REG	Message	0x00F4	R/W
SHA_M_30_REG	Message	0x00F8	R/W
SHA_M_31_REG	Message	0x00FC	R/W

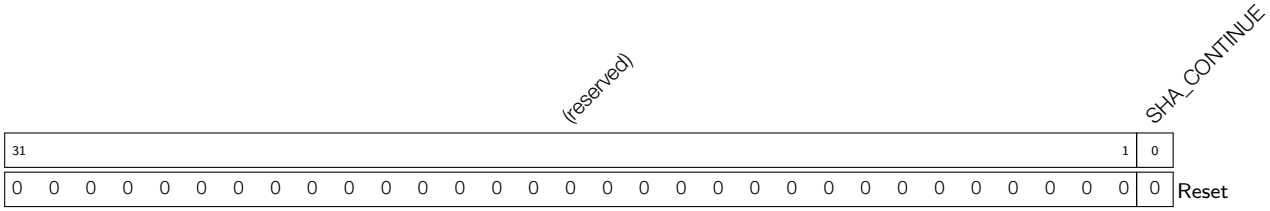
14.7 Registers

Register 14.1: SHA_START_REG (0x0010)



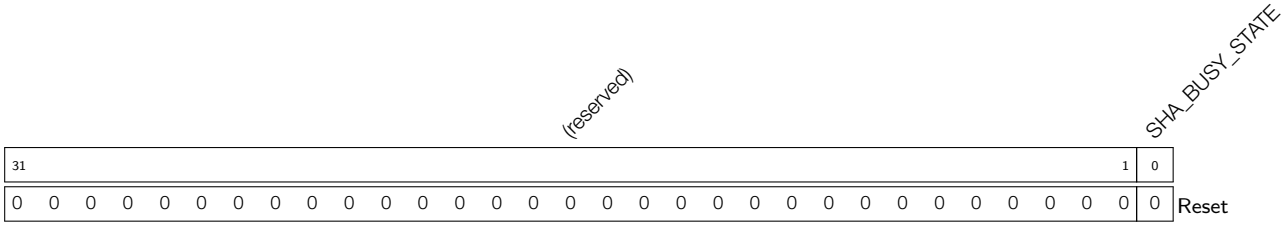
SHA_START Write 1 to start Typical SHA calculation. (WO)

Register 14.2: SHA_CONTINUE_REG (0x0014)



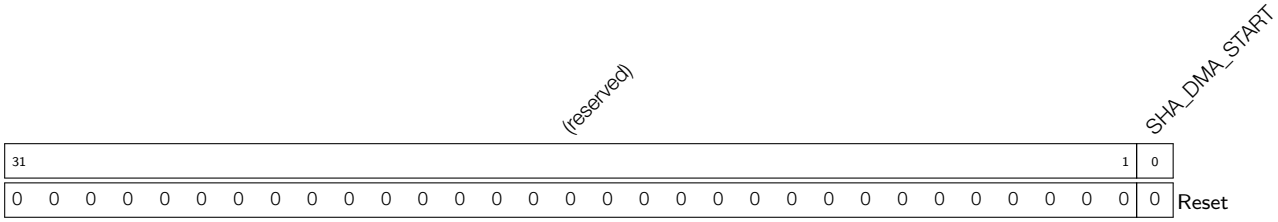
SHA_CONTINUE Write 1 to continue Typical SHA calculation. (WO)

Register 14.3: SHA_BUSY_REG (0x0018)



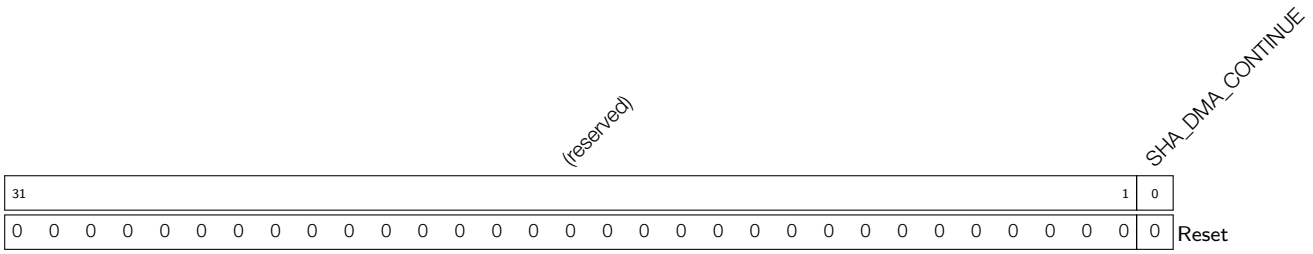
SHA_BUSY_STATE Indicates the states of SHA accelerator. (RO) 1'h0: idle 1'h1: busy

Register 14.4: SHA_DMA_START_REG (0x001C)



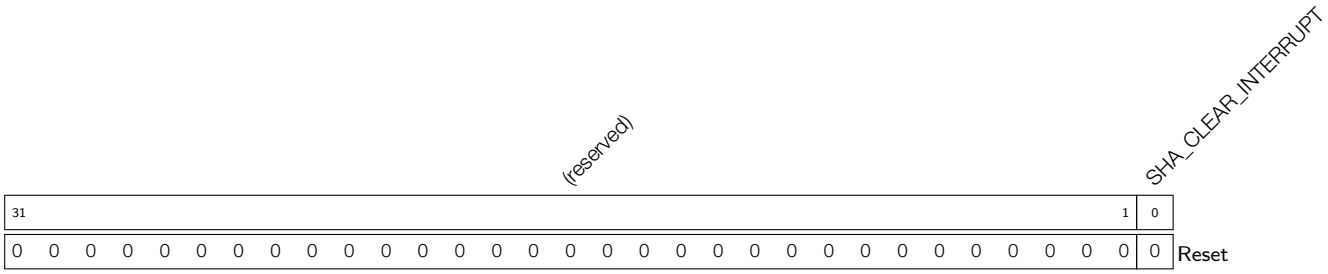
SHA_DMA_START Write 1 to start DMA-SHA calculation. (WO)

Register 14.5: SHA_DMA_CONTINUE_REG (0x0020)



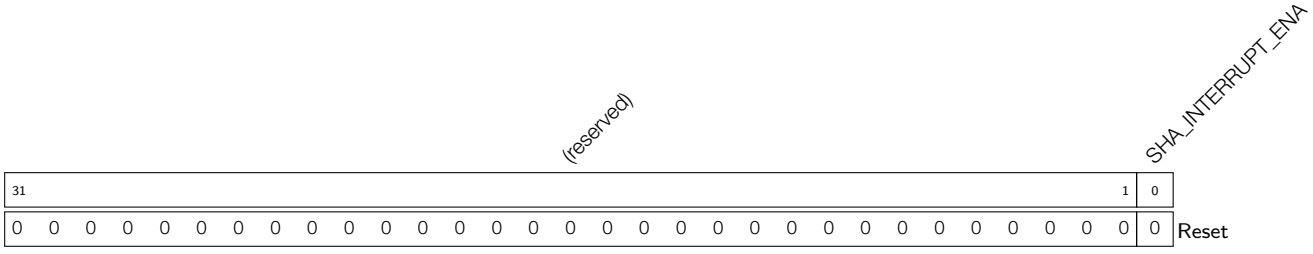
SHA_DMA_CONTINUE Write 1 to continue DMA-SHA calculation. (WO)

Register 14.6: SHA_INT_CLEAR_REG (0x0024)



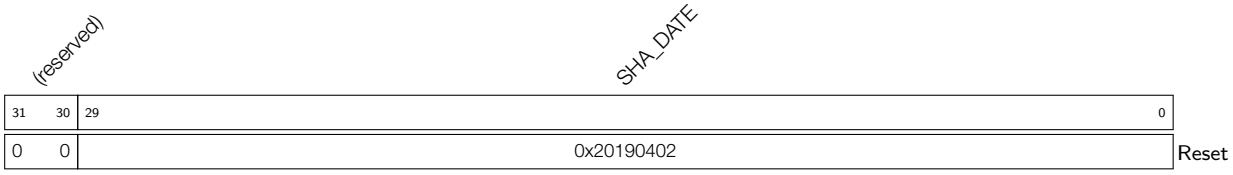
SHA_CLEAR_INTERRUPT Clears DMA-SHA interrupt. (WO)

Register 14.7: SHA_INT_ENA_REG (0x0028)



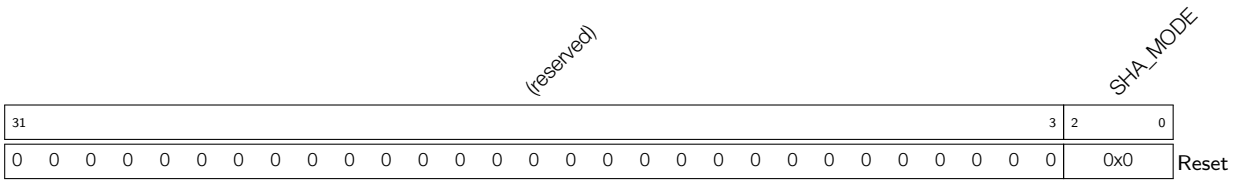
SHA_INTERRUPT_ENA Enables DMA-SHA interrupt. (R/W)

Register 14.8: SHA_DATE_REG (0x002C)



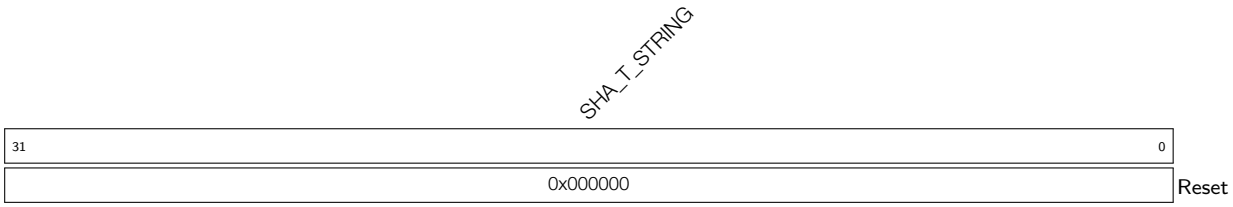
SHA_DATE Version control register. (R/W)

Register 14.9: SHA_MODE_REG (0x0000)



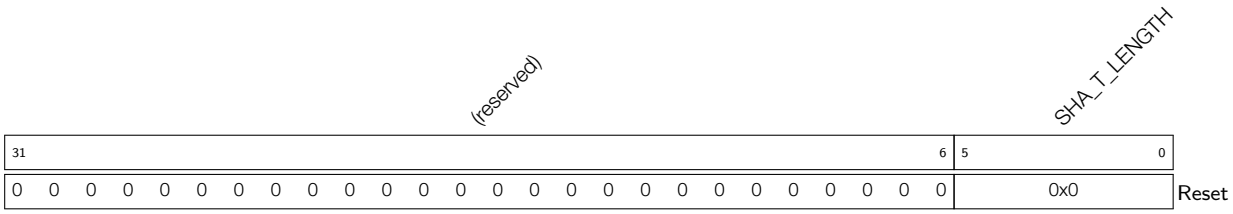
SHA_MODE Defines the SHA algorithm. For details, please see Table 14-2. (R/W)

Register 14.10: SHA_T_STRING_REG (0x0004)



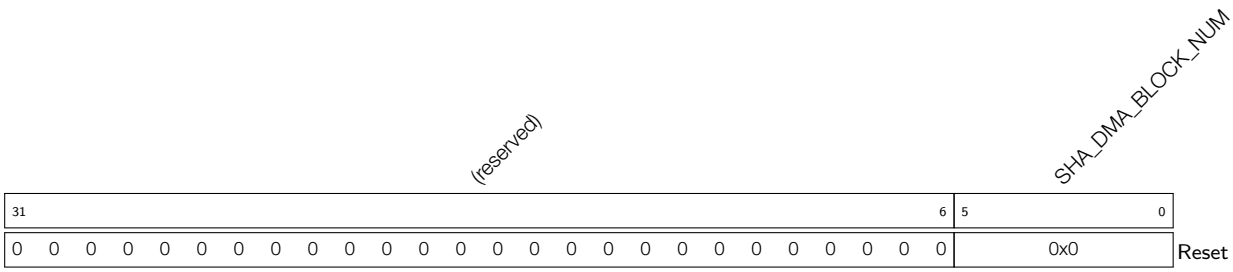
SHA_T_STRING Defines t_string for calculating the initial Hash value for SHA-512/t. (R/W)

Register 14.11: SHA_T_LENGTH_REG (0x0008)



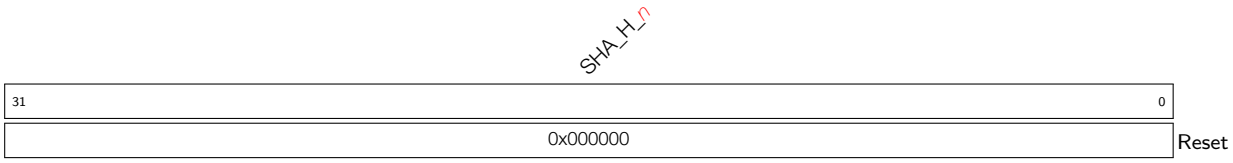
SHA_T_LENGTH Defines t_length for calculating the initial Hash value for SHA-512/t. (R/W)

Register 14.12: SHA_DMA_BLOCK_NUM_REG (0x000C)



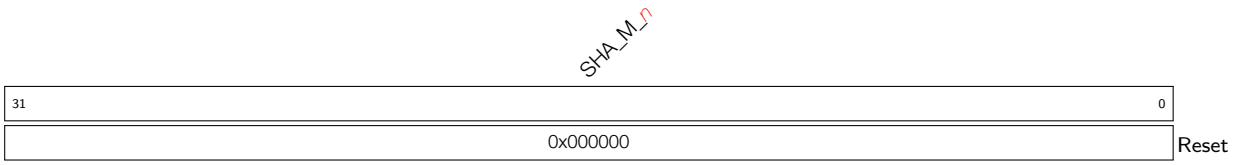
SHA_DMA_BLOCK_NUM Defines the DMA-SHA block number. (R/W)

Register 14.13: SHA_H_*n*_REG (*n*: 0-15) (0x0040+4n*)**



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

Register 14.14: SHA_M_*n*_REG (*n*: 0-31) (0x0080+4n*)**



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

15. RSA Accelerator

15.1 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.

15.2 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

15.3 Functional Description

The RSA Accelerator is activated by setting the DPORT_PERIP_CLK_EN1_REG bit in the DPORT_CRYPTO_RSA_CLK_EN peripheral clock and clearing the DPORT_RSA_PD bit in the DPORT_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.

Notice:

ESP32-S2's Digital Signature module also calls the RSA accelerator. Therefore, users cannot access the RSA accelerator when Digital Signature is working.

15.3.1 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 $x \in \{1, 2, 3, \ldots, 128\}$. 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 $\overline{r} = R^2 \bmod M$, where $R = b^n$.

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 15.3.4.
3. Write X_i, Y_i, M_i and $\overline{r}_i$ for $i \in \{0, 1, \ldots, n\}$ to memory blocks RSA_X_MEM, RSA_Y_MEM, RSA_M_MEM and RSA_Z_MEM. The capacity of each memory block is 128 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 $i \in \{0, 1, \ldots, n\}$ 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.

15.3.2 Large Number Modular Multiplication

Large-number modular multiplication performs $Z = X \times Y \bmod M$. This computation is based on Montgomery multiplication. The same values $\overline{r}$ and M' are derived by software.

The RSA Accelerator supports large-number modular multiplication with operands of 128 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 $i \in \{0, 1, \ldots, n\}$ to registers RSA_X_MEM, RSA_Y_MEM, RSA_M_MEM and RSA_Z_MEM. The capacity of each memory block is 128 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 $i \in \{0, 1, \ldots, n\}$ 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.

15.3.3 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 $x \in \{0, 1, \ldots, 64\}$. 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 $\in \{0, 1, \ldots, n\}$ to registers RSA_X_MEM and RSA_Z_MEM. The capacity of each memory block is 128 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. n is $\frac{N}{32}$.

   Write X_i for $i \in \{0, 1, \ldots, n\}$ to the address of the i words of the RSA_X_MEM register. Note that Y_i for $i \in \{0, 1, \ldots, n\}$ 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 $i \in \{0, 1, \ldots, n\}$ 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 $\overline{r}_i$ in memory RSA_Z_MEM has already been overwritten, and only this overwritten memory block needs to be re-initialized before starting another computation.

15.3.4 Acceleration Options

ESP32-S2 RSA provides two acceleration options for the large-number modular exponentiation, which are SEARCH Option and the CONSTANT_TIME Option. These two options are both disabled by default, but can be enabled at the same time.

When neither of these two options are enabled, the time required to calculate $Z = X^Y \bmod M$ is solely determined by the lengths of operands. However, when either one of these two options is enabled, the time required is also correlated with the 0/1 distribution of Y.

To better illustrate the acceleration options, 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 the acceleration options are enabled, the RSA accelerator:

- SEARCH Option
  - 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 accleration option is disabled. 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
  - 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 when different acceleration options are enabled. Here we perform $Z = X^Y \bmod M$ with N = 3072 and Y = 65537. Table 15-1 below demonstrates the time costs when different acceleration options are enabled. It's obvious that the time cost can be dramatically reduced when acceleration option(s) is enabled. Here, we should also mention that, α is set to 16 when the SEARCH option is enabled.

Table 15-1. Acceleration Performace

SEARCH Option	CONSTANT_TIME Option	Time Cost	Acceleration Performance by Percentage
Disabled	Disabled	376.405 ms	0%
Enabled	Disabled	2.260 ms	99.41%
Disabled	Enabled	1.203 ms	99.68%
Enabled	Enabled	1.165 ms	99.69%

15.4 Base Address

Users can access RSA with two base addresses, which can be seen in Table 15-2. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 15-2. RSA Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F43C000
PeriBUS2	0x6003C000

15.5 Memory Summary

Both the starting address and ending address in the following table are relative to RSA base addresses provided in Section 15.4.

Name	Description	Size (byte)	Starting Address	Ending Address	Access
RSA_M_MEM	Memory M	512	0x0000	0x01FF	WO
RSA_Z_MEM	Memory Z	512	0x0200	0x03FF	R/W
RSA_Y_MEM	Memory Y	512	0x0400	0x05FF	WO
RSA_X_MEM	Memory X	512	0x0600	0x07FF	WO

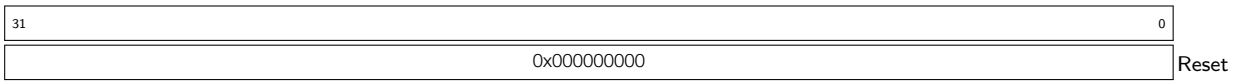
15.6 Register Summary

The addresses in the following table are relative to RSA base addresses provided in Section 15.4.

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
Interupt 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

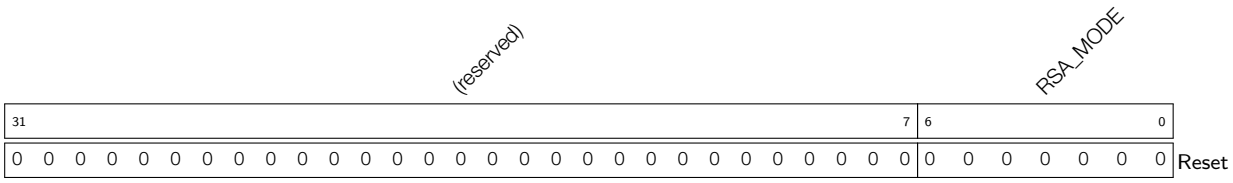
15.7 Registers

Register 15.1: RSA_M_PRIME_REG (0x800)



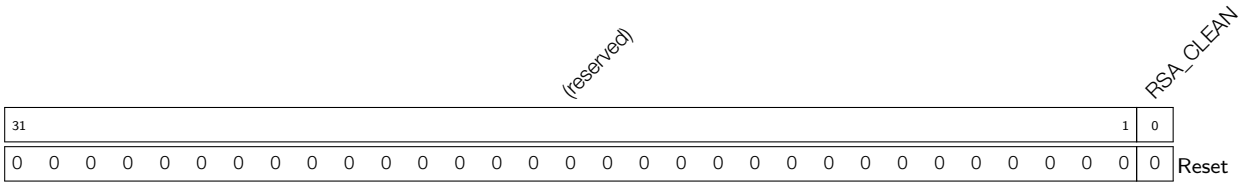
RSA_M_PRIME_REG Stores M'.(R/W)

Register 15.2: RSA_MODE_REG (0x804)



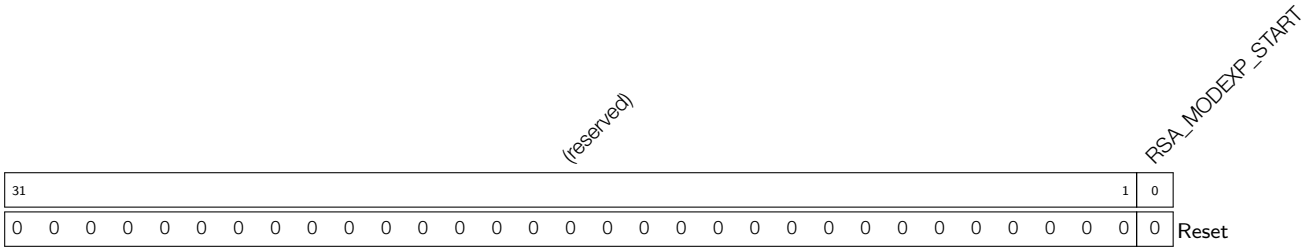
RSA_MODE Stores the mode of modular exponentiation. (R/W)

Register 15.3: RSA_CLEAN_REG (0x0808)



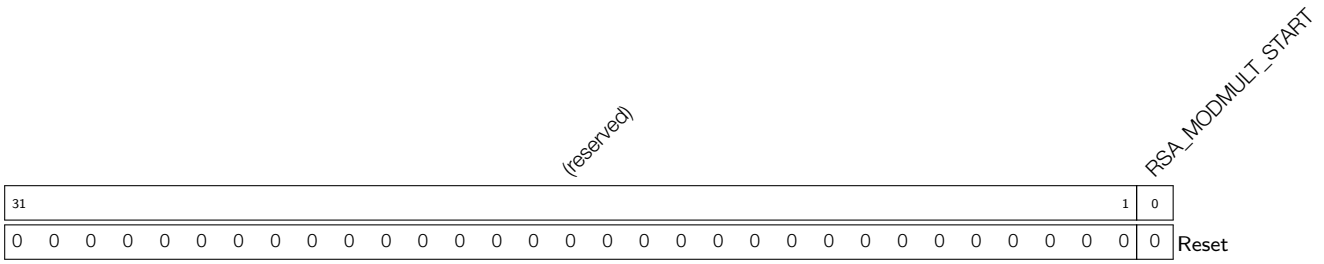
RSA_CLEAN The content of this bit is 1 when memories complete initialization. (RO)

Register 15.4: RSA_MODEXP_START_REG (0x080C)



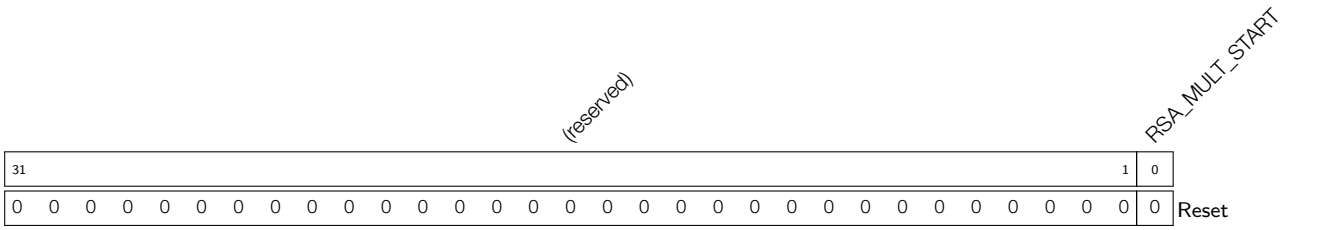
RSA_MODEXP_START Set this bit to 1 to start the modular exponentiation. (WO)

Register 15.5: RSA_MODMULT_START_REG (0x0810)



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

Register 15.6: RSA_MULT_START_REG (0x0814)

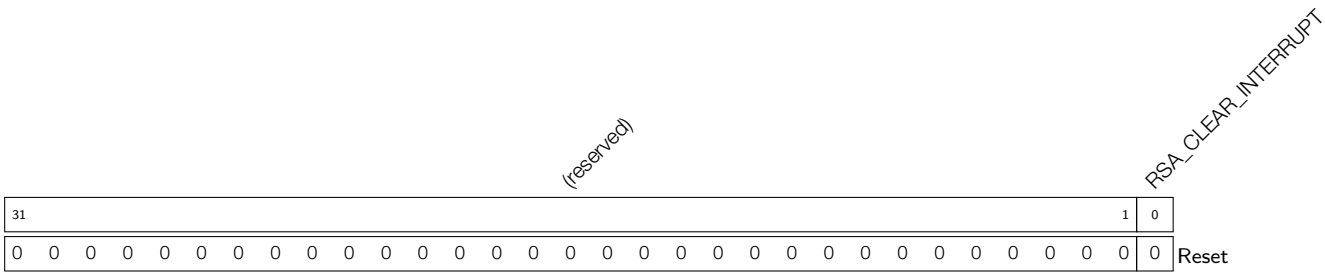


RSA_MULT_START Set this bit to 1 to start the multiplication. (WO)

Register 15.7: RSA_IDLE_REG (0x0818)

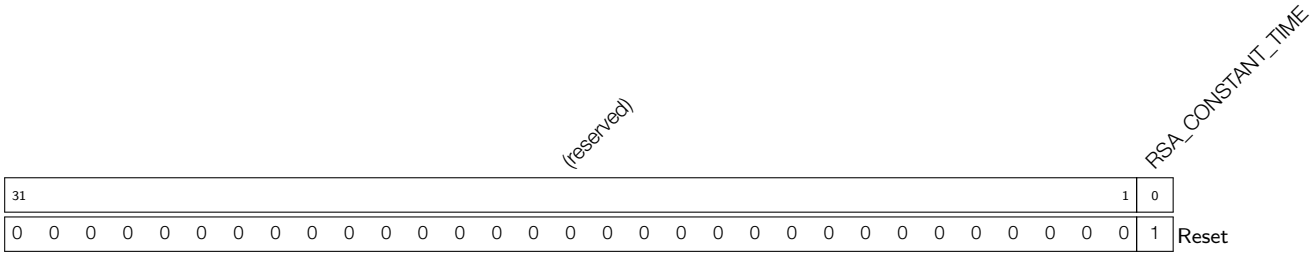
RSA_IDLE The content of this bit is 1 when the RSA accelerator is idle. (RO)

Register 15.8: RSA_CLEAR_INTERRUPT_REG (0x081C)



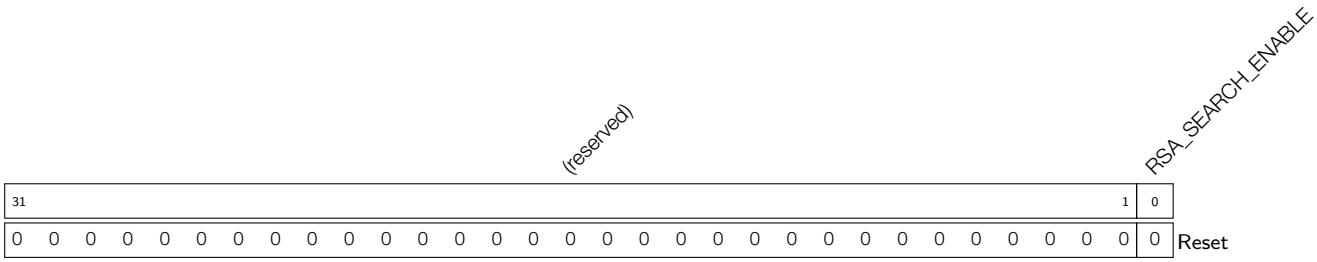
RSA_CLEAR_INTERRUPT Set this bit to 1 to clear the RSA interrupts. (WO)

Register 15.9: RSA_CONSTANT_TIME_REG (0x0820)



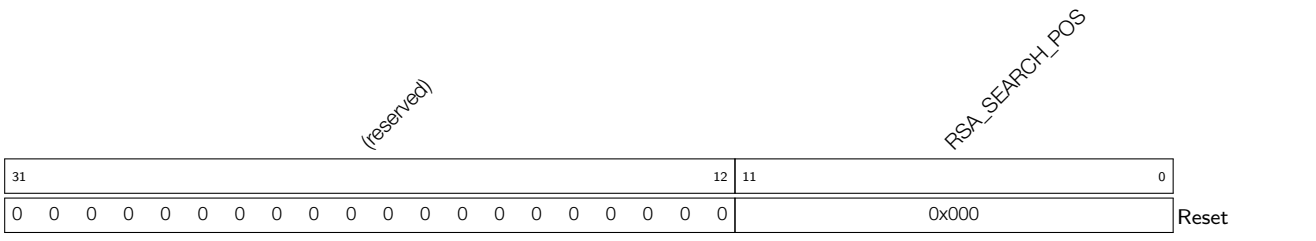
RSA_CONSTANT_TIME_REG Set this bit to 0 to enable the acceleration option of constant_time for modular exponentiation. Set to 1 to disable the acceleration (by default). (R/W)

Register 15.10: RSA_SEARCH_ENABLE_REG (0x0824)



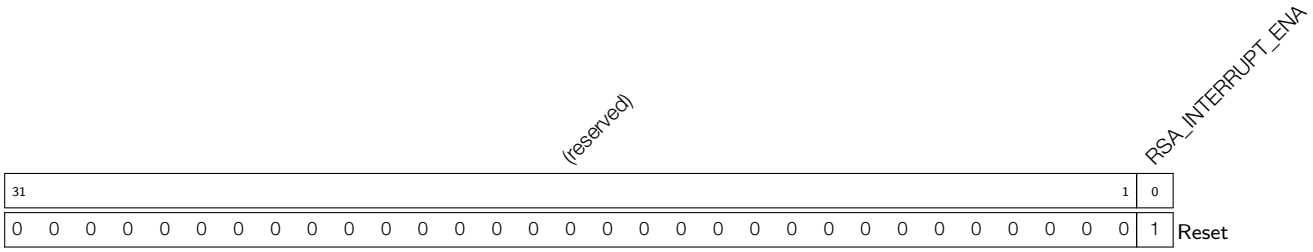
RSA_SEARCH_ENABLE Set this bit to 1 to enable the acceleration option of search for modular exponentiation. Set to 0 to disable the acceleration (by default). (R/W)

Register 15.11: RSA_SEARCH_POS_REG (0x0828)



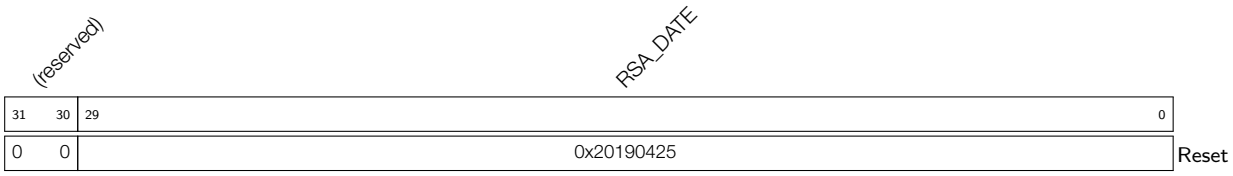
RSA_SEARCH_POS Is used to configure the starting address when the acceleration option of search is used. (R/W)

Register 15.12: 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 15.13: RSA_DATE_REG (0x0830)



RSA_DATE Version control register. (R/W)

16. Random Number Generator

16.1 Introduction

The ESP32-S2 contains a true random number generator, which generates random numbers that can be used for cryptographical operations, among other things.

16.2 Features

The random number generator generates true random numbers, which means no number generated within the specified range is more or less likely to appear than any other number.

16.3 Functional Description

When used correctly, 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.

Either the high-speed ADC, SAR ADC, or both can be used as the noise source for the random number generator.

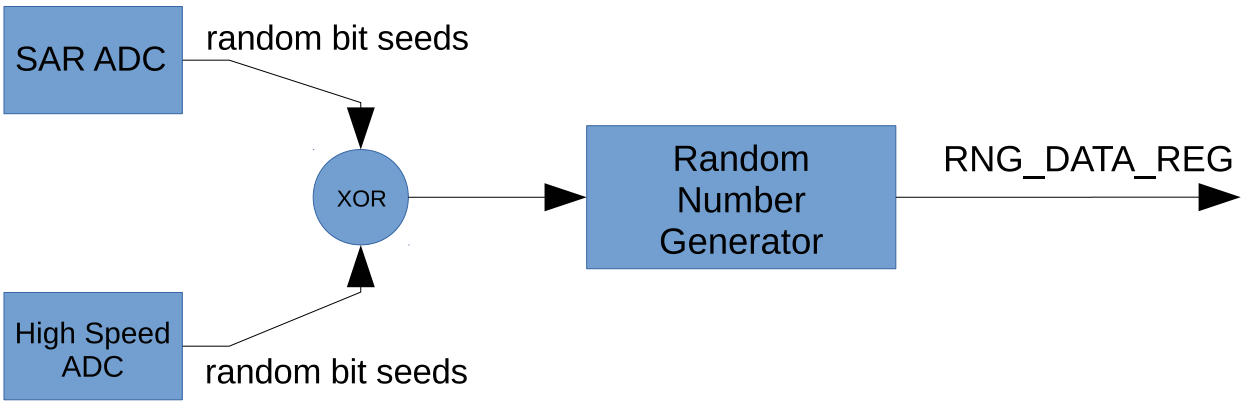


Figure 16-1. Noise Source

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.

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 8 MHz, which is generated from an internal RC oscillator (see Reset and Clock for details). Thus, it is advisable to read the RNG_DATA_REG register at a maximum rate of 500 kHz 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.

Note:
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 enable the SAR ADC as the noise source for the random number generator when the Wi-Fi module is enabled.

To facilitate the generation of random numbers, a system API for generating random numbers will be provided. It's advisable to call this API directly.

16.4 Base Address

Users can access the random number generator with two base addresses, which can be seen in Table 16-1. For more information about accessing peripherals from different buses, please see Chapter 1 *System and Memory*.

Table 16-1. Random Number Generator Base Address

Bus	Base Address
PeriBUS1	0x3FF75000
PeriBUS2	0x60035000

16.5 Register Summary

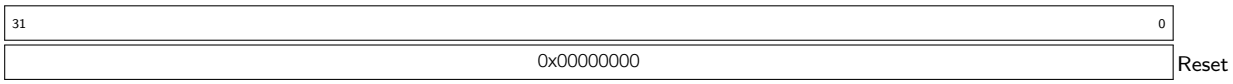
The addresses in the following table are relative to the random number generator base addresses provided in Section 16.4.

Name	Description	Address	Access
RNG_DATA_REG	Random number data	0x0110	RO

16.6 Register

The address in this section is relative to the random number generator base addresses provided in Section 16.4.

Register 16.1: RNG_DATA_REG (0x0110)



RNG_DATA Random number source. (RO)

17. External Memory Encryption and Decryption

17.1 Overview

The ESP32-S2 SoC implements an External Memory Encryption and Decryption module that secures users' application code and data stored in the external memory (flash and external RAM). The encryption and decryption algorithm complies with the XTS-AES standard specified in IEEE Std 1619-2007. Users can store proprietary firmware and sensitive data (for example credentials for gaining access to a private network) to the external flash, and general data to the external RAM.

17.2 Features

- General XTS-AES algorithm, compliant with IEEE Std 1619-2007
- Software-based manual encryption
- High-speed hardware auto encryption
- High-speed hardware auto decryption
- Encryption and decryption functions jointly determined by register configuration, eFuse parameters, and boot mode

17.3 Functional Description

The External Memory Encryption and Decryption module consists of three blocks, namely the Manual Encryption block, Auto Encryption block, and Auto Decryption block. The module architecture is shown in Figure 17-1.

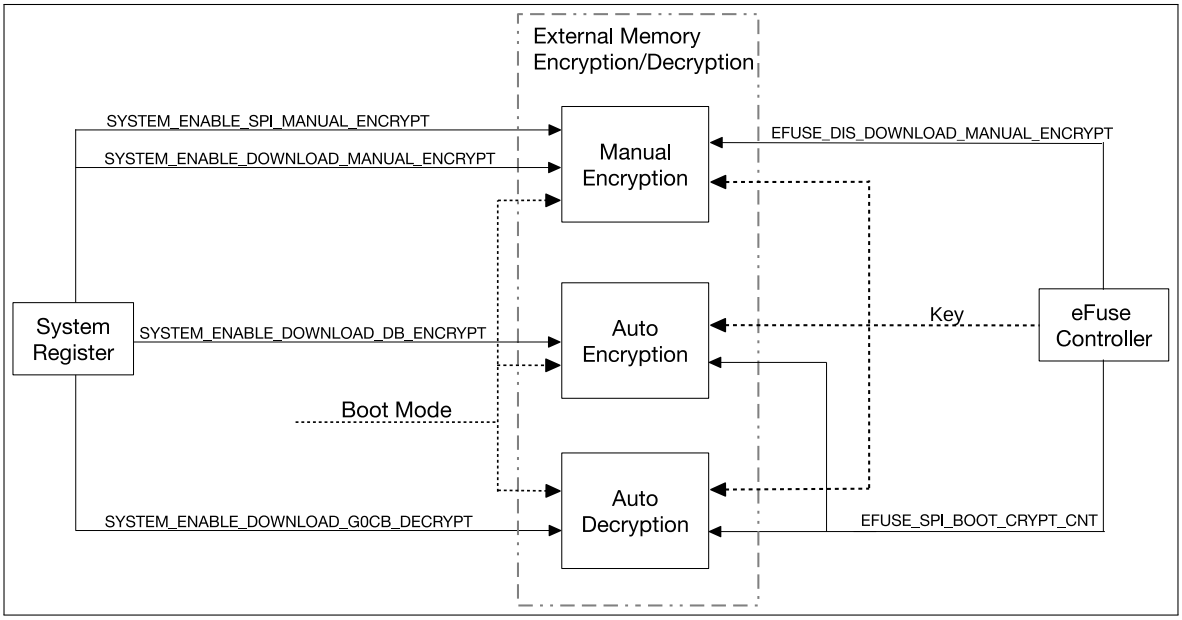


Figure 17-1. Architecture of the External Memory Encryption and Decryption Module

The Manual Encryption block can encrypt instructions and data which will then be written to the external flash as ciphertext through SPI1.

When the CPU writes the external RAM through cache, the Auto Encryption block automatically encrypts the data first, and the data is written to the external RAM as ciphertext.

When the CPU reads the external flash or RAM through cache, the Auto Decryption block automatically decrypts the ciphertext to retrieve instructions and data.

In the peripheral System Register, four bits in the SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG register are relevant to external memory encryption and decryption:

- SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT
- SYSTEM_ENABLE_DOWNLOAD_G0CB_DECRYPT
- SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT
- SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT

The External Memory Encryption and Decryption module fetches two parameters from the peripheral eFuse Controller. These parameters are: EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT and EFUSE_SPI_BOOT_CRYPT_CNT.

17.3.1 XTS Algorithm

The manual encryption, auto encryption, and auto decryption operations use the same algorithm, i.e., XTS algorithm. In real-life implementation, the XTS algorithm is characterized by "data unit" of 1024 bits. The "data unit" is defined in the *XTS-AES Tweakable Block Cipher* standard, section *XTS-AES encryption procedure*. More information on the XTS-AES algorithm can be found in IEEE Std 1619-2007.

17.3.2 Key

The Manual Encryption block, Auto Encryption block, and Auto Decryption block share the same key to perform XTS algorithm. The key is provided by the eFuse hardware and protected from user access.

The key can be either 256 bits or 512 bits long. The key is determined by the content in one or two eFuse blocks from BLOCK4 ~ BLOCK9. For easy description, define:

- Block$_A$, which refers to the block that has the key purpose set to EFUSE_KEY_PURPOSE_XTS_AES_256_KEY_1. Block$_A$ contains 256-bit Key_A.
- Block$_B$, which refers to the block that has the key purpose set to EFUSE_KEY_PURPOSE_XTS_AES_256_KEY_2. Block$_B$ contains 256-bit Key_B.
- Block$_C$, which refers to the block that has the key purpose set to EFUSE_KEY_PURPOSE_XTS_AES_128_KEY. Block$_C$ contains 256-bit Key_C.

Table 17-1 shows how the Key is generated, depending on whether Block$_A$, Block$_B$, and Block$_C$ exists or not.

Table 17-1. Key

$Block_A$	$Block_B$	$Block_C$	Key	Key Length (bit)
Yes	Yes	Don't care	$Key_A \| \| Key_B$	512
Yes	No	Don't care	$Key_A \| \| 0^{256}$	512
No	Yes	Don't care	$0^{256} \| \| Key_B$	512
No	No	Yes	Key_C	256
No	No	No	0^{256}	256

"Yes" indicates that the block exists; "No" indicates that the block does not exist; "0^{256}" indicates a bit string that consists of 256-bit zeros; "||" is a bonding operator for joining one key string to another.

For more information on setting of key purposes, please refer to Chapter 11 *eFuse Controller*.

17.3.3 Target Memory Space

The target memory space refers to a continuous address space in the external memory where the encrypted result is stored. The target memory space can be uniquely determined by three relevant parameters: $type$, $size$, and $base\_addr$. They are defined as follows:

- $type$: the type of the external memory, either flash or external RAM. Value 0 indicates flash, 1 indicates external RAM.
- $size$: the size of the target memory space, in unit of bytes. One single encryption operation supports either 16, 32, or 64 bytes of data.
- $base\_addr$: the base address of the target memory space. It is a physical address aligned to $size$, i.e., $base\_addr \% size == 0$.

Assume encrypted 16 bytes written to address 0x130 ~ 0x13F in the external flash, then, the target memory space is 0x130 ~ 0x13F, $type$ is 0 (flash), $size$ is 16 (bytes), and $base\_addr$ 0x130.

The encryption of any length (must be multiples of 16 bytes) of data can be completed separately in multiple operations. Each operation can have individual target memory space and the relevant parameters.

For auto encryption and auto decryption, these parameters are automatically defined by hardware. For manual encryption, these parameters should be configured manually by users.

17.3.4 Data Padding

For auto encryption and auto decryption, data padding is automatically completed by hardware. For manual encryption, data padding should be completed manually by users. The Manual Encryption block is equipped with 16 registers, i.e., XTS_AES_PLAIN_*n*_REG (*n*: 0-15), that are dedicated to data padding and can store up to 512 bits of plaintext at a time.

Actually, the Manual Encryption block does not care where the plaintext comes from, but only where the ciphertext is to be stored. Because of the strict correspondence between plaintext and ciphertext, in order to better describe how the plaintext is stored in the register heap, it is assumed that the plaintext is stored in the target memory space in the first place and replaced by ciphertext after encryption. Therefore, the following description no longer has the concept of "plaintext", but uses "target memory space" instead. However, users should note that the plaintext can come from anywhere, and that they should understand how the plaintext is stored in the register heap.

How mapping works between target memory space and registers:

Assume a word is stored in $address$, define $offset = address\%64$, $n = \frac{offset}{4}$, then the word will be stored in register XTS_AES_PLAIN_n_REG.

For example, if the $size$ of the target memory space is 64, then all the 16 registers will be used for data storage. The mapping between $offset$ and registers is shown in Table 17-2.

Table 17-2. Mapping Between Offsets and Registers

$offset$	Register	$offset$	Register
0x00	XTS_AES_PLAIN_0_REG	0x20	XTS_AES_PLAIN_8_REG
0x04	XTS_AES_PLAIN_1_REG	0x24	XTS_AES_PLAIN_9_REG
0x08	XTS_AES_PLAIN_2_REG	0x28	XTS_AES_PLAIN_10_REG
0x0C	XTS_AES_PLAIN_3_REG	0x2C	XTS_AES_PLAIN_11_REG
0x10	XTS_AES_PLAIN_4_REG	0x30	XTS_AES_PLAIN_12_REG
0x14	XTS_AES_PLAIN_5_REG	0x34	XTS_AES_PLAIN_13_REG
0x18	XTS_AES_PLAIN_6_REG	0x38	XTS_AES_PLAIN_14_REG
0x1C	XTS_AES_PLAIN_7_REG	0x3C	XTS_AES_PLAIN_15_REG

17.3.5 Manual Encryption Block

The Manual Encryption block is a peripheral module. It is equipped with registers that can be accessed by the CPU directly. Registers embedded in this block, System registers, eFuse parameters, and boot mode jointly configure and control this block. Please note that currently the Manual Encryption block can only encrypt flash.

The manual encryption requires software participation. The steps are as follows:

1. Configure XTS_AES:
   - Set XTS_AES_DESTINATION_REG register to $type$ = 0.
   - Set XTS_AES_PHYSICAL_ADDRESS_REG register to $base\_addr$.
   - Set XTS_AES_LINESIZE_REG register to $\frac{size}{32}$.

   For definitions of $type$, $base\_addr$, $size$, please refer to Section 17.3.3.
2. Fill registers XTS_AES_PLAIN_n_REG (n: 0-15) in with plaintext (refer to Section 17.3.4). Registers that are not used can be written into any value.
3. Poll XTS_AES_STATE_REG until it reads 0 that indicates the Manual Encryption block is idle.
4. Activate encryption by writing 1 to XTS_AES_TRIGGER_REG register.
5. Wait for the encryption to complete. Poll register XTS_AES_STATE_REG until it reads 2.
   Steps 1 ~ 5 complete the encryption operation, where Key is used.
6. Grant SPI1 access to the encrypted result by writing 1 to XTS_AES_RELEASE_REG register.
   XTS_AES_STATE_REG will read 3 afterwards.
7. Call SPI1 and write the encrypted result to the external flash.
8. Destroy the encrypted result by writing 1 to XTS_AES_DESTROY_REG. XTS_AES_STATE_REG register will read 0 afterwards.

Repeat the steps above to complete multiple encryption operations.

The Manual Encryption block is operational only with granted permission. The operating conditions are:

- In SPI Boot mode
  If bit SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT in register SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG is 1, the Manual Encryption block is granted permission. Otherwise, it is not operational.
- In Download Boot mode
  If bit SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT in register SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG is 1 and the eFuse parameter EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT is 0, the Manual Encryption block is granted permission. Otherwise, it is not operational.

Note:

- Even though the CPU can skip cache and get the encrypted result directly by reading the external memory, software can by no means access Key.
- The Manual Encryption block needs to call the AES accelerator to perform encryption. Therefore, users cannot access AES accelerator during the process.

17.3.6 Auto Encryption Block

The Auto Encryption block is not a conventional peripheral, and is not equipped with registers. Therefore, the CPU cannot directly access this block. The System Register, eFuse parameters, and boot mode jointly control this block.

The Auto Encryption block is operational only with granted permission. The operating conditions are:

- In SPI Boot mode
  If the 3-bit parameter SPI_BOOT_CRYPT_CNT has 1 or 3 bits set to 1, then the Auto Encryption block is granted permission. Otherwise, it is not operational.
- In Download Boot mode
  If bit SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT in register SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG is 1, the Auto Encryption block is granted permission. Otherwise, it is not operational.

Note:

- When the Auto Encryption block is operational, the CPU will read data from the external RAM via the cache. The Auto Encryption block automatically encrypts the data and write it to the external RAM. The entire encryption process does not need software participation and is transparent to the cache. Software cannot access the encryption Key.
- When the Auto Encryption block is not operational, it will ignore the CPU's request to access cache and do not process the data. Therefore, data will be written to the external RAM as plaintext.

17.3.7 Auto Decryption Block

The Auto Decryption block is not a conventional peripheral, and is not equipped with registers. Therefore, the CPU cannot directly access this block. The System Register, eFuse parameters, and boot mode jointly control and configure this block.

The Auto Decryption block is operational only with granted permission. The operating conditions are:

- In SPI Boot mode
  If the 3-bit parameter SPI_BOOT_CRYPT_CNT has 1 or 3 bits set to 1, then the Auto Decryption block is granted permission. Otherwise, it is not operational.
- In Download Boot mode
  If bit SYSTEM_ENABLE_DOWNLOAD_G0CB_DECRYPT in register SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG is 1, the Auto Decryption block is granted permission. Otherwise, it is not operational.

Note:

- When the Auto Decryption block is operational, the CPU will read instructions and data from the external memory via cache. The Auto Decryption block automatically decrypts and retrieves the instructions and data. The entire decryption process does not need software participation and is transparent to the cache. Software cannot access the decryption Key.
- When the Auto Decryption block is not operational, it does not have any effect on the contents stored in the external memory, be they encrypted or unencrypted. What the CPU reads via cache is the original information stored in the external memory.

17.4 Base Address

Users can access the Manual Encryption block with two base addresses, which can be seen in the following table. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 17-3. Manual Encryption Block Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F43A000
PeriBUS2	0x6003A000

17.5 Register Summary

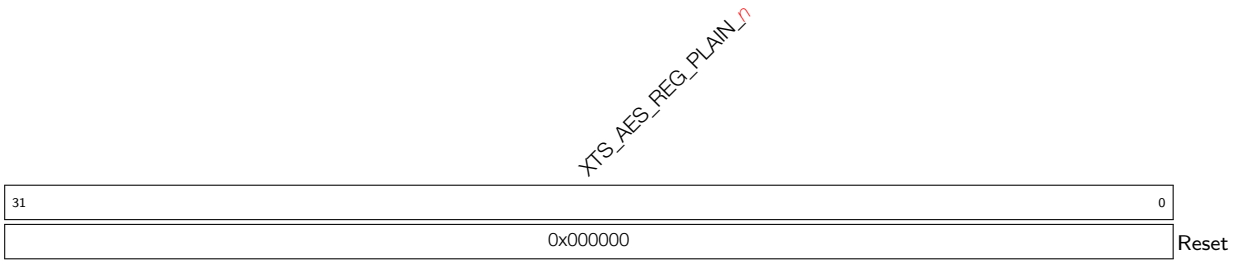
The addresses in the following table are relative to the Manual Encryption block's base addresses provided in Section 17.4.

Name	Description	Address	Access
Plaintext Register Heap			
XTS_AES_PLAIN_0_REG	Plaintext register 0	0x0100	R/W
XTS_AES_PLAIN_1_REG	Plaintext register 1	0x0104	R/W

Name	Description	Address	Access
XTS_AES_PLAIN_2_REG	Plaintext register 2	0x0108	R/W
XTS_AES_PLAIN_3_REG	Plaintext register 3	0x010C	R/W
XTS_AES_PLAIN_4_REG	Plaintext register 4	0x0110	R/W
XTS_AES_PLAIN_5_REG	Plaintext register 5	0x0114	R/W
XTS_AES_PLAIN_6_REG	Plaintext register 6	0x0118	R/W
XTS_AES_PLAIN_7_REG	Plaintext register 7	0x011C	R/W
XTS_AES_PLAIN_8_REG	Plaintext register 8	0x0120	R/W
XTS_AES_PLAIN_9_REG	Plaintext register 9	0x0124	R/W
XTS_AES_PLAIN_10_REG	Plaintext register 10	0x0128	R/W
XTS_AES_PLAIN_11_REG	Plaintext register 11	0x012C	R/W
XTS_AES_PLAIN_12_REG	Plaintext register 12	0x0130	R/W
XTS_AES_PLAIN_13_REG	Plaintext register 13	0x0134	R/W
XTS_AES_PLAIN_14_REG	Plaintext register 14	0x0138	R/W
XTS_AES_PLAIN_15_REG	Plaintext register 15	0x013C	R/W
Configuration Registers			
XTS_AES_LINESIZE_REG	Configures the size of target memory space	0x0140	R/W
XTS_AES_DESTINATION_REG	Configures the type of the external memory	0x0144	R/W
XTS_AES_PHYSICAL_ADDRESS_REG	Physical address	0x0148	R/W
Control/Status Registers			
XTS_AES_TRIGGER_REG	Activates AES algorithm	0x014C	WO
XTS_AES_RELEASE_REG	Release control	0x0150	WO
XTS_AES_DESTROY_REG	Destroys control	0x0154	WO
XTS_AES_STATE_REG	Status register	0x0158	RO
Version Register			
XTS_AES_DATE_REG	Version control register	0x015C	RO

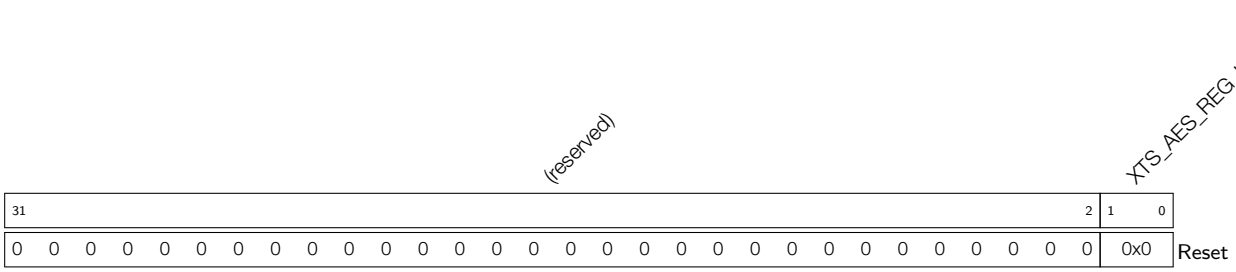
17.6 Registers

Register 17.1: XTS_AES_PLAIN_*n*_REG (*n*: 0-15) (0x0100+4n*)**



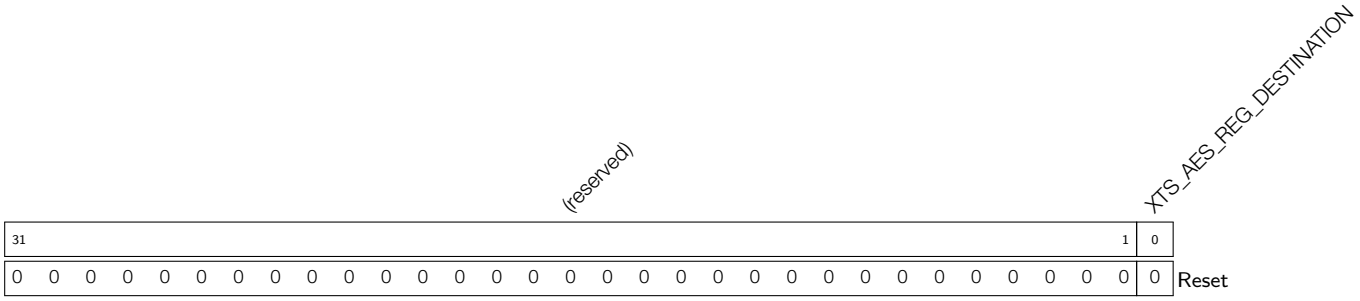
XTS_AES_REG_PLAIN_*n* This register stores *n*th 32-bit piece of plaintext. (R/W)

Register 17.2: XTS_AES_LINESIZE_REG (0x0140)



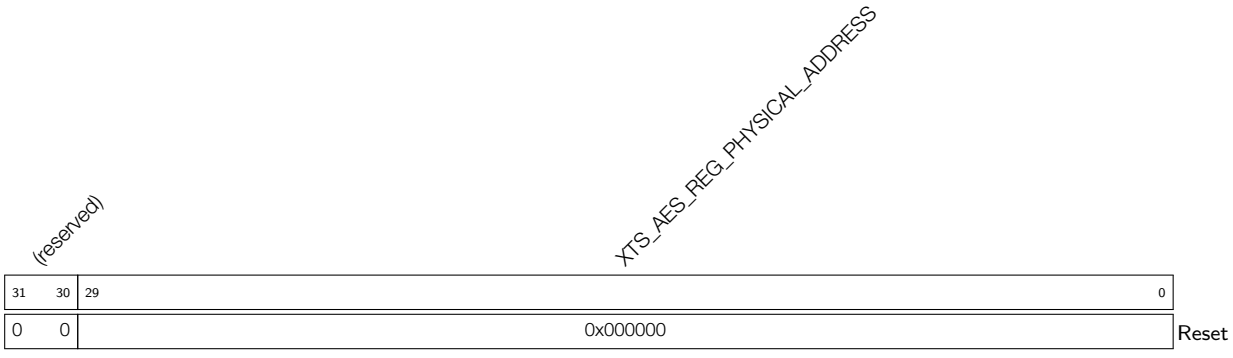
XTS_AES_REG_LINESIZE Configures the data size of a single encryption. 0: 128 bits; 1: 256 bits; 2: 512 bits. (R/W)

Register 17.3: XTS_AES_DESTINATION_REG (0x0144)



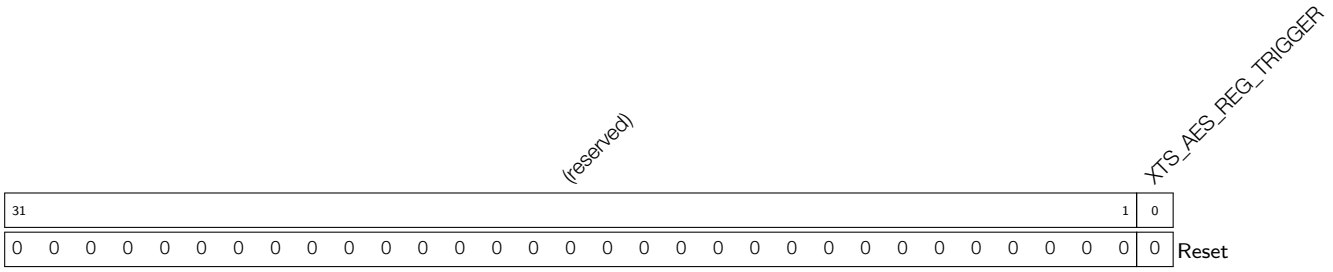
XTS_AES_REG_DESTINATION Configures the type of the external memory. Currently, it must be set to 0, as the Manual Encryption block only supports flash encryption. Errors may occur if users write 1. 0: flash; 1: external RAM. (R/W)

Register 17.4: XTS_AES_PHYSICAL_ADDRESS_REG (0x0148)



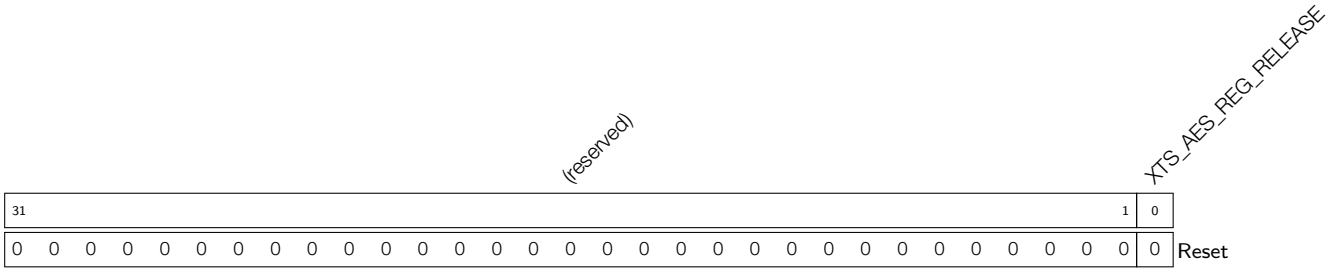
XTS_AES_REG_PHYSICAL_ADDRESS Physical address. (R/W)

Register 17.5: XTS_AES_TRIGGER_REG (0x014C)



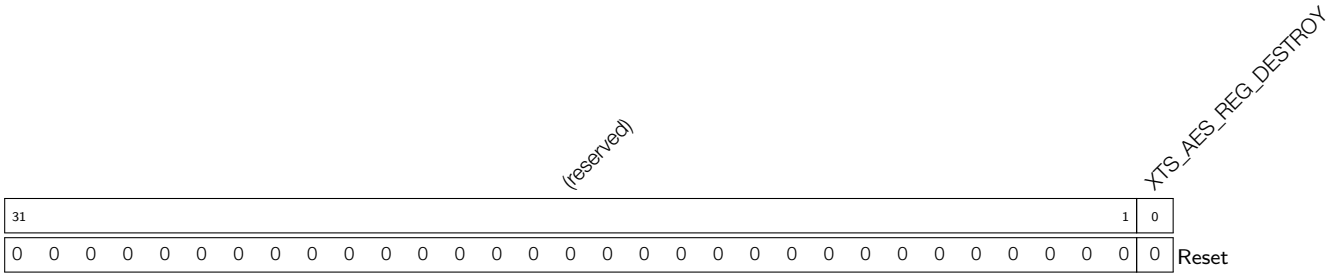
XTS_AES_REG_TRIGGER Set to enable manual encryption. (WO)

Register 17.6: XTS_AES_RELEASE_REG (0x0150)



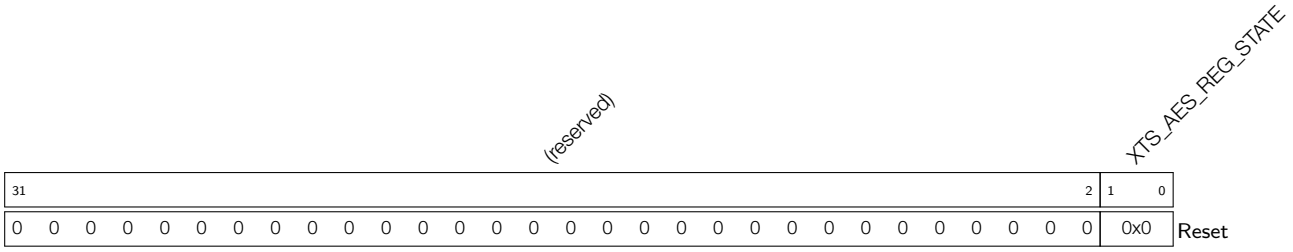
XTS_AES_REG_RELEASE Set to grant SPI1 access to encrypted result. (WO)

Register 17.7: XTS_AES_DESTROY_REG (0x0154)



XTS_AES_REG_DESTROY Set to destroy encrypted result. (WO)

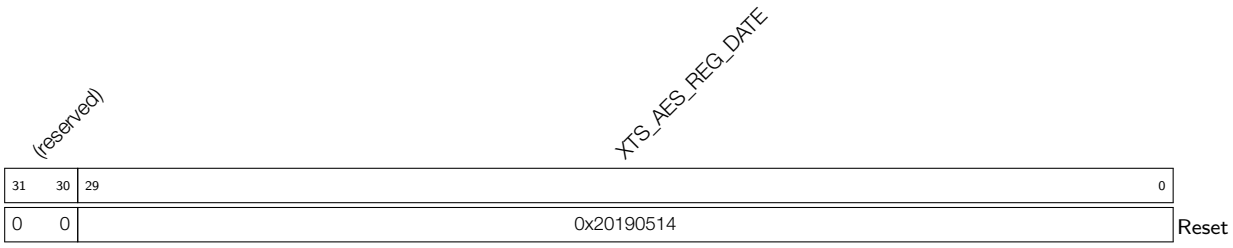
Register 17.8: XTS_AES_STATE_REG (0x0158)



XTS_AES_REG_STATE Indicates the status of the Manual Encryption block. (RO)

- 0x0 (XTS_AES_IDLE): idle;
- 0x1 (XTS_AES_BUSY): busy with encryption;
- 0x2 (XTS_AES_DONE): encryption is completed, but the encrypted result is not accessible to SPI;
- 0x3 (XTS_AES_RELEASE): encrypted result is accessible to SPI.

Register 17.9: XTS_AES_DATE_REG (0x015C)



XTS_AES_REG_DATE Version control register. (RO)

18. Digital Signature

18.1 Overview

Digital signatures provide a way to cryptographically authenticate a message using a private key, to be verified using the corresponding public key. This can be used to validate a device's identity to a server, or to authenticate the integrity of a message has not been tampered with.

ESP32-S2 includes a digital signature (DS) peripheral which produces hardware accelerated RSA digital signatures, without the RSA private key being accessible by software.

18.2 Features

- RSA Digital Signatures with key lengths up to 4096 bits
- Private key data is encrypted and only readable by DS peripheral
- SHA-256 digest is used to protect private key data against tampering by an attacker

18.3 Functional Description

18.3.1 Overview

The DS peripheral calculates the RSA encryption operation $Z = X^Y \bmod M$ where Z is the signature, X is the input message, Y and M are the RSA private key parameters.

Private key parameters are stored in flash or another form of storage, in an encrypted form. They are encrypted using a key which can only be read by the DS peripheral via the HMAC peripheral. The required inputs to generate the key are stored in eFuse and can only be accessed by the HMAC peripheral. This means that only the DS peripheral hardware can decrypt the private key, and the plaintext private key data is never accessed by software.

The input message X is input directly to the DS peripheral by software, each time a signature is needed. After the operation, the signature Z is read back by software.

18.3.2 Private Key Operands

Private key operands Y (private key exponent) and M (key modulus) are generated by the user. They will have a particular RSA key length (up to 4096 bits). A corresponding public key is also generated and stored separately, it can be used independently to verify DS signatures.

Two additional private key operands are needed — $\overline{r}$ and M'. These two operands are derived from Y and M, but they are calculated in advance by software.

Operands Y, M, $\overline{r}$, and M' are encrypted by the user along with an authentication digest and stored as a single ciphertext C. C is input to the DS peripheral in this encrypted format, then the hardware decrypts C and uses the key data to generate the signature. Detailed description of the encryption process to prepare C is provided in Section 18.3.4.

The DS peripheral needs to activate RSA to perform $Z = X^Y \bmod M$. For detailed information on the RSA algorithm, please refer to Section 15.3.1 *Large Number Modular Exponentiation* in Chapter 15 *RSA Accelerator*.

18.3.3 Conventions

The following sections of this chapter will use the following symbols and functions:

- 1^s A bit string that consists of s "1" bits.
- $[x]_s$ A bit string of length s bits. If x is a number ($x < 2^s$), it is represented in little endian byte order in the bit string. x may be a variable value such as $[Y]_{4096}$ or as a hexadecimal constant such as $[0x0C]_8$. If necessary, the value $[x]$ is right-padded with 0s to reach s bits in length. For example: $[0x5]_4 = 0101$, $[0x5]_8 = 00000101$, $[0x5]_{16} = 0000010100000000$, $[0x13]_8 = 00010011$, $[0x13]_{16} = 0001001100000000$.
- $||$ A bit string concatenation operator for joining multiple bit strings into a longer bit string.

18.3.4 Software Storage of Private Key Data

To store a private key for use with the DS peripheral, users need to complete the following preparations:

- Generate the RSA private key (Y, M) and associated operands $\overline{r}$ and M', as described in Section 18.3.2.
- Generate a 256-bit HMAC key ($[HMAC\_KEY]_{256}$) that is stored in eFuse. This HMAC key is read by the HMAC peripheral to derive a key, as DS_KEY = HMAC-SHA256($[HMAC\_KEY]_{256}$, 1^{256}). This key is used to securely encrypt and decrypt the stored RSA private key data.
- Prepare encrypted private key parameters as ciphertext C, 1584 bytes in length.

Figure 18-1 below describes the preparations at the software level (the left part) and the DS peripheral operation at the hardware level (the right part).

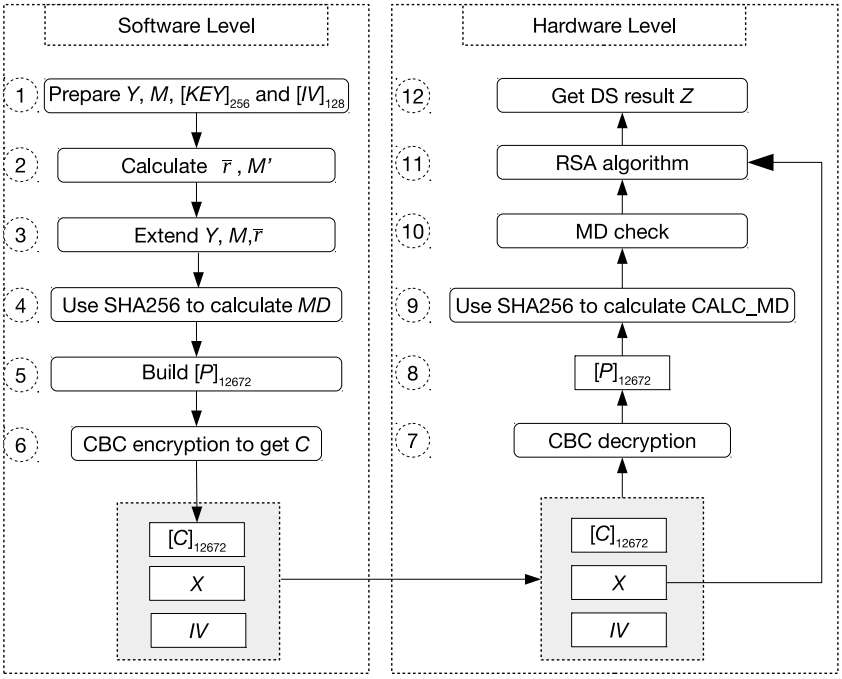


Figure 18-1. Preparations and DS Operation

Users need to follow the steps shown in the left part of Figure 18-1 to calculate C. Detailed instructions are as follows:

- **Step 1**: Prepare Y and M whose lengths should meet the aforementioned requirements. Define $[L]_{32} = \frac{N}{32}$ (i.e., for RSA 4096, $[L]_{32}$ == $[0x80]_{32}$). Prepare $[DS\_KEY]_{256}$ and generate a random $[IV]_{128}$ which should meet the requirements of the AES-CBC block encryption algorithm. For more information on AES, please refer to Chapter 13 *AES Accelerator*.
- **Step 2**: Calculate $\overline{r}$ and M' based on M.
- **Step 3**: Extend Y, M, and $\overline{r}$, in order to get $[Y]_{4096}$, $[M]_{4096}$, and $[\overline{r}]_{4096}$, respectively. Since the largest operand length for Y, M, and $\overline{r}$ is 4096 bits, this step is only required for lengths smaller than 4096 bits.
- **Step 4**: Calculate MD authentication code using the SHA-256 algorithm:

  $[MD]_{256}$ = SHA256 ($[Y]_{4096}||[M]_{4096}||[\overline{r}]_{4096}||[M']_{32}||[L]_{32}||[IV]_{128}$)
- **Step 5**: Build $[P]_{12672}$ = ($[Y]_{4096}||[M]_{4096}||[\overline{r}]_{4096}||[MD]_{256}||[M']_{32}||[L]_{32}||[\beta]_{64}$), where $[\beta]_{64}$ is a PKCS#7 padding value, i.e., a 64-bit string $[0x0808080808080808]_{64}$ that is composed of eight bytes (value = 0x08). The purpose of $[\beta]_{64}$ is to make the bit length of P a multiple of 128.
- **Step 6**: Calculate C = $[C]_{12672}$ = AES-CBC-ENC ($[P]_{12672}$, $[DS\_KEY]_{256}$, $[IV]_{128}$), where C is the ciphertext that includes RSA operands Y, M, $\overline{r}$, M', and L as well as the MD authentication code and $[\beta]_{64}$. DS_KEY is derived from the $HMAC\_KEY$ stored in eFuse, as described above in Section 18.3.4.

18.3.5 DS Operation at the Hardware Level

The hardware operation is triggered each time a Digital Signature needs to be calculated. The inputs are the pre-generated private key ciphertext C, a unique message X, and IV.

The DS operation at the hardware level is a reverse process of preparing C described in Section 18.3.4. The hardware operation can be divided into the following three stages.

1. **Decryption: Step 7 and 8**

   The decryption process is the reverse of Step 6. The DS peripheral will call AES accelerator to decrypt C in CBC block mode and get the resulted plaintext. The decryption process can be represented by P = AES-CBC-DEC (C, DS_KEY, IV), where IV (i.e., $[IV]_{128}$) is defined by users. $[DS\_KEY]_{256}$ is provided by HMAC module, derived from $HMAC\_KEY$ stored in eFuse. $[DS\_KEY]_{256}$ is not readable by software.

   With P, the DS peripheral can work out $[Y]_{4096}$, $[M]_{4096}$, $[\overline{r}]_{4096}$, $[M']_{32}$, $[L]_{32}$, MD authentication code, and the padding value $[\beta]_{64}$. This process is the reverse of Step 5.

2. **Check: Step 9 and 10**

   The DS peripheral will perform two check operations: MD check and padding check. Padding check is not shown in Figure 18-1, as it happens at the same time with MD check.

   - MD check: The DS peripheral calls SHA-256 to get the hash value $[CALC\_MD]_{256}$. This step is the reverse of Step 4. Then, $[CALC\_MD]_{256}$ is compared against $[MD]_{256}$. Only when the two match, MD check passes.
   - Padding check: The DS peripheral checks if $[\beta]_{64}$ complies with the aforementioned PKCS#7 format. Only when $[\beta]_{64}$ complies with the format, padding check passes.

If MD check passes, the DS peripheral will perform subsequent operations, otherwise, it will not. If padding check fails, an error bit is set in the query register, but it does not affect the subsequent operations.

3. **Calculation: Step 11 and 12**

   The DS peripheral treats X, Y, M, and $\overline{r}$ as big numbers. With M', all operands to perform X^Y mod M are in place. The operand length is defined by L. The DS peripheral will get the signed result Z by calling RSA to perform $Z = X^Y$ mod M.

18.3.6 DS Operation at the Software Level

The following software steps should be followed each time a Digital Signature needs to be calculated. The inputs are the pre-generated private key ciphertext C, a unique message X, and IV. These software steps trigger the hardware steps described in Section 18.3.5.

1. **Activate the DS peripheral**: Write 1 to DS_SET_START_REG.
2. **Check if DS_KEY is ready**: Poll DS_QUERY_BUSY_REG until it reads 0.

   If DS_QUERY_BUSY_REG does not read 0 after approximately 1 ms, it indicates a problem with HMAC initialization. In such case, software can read register DS_QUERY_KEY_WRONG_REG to get more information.

   - If DS_QUERY_KEY_WRONG_REG reads 0, it indicates that HMAC peripheral was not activated.
   - If DS_QUERY_KEY_WRONG_REG reads any value from 1 to 15, it indicates that HMAC was activated, but the DS peripheral did not successfully receive the DS_KEY value from the HMAC peripheral. This may indicate that the HMAC operation was interrupted due to a software concurrency problem.
3. **Configure register**: Write IV block to register DS_IV_*m*_REG (*m*: 0-3). For more information on IV block, please refer to Chapter 13 *AES Accelerator*.
4. **Write X to memory block DS_X_MEM**: Write X_i ($i \in [0, n) \cap \mathbb{N}$) to memory block DS_X_MEM whose capacity is 128 words. Each word can store one base-b digit. The memory block uses the little endian format for storage, i.e., the least significant digit of the operand is in the lowest address. Words in DS_X_MEM block after the configured length of X (N bits, as described in Section 18.3.2) are ignored.
5. **Write C to memory block DS_C_MEM**: Write C_i ($i \in [0, 396) \cap \mathbb{N}$) to memory block DS_C_MEM whose capacity is 396 words. Each word can store one base-b digit.
6. **Start DS operation**: Write 1 to register DS_SET_ME_REG.
7. **Wait for the operation to be completed**: Poll register DS_QUERY_BUSY_REG until it reads 0.
8. **Query check result**: Read register DS_QUERY_CHECK_REG and determine the subsequent operations based on the return value.

   - If the value is 0, it indicates that both padding check and MD check pass. Users can continue to get the signed result Z.
   - If the value is 1, it indicates that the padding check passes but MD check fails. The signed result Z is invalid. The operation would resume directly from Step 10.
   - If the value is 2, it indicates that the padding check fails but MD check passes. Users can continue to get the signed result Z.

- If the value is 3, it indicates that both padding check and MD check fail. The signed result Z is invalid. The operation would resume directly from Step 10.

9. **Read the signed result**: Read the signed result Z_i ($i \in \{0, 1, 2..., n\}$) from memory block DS_Z_MEM. The memory block stores Z in little-endian byte order.
10. **Exit the operation**: Write 1 to DS_SET_FINISH_REG, then poll DS_QUERY_BUSY_REG until it reads 0.

After the operation, all the input/output registers and memory blocks are cleared.

18.4 Base Address

Users can access the DS peripheral with two base addresses, which can be seen in Table 18-1. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

Table 18-1. Base Address

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F43D000
PeriBUS2	0x6003D000

18.5 Memory Blocks

Both the starting address and ending address in the following table are relative to the DS peripheral base addresses provided in Section 18.4.

Name	Description	Size (byte)	Starting Address	Ending Address	Access
DS_C_MEM	Memory block C	1584	0x0000	0x062F	WO
DS_X_MEM	Memory block X	512	0x0800	0x09FF	WO
DS_Z_MEM	Memory block Z	512	0x0A00	0x0BFF	RO

18.6 Register Summary

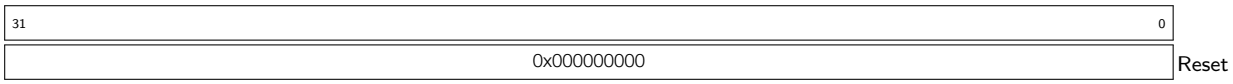
The addresses in the following table are relative to the DS peripheral base addresses provided in Section 18.4.

Name	Description	Address	Access
Configuration Registers			
DS_IV_0_REG	IV block data	0x0630	WO
DS_IV_1_REG	IV block data	0x0634	WO
DS_IV_2_REG	IV block data	0x0638	WO
DS_IV_3_REG	IV block data	0x063C	WO
Status/Control Registers			
DS_SET_START_REG	Activates the DS peripheral	0x0E00	WO
DS_SET_ME_REG	Starts DS operation	0x0E04	WO
DS_SET_FINISH_REG	Ends DS operation	0x0E08	WO
DS_QUERY_BUSY_REG	Status of the DS	0x0E0C	RO

Name	Description	Address	Access
DS_QUERY_KEY_WRONG_REG	Checks the reason why DS_KEY is not ready	0x0E10	RO
DS_QUERY_CHECK_REG	Queries DS check result	0x0814	RO
Version Register			
DS_DATE_REG	Version control egister	0x0820	W/R

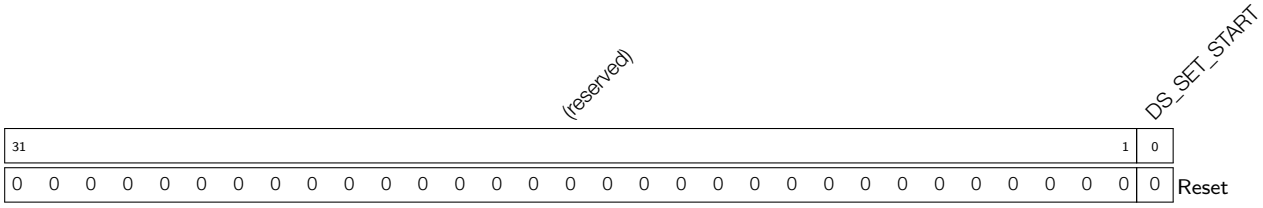
18.7 Registers

Register 18.1: DS_IV_*m*_REG (*m*: 0-3) (0x0630+4m*)**



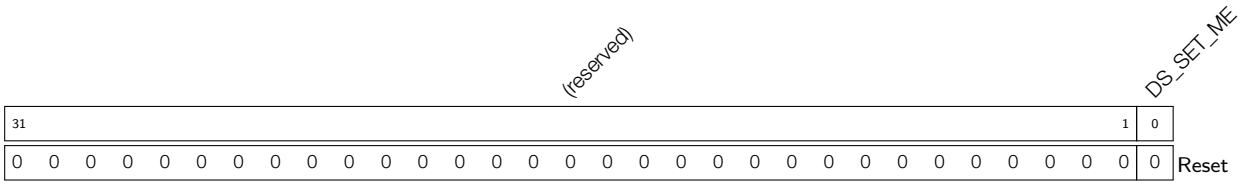
DS_IV_*m*_REG (*m*: 0-3) IV block data. (WO)

Register 18.2: DS_SET_START_REG (0x0E00)



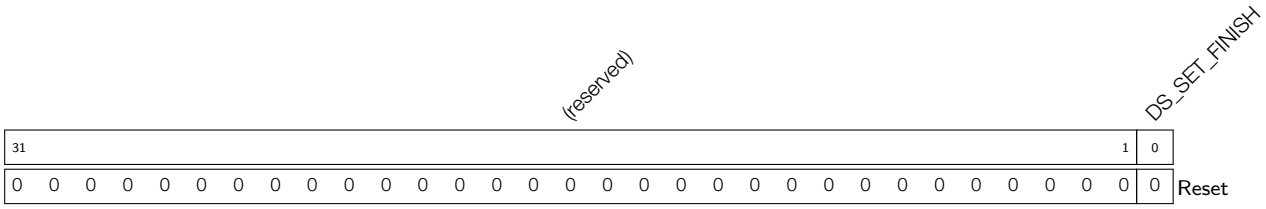
DS_SET_START Write 1 to this register to activate the DS peripheral. (WO)

Register 18.3: DS_SET_ME_REG (0x0E04)



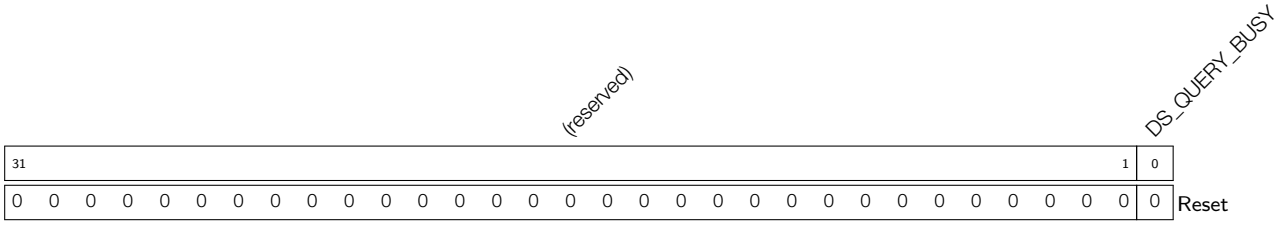
DS_SET_ME Write 1 to this register to start DS operation. (WO)

Register 18.4: DS_SET_FINISH_REG (0x0E08)



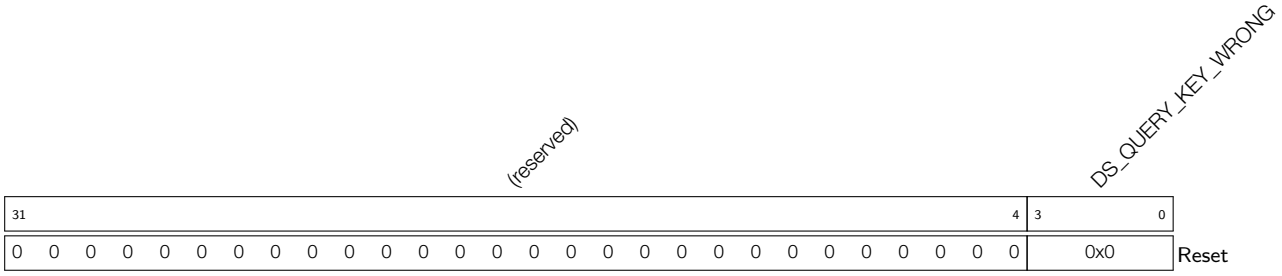
DS_SET_FINISH Write 1 to this register to end DS operation. (WO)

Register 18.5: DS_QUERY_BUSY_REG (0x0E0C)



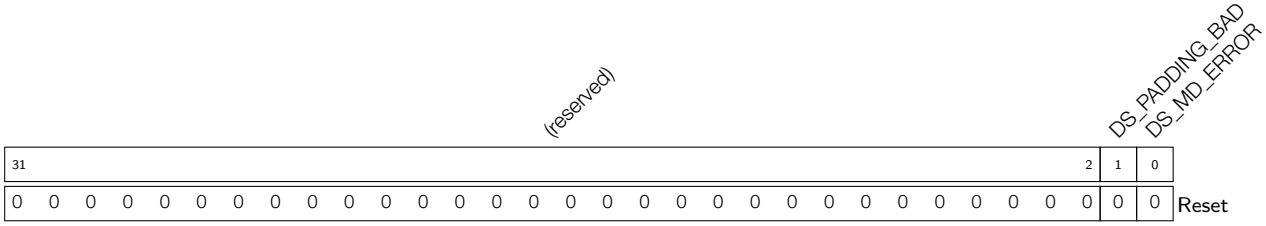
DS_QUERY_BUSY 1: The DS peripheral is busy; 0: The DS peripheral is idle. (RO)

Register 18.6: DS_QUERY_KEY_WRONG_REG (0x0E10)



DS_QUERY_KEY_WRONG 1-15: HMAC was activated, but the DS peripheral did not successfully receive the DS_KEY value from the HMAC peripheral. The biggest value is 15. 0: HMAC is not activated. (RO)

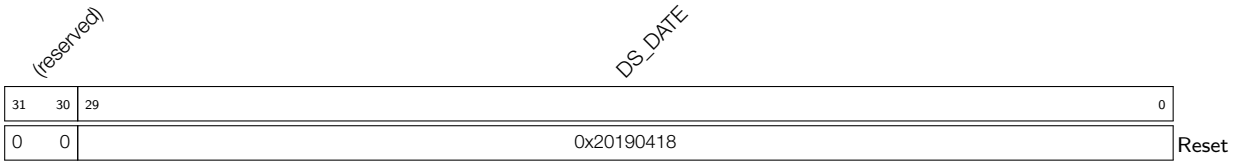
Register 18.7: DS_QUERY_CHECK_REG (0x0E14)



DS_PADDING_BAD 1: The padding check fails; 0: The padding check passes. (RO)

DS_MD_ERROR 1: The MD check fails; 0: The MD check passes. (RO)

Register 18.8: DS_DATE_REG (0x0E20)



DS_DATE Version control register. (R/W)

Revision History

Date	Version	Release notes
2019.11	V0.1	Preliminary release.