

# ESP32-S2

## Technical Reference Manual



Preliminary V0.2  
Espressif Systems  
Copyright © 2020

## 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](http://www.espressif.com/en/subscribe).

## Certification

Download certificates for Espressif products from [www.espressif.com/en/certificates](http://www.espressif.com/en/certificates).

## Disclaimer and Copyright Notice

Information in this document, including URL references, is subject to change without notice. THIS DOCUMENT IS PROVIDED AS IS WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

All liability, including liability for infringement of any proprietary rights, relating to the use of information in this document, is disclaimed. No licenses express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein. The Wi-Fi Alliance Member logo is a trademark of the Wi-Fi Alliance. The Bluetooth logo is a registered trademark of Bluetooth SIG.

All trade names, trademarks and registered trademarks mentioned in this document are property of their respective owners, and are hereby acknowledged.

Copyright © 2020 Espressif Inc. All rights reserved.

## Contents

<b>1 System and Memory</b>	16
1.1 Overview	16
1.2 Features	16
1.3 Functional Description	17
1.3.1 Address Mapping	17
1.3.2 Internal Memory	18
1.3.2.1 Internal ROM 0	19
1.3.2.2 Internal ROM 1	19
1.3.2.3 Internal SRAM 0	19
1.3.2.4 Internal SRAM 1	20
1.3.2.5 RTC FAST Memory	20
1.3.2.6 RTC SLOW Memory	20
1.3.3 External Memory	20
1.3.3.1 External Memory Address Mapping	20
1.3.3.2 Cache	21
1.3.3.3 Cache Operations	21
1.3.4 DMA	22
1.3.5 Modules / Peripherals	23
1.3.5.1 Naming Conventions for Peripheral Buses	23
1.3.5.2 Differences Between PeriBus1 and PeriBus2	23
1.3.5.3 Module / Peripheral Address Mapping	23
1.3.5.4 Addresses with Restricted Access from PeriBus1	25
<b>2 Reset and Clock</b>	26
2.1 Reset	26
2.1.1 Introduction	26
2.1.2 Reset Source	26
2.2 Clock	27
2.2.1 Introduction	27
2.2.2 Clock Source	28
2.2.3 CPU Clock	28
2.2.4 Peripheral Clock	30
2.2.4.1 APB_CLK Source	30
2.2.4.2 REF_TICK Source	30
2.2.4.3 LEDC_PWM_CLK Source	31
2.2.4.4 APLL_SCLK Source	31
2.2.4.5 PLL_160M_CLK Source	31
2.2.4.6 Clock Source Considerations	31
2.2.5 Wi-Fi Clock	32
2.2.6 RTC Clock	32
2.2.7 Audio PLL Clock	32
<b>3 Chip Boot Control</b>	33

3.1	Overview	33
3.2	Boot Mode	33
3.3	ROM Code Printing to UART	34
3.4	VDD_SPI Voltage	34

## 4 Interrupt Matrix

---

4.1	Overview	35
4.2	Features	35
4.3	Functional Description	35
4.3.1	Peripheral Interrupt Sources	35
4.3.2	CPU Interrupts	39
4.3.3	Allocate Peripheral Interrupt Source to CPU Interrupt	40
4.3.3.1	Allocate one peripheral interrupt source Source_X to CPU	40
4.3.3.2	Allocate multiple peripheral interrupt sources Source_Xn to CPU	40
4.3.3.3	Disable CPU peripheral interrupt source Source_X	41
4.3.4	Disable CPU NMI Interrupt Sources	41
4.3.5	Query Current Interrupt Status of Peripheral Interrupt Source	41
4.4	Base Address	41
4.5	Register Summary	41
4.6	Registers	46

## 5 IO MUX and GPIO Matrix

---

5.1	Overview	83
5.2	Peripheral Input via GPIO Matrix	84
5.2.1	Overview	84
5.2.2	Synchronization	84
5.2.3	Functional Description	85
5.2.4	Simple GPIO Input	86
5.3	Peripheral Output via GPIO Matrix	86
5.3.1	Overview	86
5.3.2	Functional Description	87
5.3.3	Simple GPIO Output	87
5.3.4	Sigma Delta Modulated Output	88
5.3.4.1	Functional Description	88
5.3.4.2	SDM Configuration	89
5.4	Direct I/O via IO MUX	89
5.4.1	Overview	89
5.4.2	Functional Description	89
5.5	RTC IO MUX for Low Power and Analog I/O	89
5.5.1	Overview	89
5.5.2	Functional Description	90
5.6	Pin Functions in Light-sleep	90
5.7	Pad Hold Feature	90
5.8	I/O Pad Power Supplies	91
5.8.1	Power Supply Management	91
5.9	Peripheral Signal List	91

5.10 IO MUX Pad List	95
5.11 RTC IO MUX Pin List	97
5.12 Base Address	98
5.13 Register Summary	98
5.13.1 GPIO Matrix Register Summary	98
5.13.2 IO MUX Register Summary	99
5.13.3 Sigma Delta Modulated Output Register Summary	101
5.13.4 RTC IO MUX Register Summary	101
5.14 Registers	102
5.14.1 GPIO Matrix Registers	102
5.14.2 IO MUX Registers	114
5.14.3 Sigma Delta Modulated Output Registers	116
5.14.4 RTC IO MUX Registers	117
<b>6 System Registers</b>	131
6.1 Overview	131
6.2 Features	131
6.3 Function Description	131
6.3.1 System and Memory Registers	131
6.3.2 Reset and Clock Registers	133
6.3.3 Interrupt Matrix Registers	133
6.3.4 JTAG Software Enable Registers	133
6.3.5 Low-power Management Registers	134
6.3.6 Peripheral Clock Gating and Reset Registers	134
6.4 Base Address	136
6.5 Register Summary	136
6.6 Registers	137
<b>7 DMA Controller</b>	152
7.1 Overview	152
7.2 Features	152
7.3 Functional Description	153
7.3.1 DMA Engine Architecture	153
7.3.2 Linked List	153
7.3.3 Enabling DMA	154
7.3.4 Linked List reading process	155
7.3.5 EOF	155
7.3.6 Accessing External RAM	155
7.4 Copy DMA Controller	156
7.5 UART DMA (UDMA) Controller	157
7.6 SPI DMA Controller	157
7.7 I <sup>2</sup> S DMA Controller	158
<b>8 UART Controller</b>	160
8.1 Overview	160
8.2 Features	160

8.3	Functional Description	160
8.3.1	UART Introduction	160
8.3.2	UART Structure	161
8.3.3	UART RAM	162
8.3.4	Baud Rate Generation and Detection	162
8.3.4.1	Baud Rate Generation	162
8.3.4.2	Baud Rate Detection	163
8.3.5	UART Data Frame	164
8.3.6	RS485	165
8.3.6.1	Driver Control	165
8.3.6.2	Turnaround Delay	165
8.3.6.3	Bus Snooping	166
8.3.7	IrDA	166
8.3.8	Wake-up	167
8.3.9	Flow Control	167
8.3.9.1	Hardware Flow Control	168
8.3.9.2	Software Flow Control	169
8.3.10	UDMA	169
8.3.11	UART Interrupts	169
8.3.12	UHCI Interrupts	170
8.4	Base Address	171
8.5	Register Summary	171
8.6	Registers	174

## 9 LED PWM Controller

9.1	Overview	221
9.2	Features	221
9.3	Functional Description	221
9.3.1	Architecture	221
9.3.2	Timers	222
9.3.3	PWM Generators	223
9.3.4	Duty Cycle Fading	223
9.3.5	Interrupts	224
9.4	Base Address	224
9.5	Register Summary	225
9.6	Registers	227

## 10 Remote Control Peripheral

10.1	Introduction	234
10.2	Functional Description	234
10.2.1	RMT Architecture	234
10.2.2	RMT RAM	235
10.2.3	Clock	235
10.2.4	Transmitter	235
10.2.5	Receiver	236
10.2.6	Interrupts	237

10.3 Base Address	237
10.4 Register Summary	237
10.5 Registers	239

## 11 Pulse Count Controller

11.1 Features	248
11.2 Functional Description	249
11.3 Applications	251
11.3.1 Channel 0 Incrementing Independently	251
11.3.2 Channel 0 Decrementing Independently	252
11.3.3 Channel 0 and Channel 1 Incrementing Together	252
11.4 Base Address	253
11.5 Register Summary	253
11.6 Registers	255

## 12 64-bit Timers

12.1 Overview	261
12.2 Functional Description	262
12.2.1 16-bit Prescaler and Clock Selection	262
12.2.2 64-bit Time-based Counter	262
12.2.3 Alarm Generation	262
12.2.4 Timer Reload	262
12.2.5 Interrupts	263
12.3 Configuration and Usage	264
12.3.1 Timer as a Simple Clock	264
12.3.2 Timer as One-shot Alarm	264
12.3.3 Timer as Periodic Alarm	265
12.4 Base Address	265
12.5 Register Summary	265
12.6 Registers	267

## 13 Watchdog Timers

13.1 Overview	276
13.2 Features	276
13.3 Functional Description	276
13.3.1 Clock Source and 32-Bit Counter	276
13.3.2 Stages and Timeout Actions	277
13.3.3 Write Protection	277
13.3.4 Flash Boot Protection	278
13.4 Registers	278

## 14 eFuse Controller

14.1 Overview	279
14.2 Features	279
14.3 Functional Description	279
14.3.1 Structure	279

14.3.1.1	EFUSE_WR_DIS	283
14.3.1.2	EFUSE_RD_DIS	284
14.3.1.3	Data Storage	284
14.3.2	Software Programming of Parameters	284
14.3.3	Software Reading of Parameters	286
14.3.4	Timing	287
14.3.4.1	eFuse-Programming Timing	287
14.3.4.2	eFuse VDDQ Timing Setting	288
14.3.4.3	eFuse-Read Timing	288
14.3.5	The Use of Parameters by Hardware Modules	289
14.3.6	Interrupts	289
14.4	Base Address	290
14.5	Register Summary	290
14.6	Registers	293

## 15 I<sup>2</sup>C Controller

15.1	Overview	316
15.2	Features	316
15.3	I <sup>2</sup> C Functional Description	316
15.3.1	I <sup>2</sup> C Introduction	316
15.3.2	I <sup>2</sup> C Architecture	317
15.3.2.1	TX/RX RAM	318
15.3.2.2	CMD_Controller	318
15.3.2.3	SCL_FSM	320
15.3.2.4	SCL_MAIN_FSM	320
15.3.2.5	DATA_Shifter	320
15.3.2.6	SCL_Filter and SDA_Filter	320
15.3.3	I <sup>2</sup> C Bus Timing	320
15.4	Typical Applications	322
15.4.1	An I <sup>2</sup> C Master Writes to an I <sup>2</sup> C Slave with a 7-bit Address in One Command Sequence	322
15.4.2	An I <sup>2</sup> C Master Writes to an I <sup>2</sup> C Slave with a 10-bit Address in One Command Sequence	324
15.4.3	An I <sup>2</sup> C Master Writes to an I <sup>2</sup> C Slave with Two 7-bit Addresses in One Command Sequence	324
15.4.4	An I <sup>2</sup> C Master Writes to an I <sup>2</sup> C Slave with a 7-bit Address in Multiple Command Sequences	325
15.4.5	An I <sup>2</sup> C Master Reads an I <sup>2</sup> C Slave with a 7-bit Address in One Command Sequence	326
15.4.6	An I <sup>2</sup> C Master Reads an I <sup>2</sup> C Slave with a 10-bit Address in One Command Sequence	327
15.4.7	An I <sup>2</sup> C Master Reads an I <sup>2</sup> C Slave with Two 7-bit Addresses in One Command Sequence	327
15.4.8	An I <sup>2</sup> C Master Reads an I <sup>2</sup> C Slave with a 7-bit Address in Multiple Command Sequences	328
15.5	Clock Stretching	329
15.6	Interrupts	329
15.7	Base Address	330
15.8	Register Summary	330
15.9	Registers	332

## 16 AES Accelerator

16.1	Introduction	354
16.2	Features	354

16.3 Working Modes	354
16.4 Typical AES Working Mode	355
16.4.1 Key, Plaintext, and Ciphertext	356
16.4.2 Endianness	356
16.4.3 Operation Process	360
16.5 DMA-AES Working Mode	361
16.5.1 Key, Plaintext, and Ciphertext	361
16.5.2 Endianness	362
16.5.3 Standard Incrementing Function	363
16.5.4 Block Number	363
16.5.5 Initialization Vector	363
16.5.6 Block Operation Process	363
16.5.7 GCM Operation Process	364
16.6 GCM Algorithm	366
16.6.1 Hash Subkey	367
16.6.2 $J_0$	367
16.6.3 Authenticated Tag	367
16.6.4 AAD Block Number	367
16.6.5 Remainder Bit Number	368
16.7 Base Address	368
16.8 Memory Summary	368
16.9 Register Summary	368
16.10 Registers	370

## 17 SHA Accelerator

17.1 Introduction	375
17.2 Features	375
17.3 Working Modes	375
17.4 Function Description	376
17.4.1 Preprocessing	376
17.4.1.1 Padding the Message	376
17.4.1.2 Parsing the Message	377
17.4.1.3 Initial Hash Value	378
17.4.2 Hash Computation Process	379
17.4.2.1 Typical SHA Process	379
17.4.2.2 DMA-SHA Process	381
17.4.3 Message Digest	383
17.4.4 Interrupt	383
17.5 Base Address	384
17.6 Register Summary	384
17.7 Registers	386

## 18 RSA Accelerator

18.1 Introduction	390
18.2 Features	390
18.3 Functional Description	390

18.3.1 Large Number Modular Exponentiation	391
18.3.2 Large Number Modular Multiplication	392
18.3.3 Large Number Multiplication	393
18.3.4 Acceleration Options	393
18.4 Base Address	395
18.5 Memory Summary	395
18.6 Register Summary	395
18.7 Registers	396
<b>19 Random Number Generator</b>	400
19.1 Introduction	400
19.2 Features	400
19.3 Functional Description	400
19.4 Base Address	401
19.5 Register Summary	401
19.6 Register	401
<b>20 External Memory Encryption and Decryption</b>	402
20.1 Overview	402
20.2 Features	402
20.3 Functional Description	402
20.3.1 XTS Algorithm	403
20.3.2 Key	403
20.3.3 Target Memory Space	404
20.3.4 Data Padding	404
20.3.5 Manual Encryption Block	405
20.3.6 Auto Encryption Block	406
20.3.7 Auto Decryption Block	407
20.4 Base Address	407
20.5 Register Summary	407
20.6 Registers	408
<b>21 Digital Signature</b>	412
21.1 Overview	412
21.2 Features	412
21.3 Functional Description	412
21.3.1 Overview	412
21.3.2 Private Key Operands	412
21.3.3 Conventions	413
21.3.4 Software Storage of Private Key Data	413
21.3.5 DS Operation at the Hardware Level	414
21.3.6 DS Operation at the Software Level	415
21.4 Base Address	416
21.5 Memory Blocks	416
21.6 Register Summary	416
21.7 Registers	417

## Revision History

420

## List of Tables

1	Address Mapping	18
2	Internal Memory Address Mapping	18
3	External Memory Address Mapping	21
4	Peripherals with DMA Support	22
5	Module / Peripheral Address Mapping	23
6	Addresses with Restricted Access	25
7	Reset Source	27
8	CPU_CLK Source	29
9	CPU_CLK Selection	29
10	Peripheral Clock Usage	30
11	APB_CLK Source	30
12	REF_TICK Source	31
13	LEDC_PWM_CLK Source	31
14	Default configuration of strapping pins	33
15	Boot Mode	33
16	ROM Code Printing Control	34
17	CPU peripheral interrupt configuration/status registers and peripheral interrupt sources	36
18	CPU Interrupts	39
19	Interrupt Matrix Base Address	41
21	Pin Function Register for IO MUX Light-sleep Mode	90
22	GPIO Matrix	91
23	IO MUX Pad List	95
24	RTC IO MUX Pin Summary	97
25	GPIO and RTCIO Base Address	98
30	ROM Controlling Bit	132
31	SRAM Controlling Bit	132
32	Peripheral Clock Gating and Reset Bits	134
33	System Register Base Address	136
35	Relationship Between Configuration Register and Destination Address	156
36	Base addresses of UART0, UART1 and UHCI0	171
39	LED_PWM Base Address	225
41	RMT Base Address	237
43	Counter Mode. Positive Edge of Input Pulse Signal. Control Signal in Low State	250
44	Counter Mode. Positive Edge of Input Pulse Signal. Control Signal in High State	250
45	Counter Mode. Negative Edge of Input Pulse Signal. Control Signal in Low State	250
46	Counter Mode. Negative Edge of Input Pulse Signal. Control Signal in High State	250
47	PCNT Base Address	253
49	64-bit Timers Base Address	265
51	Parameters in BLOCK0	279
52	Key Purpose Values	282
53	Parameters in BLOCK1-10	282
55	Configuration of eFuse-Programming Timing Parameters	287
56	Configuration of VDDQ Timing Parameters	288
57	Configuration of eFuse-Reading Parameters	288

58	eFuse Controller Base Address	290
60	I <sup>2</sup> C Controller Base Address	330
62	AES Accelerator Working Mode	355
63	Operation Type under Typical AES Working Mode	355
64	Working Status under Typical AES Working Mode	355
65	Text Endianness Types for Typical AES	356
66	Key Endianness Types for AES-128 Encryption and Decryption	358
67	Key Endianness Types for AES-192 Encryption and Decryption	358
68	Key Endianness Types for AES-256 Encryption and Decryption	359
69	Operation Type under DMA-AES Working Mode	361
70	Working Status under DMA-AES Working mode	361
71	TEXT-PADDING	362
72	Text Endianness for DMA-AES	362
73	AES Accelerator Base Address	368
76	SHA Accelerator Working Mode	376
77	SHA Hash Algorithm	376
81	The Storage and Length of Message Digests from Different Algorithms	383
82	Base Address	384
84	Acceleration Performace	394
85	RSA Base Address	395
88	Random Number Generator Base Address	401
90	Key	404
91	Mapping Between Offsets and Registers	405
92	Manual Encryption Block Base Address	407
94	Base Address	416

## List of Figures

1-1	System Structure and Address Mapping	17
1-2	Cache Structure	21
2-1	System Reset	26
2-2	System Clock	28
4-1	Interrupt Matrix Structure	35
5-1	IO MUX, RTC IO MUX and GPIO Matrix Overview	83
5-2	GPIO Input Synchronized on Clock Rising Edge or on Falling Edge	85
5-3	Filter Timing Diagram of GPIO Input Signals	85
7-1	Modules with DMA and Supported Data Transfers	152
7-2	DMA Engine Architecture	153
7-3	Structure of a Linked List	153
7-4	Relationship among Linked Lists	155
7-5	Copy DMA Engine Architecture	156
7-6	Data Transfer in UDMA Mode	157
7-7	SPI DMA	157
8-1	UART Structure	161
8-2	UART Controllers Sharing RAM	162
8-3	UART Controllers Division	163
8-4	The Timing Diagram of Weak UART Signals Along Falling Edges	164
8-5	Structure of UART Data Frame	164
8-6	AT_CMD Character Structure	165
8-7	Driver Control Diagram in RS485 Mode	166
8-8	The Timing Diagram of Encoding and Decoding in SIR mode	167
8-9	IrDA Encoding and Decoding Diagram	167
8-10	Hardware Flow Control Diagram	168
8-11	Connection between Hardware Flow Control Signals	168
9-1	LED_PWM Architecture	221
9-2	LED_PWM generator Diagram	222
9-3	LED_PWM Divider	222
9-4	LED_PWM Output Signal Diagram	223
9-5	Output Signal Diagram of Fading Duty Cycle	224
10-1	RMT Architecture	234
10-2	Format of Pulse Code in RAM	235
11-1	PCNT Block Diagram	248
11-2	PCNT Unit Architecture	249
11-3	Channel 0 Up Counting Diagram	251
11-4	Channel 0 Down Counting Diagram	252
11-5	Two Channels Up Counting Diagram	252
12-1	Timer Units within Groups	261
14-1	Shift Register Circuit	284
14-2	eFuse-Programming Timing Diagram	289
14-3	Timing Diagram for Reading eFuse	289
15-1	I <sup>2</sup> C Master Architecture	317
15-2	I <sup>2</sup> C Slave Architecture	317

15-3 Structure of I <sup>2</sup> C Command Register	318
15-4 I <sup>2</sup> C Timing Diagram	321
15-5 An I <sup>2</sup> C Master Writing to an I <sup>2</sup> C Slave with a 7-bit Address	322
15-6 A Master Writing to a Slave with a 10-bit Address	324
15-7 An I <sup>2</sup> C Master Writing Address M in the RAM to an I <sup>2</sup> C Slave with a 7-bit Address	324
15-8 An I <sup>2</sup> C Master Writing to an I <sup>2</sup> C Slave with a 7-bit Address in Multiple Sequences	325
15-9 An I <sup>2</sup> C Master Reading an I <sup>2</sup> C Slave with a 7-bit Address	326
15-10 An I <sup>2</sup> C Master Reading an I <sup>2</sup> C Slave with a 10-bit Address	327
15-11 An I <sup>2</sup> C Master Reading N Bytes of Data from addrM of an I <sup>2</sup> C Slave with a 7-bit Address	327
15-12 An I <sup>2</sup> C Master Reading an I <sup>2</sup> C Slave with a 7-bit Address in Segments	328
16-1 GCM Encryption Process	366
19-1 Noise Source	400
20-1 Architecture of the External Memory Encryption and Decryption Module	402
21-1 Preparations and DS Operation	413

# 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.

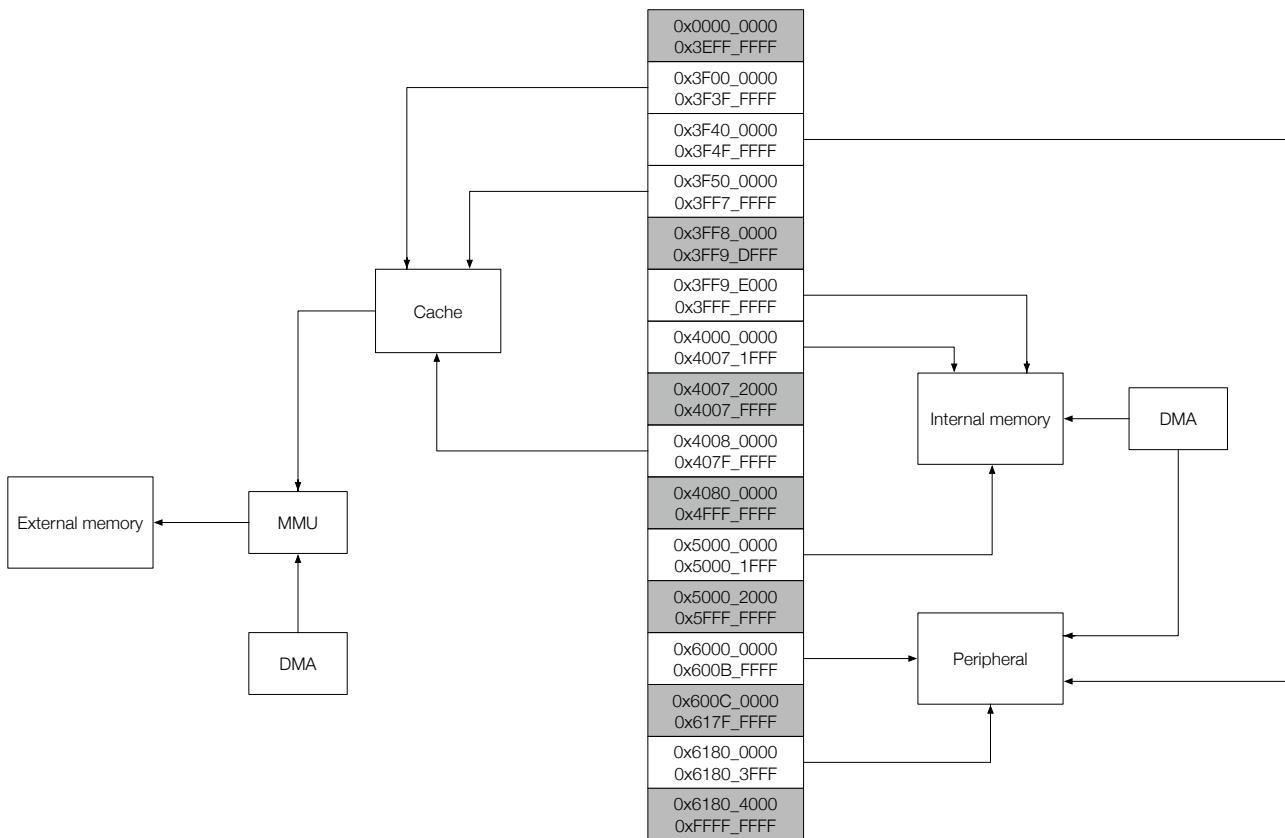


Figure 1-1. System Structure and Address Mapping

**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 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: Address Mapping**

Bus Type	Boundary Address		Size	Target
	Low Address	High Address		
Data bus	0x0000_0000	0x3EFF_FFFF		Reserved
	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 2 lists all types of internal memory and their address ranges on the data bus and instruction bus.

**Table 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 Low Address	High Address	Size	Target	Permission Control
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 Low Address	High Address	Size	Target	Permission Control
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 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 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 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 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 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 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 3 lists the mapping between the cache and the corresponding address ranges on the data bus and instruction bus.

**Table 3: External Memory Address Mapping**

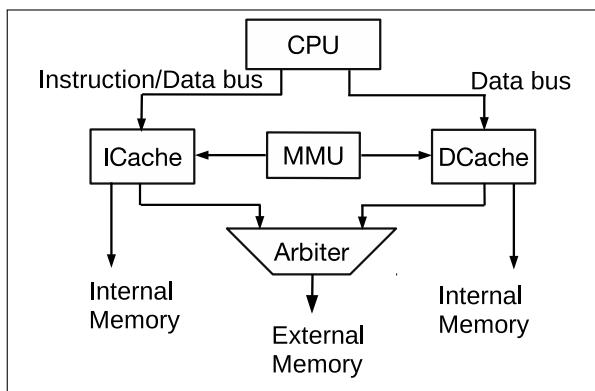
Bus Type	Boundary Address		Size	Target	Permission Control
	Low Address	High Address			
Data bus	0x3F00_0000	0x3F3F_FFFF	4 MB	ICache	YES
Data bus	0x3F50_0000	0x3FF7_FFFF	10.5 MB	DCache	YES
Instruction bus	0x4008_0000	0x407F_FFFF	7.5 MB	ICache	YES

**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:

- 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.
- 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 4. With DMA, some of them can only access internal memory, some can access both internal and external memory.

**Table 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 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 5: Module / Peripheral Address Mapping**

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

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

**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 [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 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

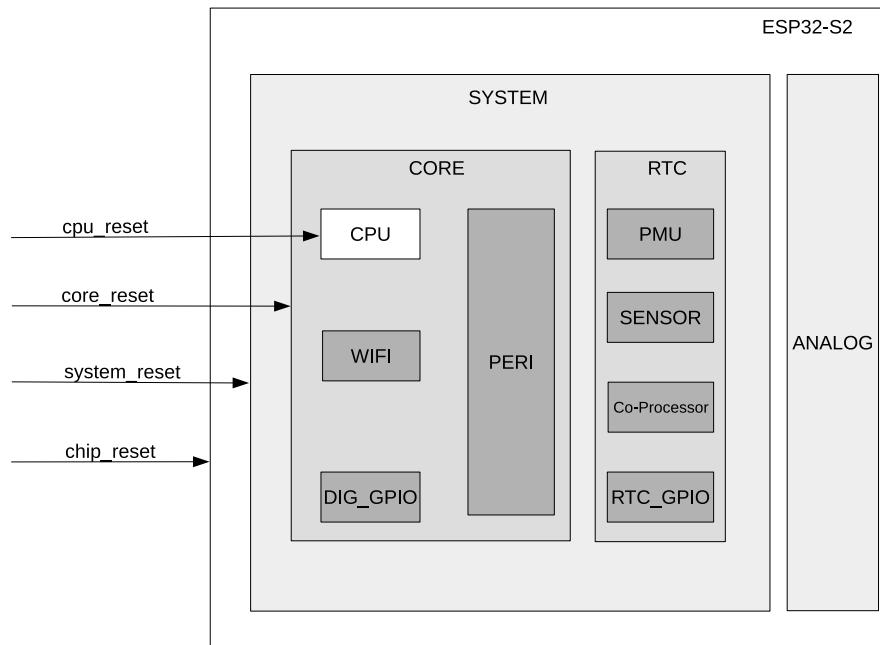
## 2. Reset and Clock

### 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 7 lists different reset sources and the types of reset they trigger.

**Table 7: 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 <a href="#">13 Watchdog Timers</a>
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 <a href="#">13 Watchdog Timers</a>
0x08	MWDT1 global reset	Core Reset	See Chapter <a href="#">13 Watchdog Timers</a>
0x09	RWDT core reset	Core Reset	See Chapter <a href="#">13 Watchdog Timers</a>
0x0B	MWDT0 CPU reset	CPU Reset	See Chapter <a href="#">13 Watchdog Timers</a>
0x0C	Software CPU reset	CPU Reset	Triggered by configuring RTC_CNTL_SW_PROCPU_RST
0x0D	RWDT CPU reset	CPU Reset	See Chapter <a href="#">13 Watchdog Timers</a>
0x11	MWDT1 CPU reset	CPU Reset	See Chapter <a href="#">13 Watchdog Timers</a>

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.

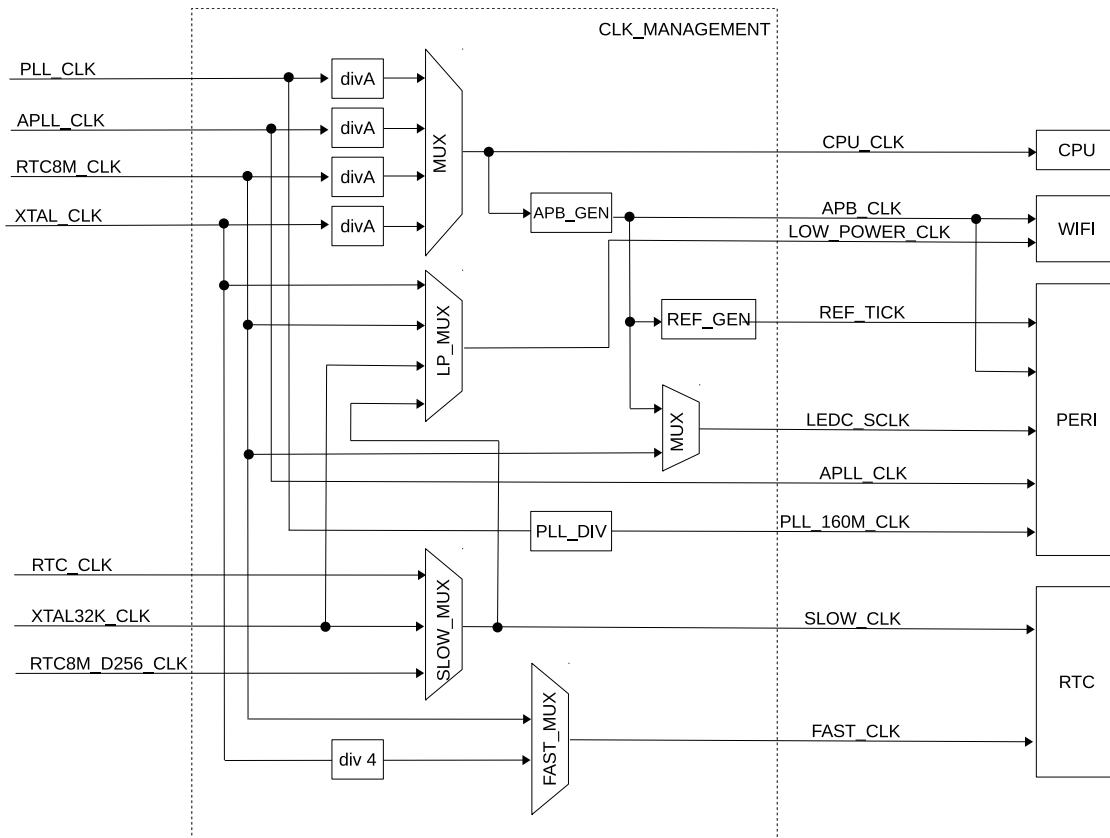


Figure 2-2. System Clock

### 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 8 and Table 9.

**Table 8: CPU\_CLK Source**

<a href="#">SYSTEM_SOC_CLK_SEL</a> Value	Clock Source
0	XTAL_CLK
1	PLL_CLK
2	RTC8M_CLK
3	APLL_CLK

**Table 9: CPU\_CLK Selection**

Clock Source	SEL_0*	SEL_1*	SEL_2*	CPU Clock Frequency
XTAL_CLK	0	-	-	$\text{CPU\_CLK} = \text{XTAL\_CLK} / (\text{SYSTEM\_PRE\_DIV\_CNT} + 1)$ <a href="#">SYSTEM_PRE_DIV_CNT</a> ranges from 0 ~ 1023. Default is 1
PLL_CLK (480 MHz)	1	1	0	$\text{CPU\_CLK} = \text{PLL\_CLK} / 6$ CPU_CLK frequency is 80 MHz
PLL_CLK (480 MHz)	1	1	1	$\text{CPU\_CLK} = \text{PLL\_CLK} / 3$ CPU_CLK frequency is 160 MHz
PLL_CLK (480 MHz)	1	1	2	$\text{CPU\_CLK} = \text{PLL\_CLK} / 2$ CPU_CLK frequency is 240 MHz
PLL_CLK (320 MHz)	1	0	0	$\text{CPU\_CLK} = \text{PLL\_CLK} / 4$ CPU_CLK frequency is 80 MHz
PLL_CLK (320 MHz)	1	0	1	$\text{CPU\_CLK} = \text{PLL\_CLK} / 2$ CPU_CLK frequency is 160 MHz
RTC8M_CLK	2	-	-	$\text{CPU\_CLK} = \text{RTC8M\_CLK} / (\text{SYSTEM\_PRE\_DIV\_CNT} + 1)$ <a href="#">SYSTEM_PRE_DIV_CNT</a> ranges from 0 ~ 1023. Default is 1
APLL_CLK	3	0	0	$\text{CPU\_CLK} = \text{APLL\_CLK} / 4$
APLL_CLK	3	0	1	$\text{CPU\_CLK} = \text{APLL\_CLK} / 2$

\*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 10 shows which clock can be used by which peripheral.

**Table 10: Peripheral Clock Usage**

Peripheral	APB_CLK	REF_TICK	LEDC_PWM_CLK	APLL_CLK	PLL_160M_CLK
TIMG	Y	Y			
I <sup>2</sup> S	Y			Y	Y
UHCI	Y				
UART	Y	Y			
RMT	Y	Y			
LED_PWM	Y	Y	Y		
I <sup>2</sup> C	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 11.

**Table 11: 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 12 shows the configuration of these clock divider registers.

**Table 12: 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  $1 \mu s$ , 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 13.

**Table 13: 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 I<sup>2</sup>C, 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 [10](#).

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 I<sup>2</sup>S module.

Audio PLL formula is as follows:

$$f_{\text{out}} = \frac{f_{\text{xtal}}(\text{sdm}2 + \frac{\text{sdm}1}{2^8} + \frac{\text{sdm}0}{2^{16}} + 4)}{2(\text{odiv} + 2)}$$

Parameters are defined below:

- $f_{\text{xtal}}$ : 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: the value is 0 ~ 31;
- The operating frequency range of the numerator is 350 MHz ~ 500 MHz.

$$350\text{MHz} < f_{\text{xtal}}(\text{sdm}2 + \frac{\text{sdm}1}{2^8} + \frac{\text{sdm}0}{2^{16}} + 4) < 500\text{MHz}$$

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 14).

**Table 14: 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 15: Boot Mode**

Pin	SPI Boot	Download Boot
GPIO0	1	0
GPIO46	x	0

Table 15 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 16: 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.

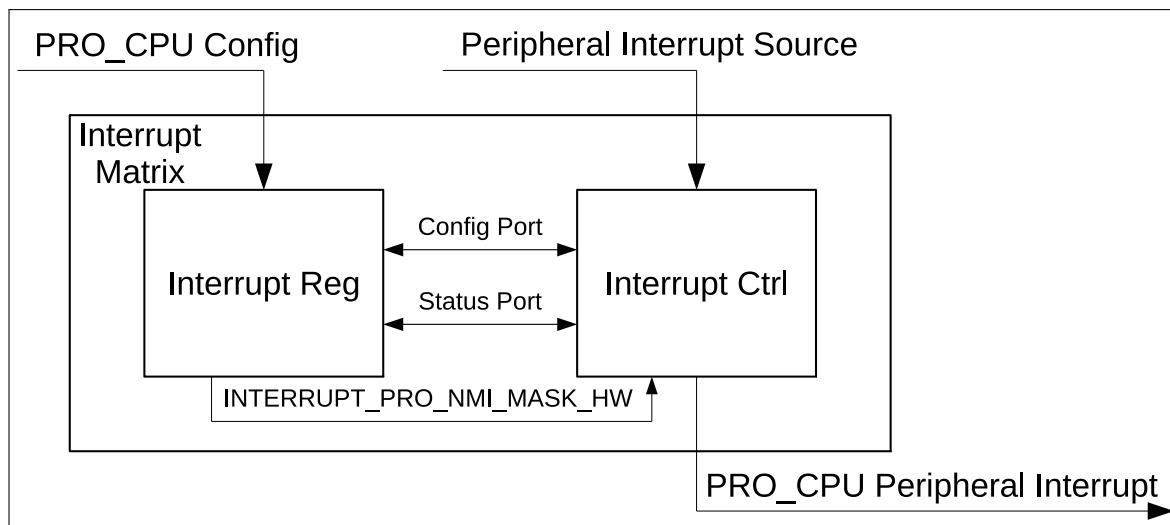


Figure 4-1. Interrupt Matrix Structure

### 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 17.

Table 17: CPU peripheral interrupt configuration/status registers and peripheral interrupt sources

No.	Source	Configuration Register	Bit	Status Register Name
0	MAC_INTR	INTERRUPT_PRO_MAC_INTR_MAP_REG	0	
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_TO_LEVEL_INT	INTERRUPT_PRO_TG_TO_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	
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	

INTERRUPT\_PRO\_INTR\_STATUS\_REG\_0\_REG

INTERRUPT\_PRO\_INTR\_STATUS\_REG\_1\_REG

No.	Source	Configuration Register	Bit	Status Register 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	
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_TO_EDGE_INT	INTERRUPT_PRO_TG_TO_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	
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	Bit	Status Register 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 18 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 18: 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 17.
- INTERRUPT\_PRO\_X\_MAP\_REG: stands for a peripheral interrupt configuration register, corresponding to the peripheral interrupt source Source\_X. In Table 17, 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 18).
- 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 18).

#### 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\_Xn to CPU

Setting the corresponding configuration register INTERRUPT\_PRO\_Xn\_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 17.

## 4.4 Base Address

Users can access interrupt matrix with the base address as shown in Table 19. For more information, see Chapter 1 *System and Memory*.

**Table 19: 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
<b>Configuration registers</b>			
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_LED_C_INT_MAP_REG	LED_C_INT 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
<b>Interrupt status registers</b>			
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
<b>Version register</b>			
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)**

The diagram shows the memory map for the INTERRUPT\_PRO\_MAC\_INTR\_MAP\_REG register. It consists of a 32-bit wide register divided into four main sections: a 31-bit field from bit 31 down to bit 0, a 5-bit field from bit 4 down to bit 0, a 4-bit field from bit 0 down to bit 0, and a 16-bit field from bit 16 down to bit 0. The 31-bit field is labeled '(reserved)'. The 5-bit field is labeled 'INTERRUPT\_PRO\_MAC\_INTR\_MAP'. The 4-bit field is labeled '0'. The 16-bit field is labeled 'Reset'.

31		5	4	0
0 0	16	Reset		

**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)**

The diagram shows the memory map for the INTERRUPT\_PRO\_MAC\_NMI\_MAP\_REG register. It consists of a 32-bit wide register divided into four main sections: a 31-bit field from bit 31 down to bit 0, a 5-bit field from bit 4 down to bit 0, a 4-bit field from bit 0 down to bit 0, and a 16-bit field from bit 16 down to bit 0. The 31-bit field is labeled '(reserved)'. The 5-bit field is labeled 'INTERRUPT\_PRO\_MAC\_NMI\_MAP'. The 4-bit field is labeled '0'. The 16-bit field is labeled 'Reset'.

31		5	4	0
0 0	16	Reset		

**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

(reserved)																																
31																																
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	0	0	0	0	0	0	0	

**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

(reserved)																																	
31																																	
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	0	0	0	0	0	0	0		

**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

(reserved)																																	
31																																	
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	0	0	0	0	0	0	0		

**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

(reserved)																														
31																														
0 0																														

**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

(reserved)																															
31																															
0 0																															

**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

(reserved)																															
31																															
0 0																															

**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

31	(reserved)												5	4	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0												16	Reset		

**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

31	(reserved)												5	4	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0												16	Reset		

**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

31	(reserved)												5	4	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0												16	Reset		

**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)**

The diagram shows a 32-bit register layout for Register 4.12. The register is divided into four main sections: a 31-bit field at the top left, a 5-bit field at the top right, a 4-bit field below it, and a 16-bit field at the bottom right. The 31-bit field is labeled '(reserved)'. The 5-bit field is labeled '5' and contains the value '4'. The 4-bit field is labeled '4' and contains the value '0'. The 16-bit field is labeled '16' and contains the value 'Reset'. The entire register is labeled 'INTERRUPT\_PRO\_SLC0\_INTR\_MAP' diagonally across the top.

31	5	4	0
0 0	16	Reset	

**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)**

The diagram shows a 32-bit register layout for Register 4.13. The register is divided into four main sections: a 31-bit field at the top left, a 5-bit field at the top right, a 4-bit field below it, and a 16-bit field at the bottom right. The 31-bit field is labeled '(reserved)'. The 5-bit field is labeled '5' and contains the value '4'. The 4-bit field is labeled '4' and contains the value '0'. The 16-bit field is labeled '16' and contains the value 'Reset'. The entire register is labeled 'INTERRUPT\_PRO\_SLC1\_INTR\_MAP' diagonally across the top.

31	5	4	0
0 0	16	Reset	

**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)**

The diagram shows a 32-bit register layout for Register 4.14. The register is divided into four main sections: a 31-bit field at the top left, a 5-bit field at the top right, a 4-bit field below it, and a 16-bit field at the bottom right. The 31-bit field is labeled '(reserved)'. The 5-bit field is labeled '5' and contains the value '4'. The 4-bit field is labeled '4' and contains the value '0'. The 16-bit field is labeled '16' and contains the value 'Reset'. The entire register is labeled 'INTERRUPT\_PRO\_UHCI0\_INTR\_MAP' diagonally across the top.

31	5	4	0
0 0	16	Reset	

**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

31	(reserved)																5	4	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset	

**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

31	(reserved)																5	4	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset	

**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

31	(reserved)																5	4	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset	

**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)**

31					5	4	0
0	0	0	0	0	0	0	16 Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	16 Reset

**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)**

(reserved)																															INTERRUPT_PRO_TG1_T0_LEVEL_INT_MAP
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	0	0	0	0	0	Reset	

**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)**

(reserved)																																				INTERRUPT_PRO_TG1_T1_LEVEL_INT_MAP
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	0	0	0	0	0	Reset						

**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)**

(reserved)																														INTERRUPT_PRO_TG1_WDT_LEVEL_INT_MAP			
31																														5	4	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	0	0	0	0	0	0	16	Reset		

**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)**

(reserved)																															INTERRUPT_PRO_TG1_LACT_LEVEL_INT_MAP			
31																													5	4	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	0	0	0	0	0	0	16	Reset			

**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

(reserved)																															
31																															
0 0																															

**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

(reserved)																															
31																															
0 0																															

**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

(reserved)																															
31																															
0 0																															

**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)**

(reserved)																INTERRUPT_PRO_GPIO_INTERRUPT_APP_NMI_MAP			
31																5   4	0	Reset	
0 0																16			

**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)**

(reserved)																INTERRUPT_PRO_DEDICATED_GPIO_IN_INTR_MAP			
31																5   4	0	Reset	
0 0																16			

**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)**

(reserved)																INTERRUPT_PRO_CPU_INTR_FROM_CPU_0_MAP			
31																5   4	0	Reset	
0 0																16			

**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)**

31																														5   4	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	0	0	0	0	0	0	16	Reset

**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)**

31																														5   4	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	0	0	0	0	0	0	16	Reset

**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)**

31																														5   4	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	0	0	0	0	0	0	16	Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

INTERRUPT\_PRO\_I2S0\_INT\_MAP

Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

INTERRUPT\_PRO\_I2S1\_INT\_MAP

Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

INTERRUPT\_PRO\_UART\_INTR\_MAP

Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	Reset

**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)**

(reserved)																INTERRUPT_PRO_PWM0_INTR_MAP	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																5   4      0	16      Reset
31																	

**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)**

(reserved)																INTERRUPT_PRO_PWM1_INTR_MAP	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																5   4      0	16      Reset
31																	

**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)**

(reserved)																INTERRUPT_PRO_PWM2_INTR_MAP	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																5   4      0	16      Reset
31																	

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

INTERRUPT\_PRO\_PWM3\_INTR\_MAP

Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

INTERRUPT\_PRO\_LED\_C\_INT\_MAP

Reset

**INTERRUPT\_PRO\_LED\_C\_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)**

31					5	4	0
0	0	0	0	0	0	0	0

INTERRUPT\_PRO\_EFUSE\_INT\_MAP

Reset

**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)**

(reserved)																INTERRUPT_PRO_CAN_INT_MAP			
0 0																5   4 0	16	Reset	
31																INTERRUPT_PRO_CAN_INT_MAP			

**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)**

(reserved)																INTERRUPT_PRO_USB_INTR_MAP			
0 0																5   4 0	16	Reset	
31																INTERRUPT_PRO_USB_INTR_MAP			

**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)**

(reserved)																INTERRUPT_PRO_RTC_CORE_INTR_MAP			
0 0																5   4 0	16	Reset	
31																INTERRUPT_PRO_RTC_CORE_INTR_MAP			

**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)**

31																	5   4	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset

**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)**

31																	5   4	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset

**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)**

31																	5   4	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset

**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)**

(reserved)																INTERRUPT_PRO_I2C_EXT1_INTR_MAP				
																5	4	0	16	Reset
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset

**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)**

(reserved)																INTERRUPT_PRO_RSA_INTR_MAP				
																5	4	0	16	Reset
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset

**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)**

(reserved)																INTERRUPT_PRO_SHA_INTR_MAP				
																5	4	0	16	Reset
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

16      Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

16      Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

16      Reset

**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)**

31				5	4	0
0	0	0	0	0	0	0

INTERRUPT\_PRO\_WDG\_INT\_MAP

Reset

**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)**

31				5	4	0
0	0	0	0	0	0	0

INTERRUPT\_PRO\_TIMER\_INT1\_MAP

Reset

**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)**

31				5	4	0
0	0	0	0	0	0	0

INTERRUPT\_PRO\_TIMER\_INT2\_MAP

Reset

**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

(reserved)																																	
31																																	
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	0	0	0	0	0	0	0	0	0

**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

(reserved)																																	
31																																	
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	0	0	0	0	0	0	0	0	0

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

16      Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	0

16      Reset

**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)**

(reserved)																INTERRUPT_PRO_TG1_T0_EDGE_INT_MAP			
31																5	4	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**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)**

(reserved)																INTERRUPT_PRO_TG1_T1_EDGE_INT_MAP			
31																5	4	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**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)**

(reserved)																													
31																													
0 0																													

**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)**

(reserved)																														
31																														
0 0																														

**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

(reserved)																																	
31																																	
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	0	0	0	0	0	0	0	0	0

**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

(reserved)																																	
31																																	
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	0	0	0	0	0	0	0	0	0

**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

(reserved)																																	
31																																	
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	0	0	0	0	0	0	0	0	0

**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)**

(reserved)																INTERRUPT_PRO_SYSTIMER_TARGET2_INT_MAP						
31																5	4	0	Reset			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset		

**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)**

(reserved)																INTERRUPT_PRO_ASSIST_DEBUG_INTR_MAP						
31																5	4	0	Reset			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset		

**INTERRUPT\_PRO\_ASSIST\_DEBUG\_INTR\_MAP** This register is used to map ASIST\_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)**

(reserved)																INTERRUPT_PRO_PMS_PRO_IRAM0_ILG_INTR_MAP						
31																5	4	0	Reset			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	Reset		

**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)**

(reserved)																INTERRUPT_PRO_PMS_PRO_DRAM0_ILG_INTR_MAP		
31																5   4	0	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																16	Reset	

**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)**

(reserved)																INTERRUPT_PRO_PMS_PRO_DPORT_ILG_INTR_MAP		
31																5   4	0	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																16	Reset	

**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)**

(reserved)																INTERRUPT_PRO_PMS_PRO_AHB_ILG_INTR_MAP		
31																5   4	0	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																16	Reset	

**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)**

(reserved)																																
31																																

**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)**

(reserved)																																
31																																

**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)**

(reserved)																																
31																																

**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)**

(reserved)																														
31																														
0 0																														

**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)**

(reserved)																															
31																															
0 0																															

**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)**

(reserved)																															
31																															
0 0																															

**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)**

31					5	4	0
0	0	0	0	0	0	0	16 Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	16 Reset

**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)**

31					5	4	0
0	0	0	0	0	0	0	16 Reset

**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

(reserved)																																
31																																

**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

(reserved)																																
31																																

**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)**

(reserved)	5	4	0
0 0	16	Reset	

**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)**

(reserved)	5	4	0
0 0	16	Reset	

**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)**

31					4	0
0	0	0	0	0	0	0

INTERRUPT\_PRO\_APB\_PERI\_ERROR\_INT\_MAP

**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)**

31					4	0
0	0	0	0	0	0	0

INTERRUPT\_PRO\_DCACHE\_SYNC\_INT\_MAP

**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)

31		5	4	0
0	0	0	0	16

**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; 0: 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)

31		0
	0x000000	Reset

**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	
31	0
0x000000	Reset

**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	
31	0
0x000000	Reset

**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_INTERRUPT_REG_DATE	
(reserved)	
31      28   27	0
0    0    0    0	Reset
0x1904180	

**INTERRUPT\_DATE** Version control register. (R/W)

## 5. IO MUX and GPIO Matrix

### 5.1 Overview

The ESP32-S2 chip features 43 physical GPIO pads. Each pad can be used as a general-purpose I/O, or be connected to an internal peripheral signal. The IO MUX, RTC IO MUX and the GPIO matrix are responsible for routing signals from the peripherals to GPIO pads. Together these modules provide highly configurable I/O.

**Note that the GPIO pads are numbered from 0 ~ 21 and 26 ~ 46, while GPIO46 is input-only.**

This chapter describes the selection and connection of the internal signals for the 43 digital pads and control signals: FUN\_SEL, IE, OE, WPU, WPD, etc. These internal signals include:

- 116 digital peripheral input signals, control signals: SIG\_IN\_SEL, SIG\_OUT\_SEL, IE, OE, etc.
- 182 digital peripheral output signals, control signals: SIG\_IN\_SEL, SIG\_OUT\_SEL, IE, OE, etc.
- fast peripheral input and output signals, control signals: IE, OE, etc.
- 22 RTC GPIO signals

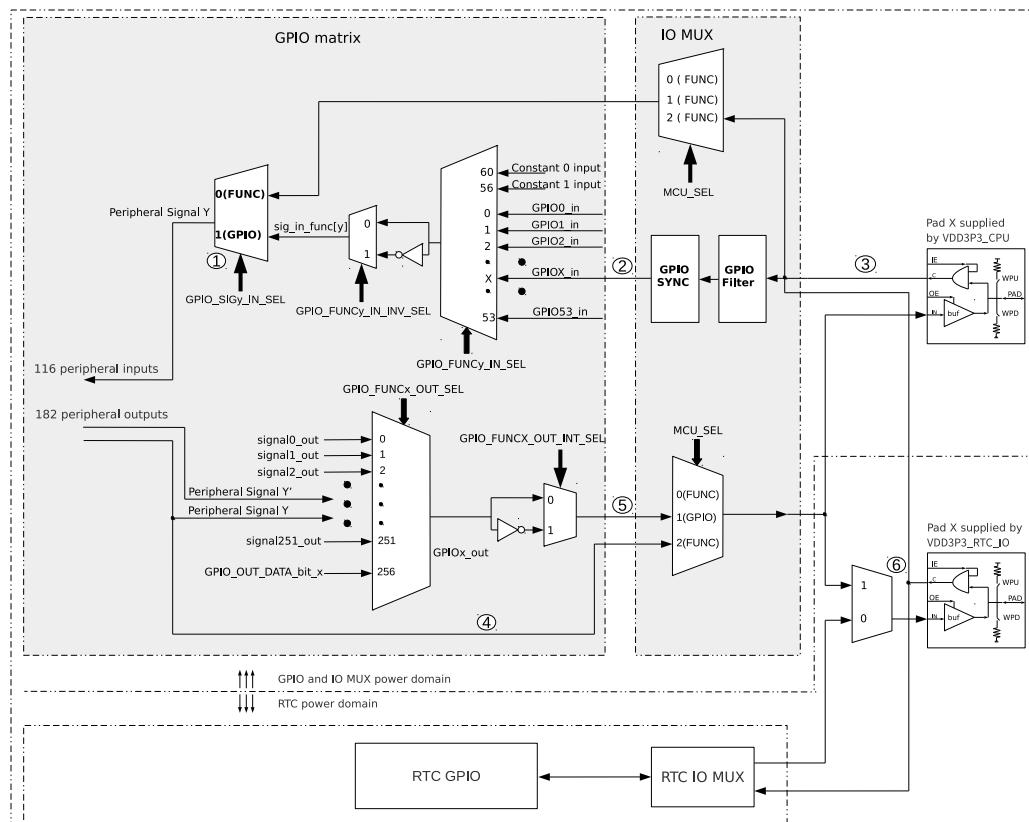


Figure 5-1. IO MUX, RTC IO MUX and GPIO Matrix Overview

Figure 5-1 shows the overview of IO MUX, RTC IO MUX and GPIO matrix.

1. IO MUX provides one configuration register `IO_MUX_n_REG` for each GPIO pad. The pad can be configured to

- perform GPIO function routed by GPIO matrix;
- or perform direct connection bypassing GPIO matrix.

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

See Section [5.10](#) for the IO MUX functions for each I/O pad.

2. GPIO matrix is a full-switching matrix between the peripheral input/output signals and the pads.

- For input to the chip: each of the 116 internal peripheral inputs can select any GPIO pad as their input source.
- For output from the chip: each GPIO pad can select any of the 182 peripheral output signals for its output.

See Section [5.9](#) for the list of peripheral signals via GPIO matrix.

3. RTC IO MUX is used to connect GPIO pads to their low-power and analog functions. Only a subset of GPIO pads have these optional RTC functions.

See Section [5.11](#) for the list of RTC IO MUX functions.

## 5.2 Peripheral Input via GPIO Matrix

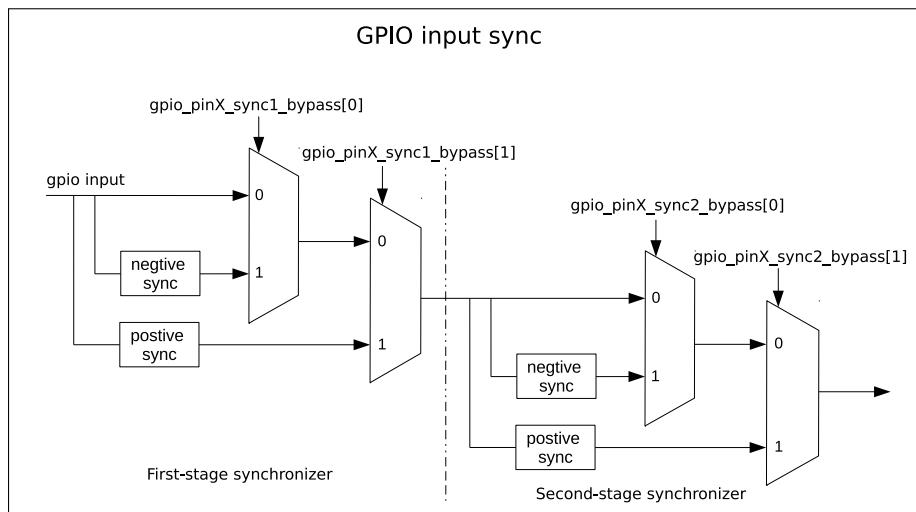
### 5.2.1 Overview

To receive a peripheral input signal via GPIO matrix, the matrix is configured to source the peripheral input signal from one of the 43 GPIOs (0 ~ 21, 26 ~ 46), see Table [22](#). Meanwhile, register corresponding to the peripheral should be set to receive input signal via GPIO matrix.

### 5.2.2 Synchronization

When signals are directed using the GPIO matrix, the signal will be synchronized to the APB bus clock by the GPIO SYNC hardware. This synchronization applies to all GPIO matrix signals but does not apply when using the IO MUX, see Figure [5-1](#).

Figure [5-2](#) shows the functionality of GPIO SYNC. In the figure, negative sync and positive sync mean GPIO input is synchronized on APB clock falling edge and on APB clock rising edge, respectively.



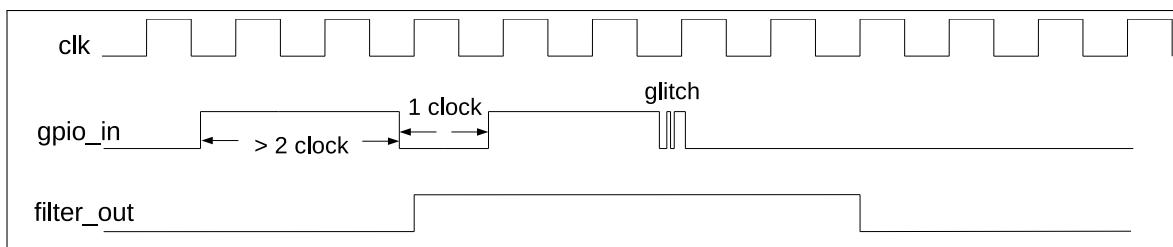
**Figure 5-2. GPIO Input Synchronized on Clock Rising Edge or on Falling Edge**

### 5.2.3 Functional Description

To read GPIO pad  $X$  into peripheral signal  $Y$ , follow the steps below:

1. Configure register `GPIO_FUNC $y$ _IN_SEL_CFG_REG` corresponding to peripheral signal  $Y$  in GPIO matrix:
  - Set `GPIO_SIG $y$ _IN_SEL` to enable peripheral signal input via GPIO matrix.
  - Set `GPIO_FUNC $y$ _IN_SEL` to the value corresponding to GPIO pad  $X$ .

**Note that** some peripheral signals have no valid `GPIO_SIG $y$ _IN_SEL` bit, namely, there is no MUX module in Figure 5-1 for these signals (see note 1 below Figure 5-1). These peripherals can only receive input signals via GPIO matrix.
2. Enable the filter for pad input signals by setting the register `IO_MUX_FILTER_EN`. Only the signals with a valid width of more than two clock cycles can be sampled, see Figure 5-3.



**Figure 5-3. Filter Timing Diagram of GPIO Input Signals**

3. Synchronize GPIO input. To do so, please set `GPIO_PIN $x$ _REG` corresponding to GPIO pad  $X$  as follows:
  - Set `GPIO_PIN $x$ _SYNC1_BYPASS` to enable input signal synchronized on rising edge or on falling edge in the first clock, see Figure 5-2.
  - Set `GPIO_PIN $x$ _SYNC2_BYPASS` to enable input signal synchronized on rising edge or on falling edge in the second clock, see Figure 5-2.
4. Configure IO MUX register to enable pad input. For this end, please set `IO_MUX_< $x$ >_REG` corresponding to GPIO pad  $X$  as follows:

- Set `IO_MUX_FUN_IE` to enable input.
- Set or clear `IO_MUX_FUN_WPU` and `IO_MUX_FUN_WPD`, as desired, to enable or disable pull-up and pull-down resistors.

For example, to connect RMT channel 0 input signal (`rmt_sig_in0`, signal index 83) to GPIO40, please follow the steps below. Note that GPIO40 is also named as MTDO pin.

1. Set `GPIO_SIG83_IN_SEL` in register `GPIO_FUNC83_IN_SEL_CFG_REG` to enable peripheral signal input via GPIO matrix.
2. Set `GPIO_FUNC83_IN_SEL` in register `GPIO_FUNC83_IN_SEL_CFG_REG` to 40.
3. Set `IO_MUX_FUN_IE` in register `IO_MUX_GPIO40_REG` to enable pad input.

**Note:**

- One input pad can be connected to multiple peripheral input signals.
- The input signal can be inverted by configuring `GPIO_FUNCy_IN_INV_SEL`.
- It is possible to have a peripheral read a constantly low or constantly high input value without connecting this input to a pad. This can be done by selecting a special `GPIO_FUNCy_IN_SEL` input, instead of a GPIO number:
  - When `GPIO_FUNCy_IN_SEL` is 0x3C, input signal `X` is always 0.
  - When `GPIO_FUNCy_IN_SEL` is 0x38, input signal `X` is always 1.

### 5.2.4 Simple GPIO Input

`GPIO_IN_REG`/`GPIO_IN1_REG` holds the input values of each GPIO pad. The input value of any GPIO pad can be read at any time without configuring GPIO matrix for a particular peripheral signal. However, it is necessary to enable the input in IO MUX by setting `IO_MUX_FUN_IE` bit in register `IO_MUX_n_REG` corresponding to pad `X`, as mentioned in Section 5.2.2.

## 5.3 Peripheral Output via GPIO Matrix

### 5.3.1 Overview

To output a signal from a peripheral via GPIO matrix, the matrix is configured to route peripheral output signals (0 ~ 11, 14 ~ 18, and etc.) to one of the 42 GPIOs (0 ~ 21, 26 ~ 45). See Table 22.

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

**Note:**

There is a range of peripheral output signals (223 ~ 227) which are not connected to any peripheral. These can be used to input a signal from one GPIO pad and output directly to another GPIO pad.

### 5.3.2 Functional Description

Some of the 182 output signals can be set to go through GPIO matrix into IO MUX and then to a pad. Figure 5-1 illustrates the configuration.

To output peripheral signal  $Y$  to a particular GPIO pad  $X$ , follow these steps:

1. Configure register `GPIO_FUNC $x$ _OUT_SEL_CFG_REG` and `GPIO_ENABLE_REG[ $x$ ]` corresponding to GPIO pad  $X$  in GPIO matrix. Recommended operation: use corresponding `W1TS` (write 1 to set) and `W1TC` (write 1 to clear) registers to set or clear `GPIO_ENABLE_REG`.
  - Set the `GPIO_FUNC $x$ _OUT_SEL` field in register `GPIO_FUNC $x$ _OUT_SEL_CFG_REG` to the index of the desired peripheral output signal  $Y$ .
  - If the signal should always be enabled as an output, set the `GPIO_FUNC $x$ _OEN_SEL` bit in register `GPIO_FUNC $x$ _OUT_SEL_CFG_REG` and the bit in register `GPIO_ENABLE_W1TS_REG` or in register `GPIO_ENABLE1_W1TS_REG`, corresponding to GPIO pad  $X$ . To have the output enable signal decided by internal logic (see the column "Output enable of output signals" in Table 22), clear `GPIO_FUNC $x$ _OEN_SEL` bit instead.
  - Clear the corresponding bit in register `GPIO_ENABLE_W1TC_REG` or in register `GPIO_ENABLE1_W1TC_REG` to disable the output from the GPIO pad.
2. For an open drain output, set the `GPIO_PIN $x$ _PAD_DRIVER` bit in register `GPIO_PIN $x$ _REG` corresponding to GPIO pad  $X$ .
3. Configure IO MUX register to enable output via GPIO matrix. Set the `IO_MUX_< $x$ >_REG` corresponding to GPIO pad  $X$  as follows:
  - Set the field `IO_MUX MCU SEL` to IO\_MUX function corresponding to GPIO pad  $X$ . This is Function 1, numeric value 1, for all pins.
  - Set the `IO_MUX FUN_DRV` field to the desired value for output strength (0 ~ 3). The higher the driver strength, the more current can be sourced/sunk from the pin.
    - 0: ~5 mA
    - 1: ~10 mA
    - 2: ~20 mA (Default value)
    - 3: ~40 mA
  - If using open drain mode, set/clear the `IO_MUX FUN_WPU` and `IO_MUX FUN_WPD` bits to enable/disable the internal pull-up/down resistors.

**Note:**

- The output signal from a single peripheral can be sent to multiple pads simultaneously.
- GPIO46 can not be used as an output.
- The output signal can be inverted by setting `GPIO_FUNC $n$ _OUT_INV_SEL` bit.

### 5.3.3 Simple GPIO Output

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

- Set GPIO matrix `GPIO_FUNCn_OUT_SEL` with a special peripheral index 256 (0x100);
- Set the corresponding bit in `GPIO_OUT_REG[31:0]` or `GPIO_OUT1_REG[21:0]` register to the desired GPIO output value.

**Note:**

- `GPIO_OUT_REG[0] ~ GPIO_OUT_REG[31]` correspond to GPIO0 ~ GPIO31, and `GPIO_OUT1_REG[25:22]` are invalid.
- `GPIO_OUT1_REG[0] ~ GPIO_OUT1_REG[13]` correspond to GPIO32 ~ GPIO45, and `GPIO_OUT1_REG[21:14]` are invalid.
- Recommended operation: use corresponding W1TS and W1TC registers, such as `GPIO_OUT_W1TS/GPIO_OUT_W1TC` to set or clear the registers `GPIO_OUT_REG/GPIO_OUT1_REG`.

### 5.3.4 Sigma Delta Modulated Output

#### 5.3.4.1 Functional Description

ESP32-S2 provides a second-order sigma delta modulation module and eight independent modulation channels. The channels are capable to output 1-bit signals (output index: 100 ~ 107) with sigma delta modulation, and by default output is enabled for these channels. This module can also output PDM (pulse density modulation) signal with configurable duty cycle. The transfer function is:

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

$E(z)$  is quantization error and  $X(z)$  is the input.

Sigma Delta modulator supports scaling down of APB\_CLK by divider 1 ~ 256:

- Set `GPIOSD_FUNCTION_CLK_EN` to enable the modulator clock.
- Configure register `GPIOSD_SDn_PRESCALE` ( $n$  is 0 ~ 7 for eight channels).

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

`GPIOSD_SDn_IN` is a signed number with a range of [-128, 127] and is used to control the duty cycle <sup>1</sup> of PDM output signal.

- `GPIOSD_SDn_IN` = -128, the duty cycle of the output signal is 0%.
- `GPIOSD_SDn_IN` = 0, the duty cycle of the output signal is near 50%.
- `GPIOSD_SDn_IN` = 127, the duty cycle of the output signal is close to 100%.

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

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

**Note:**

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

### 5.3.4.2 SDM Configuration

The configuration of SDM is shown below:

- Route one of SDM outputs to a pad via GPIO matrix, see Section [5.3.2](#).
- Enable the modulator clock by setting the register `GPIOSD_FUNCTION_CLK_EN`.
- Configure the divider value by setting the register `GPIOSD_SD $n$ _PRESCALE`.
- Configure the duty cycle of SDM output signal by setting the register `GPIOSD_SD $n$ _IN`.

## 5.4 Direct I/O via IO MUX

### 5.4.1 Overview

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

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

### 5.4.2 Functional Description

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

1. `IO_MUX MCU_SEL` for the GPIO pad must be set to the required pad function. For the list of pad functions, please refer to Section [5.10](#).
2. Set `GPIO_SIG $n$ _IN_SEL` to low level to route the input directly to the peripheral.

To bypass GPIO matrix for peripheral output signals, `IO_MUX MCU_SEL` for the GPIO pad must be set to the required pad function. For the list of pad functions, please refer to Section [5.10](#).

#### Note:

For peripheral I/O signals, not all signals can be connected to peripheral via IO MUX. Some specific input signals and some specific output signals can only be connected to peripheral via GPIO matrix.

## 5.5 RTC IO MUX for Low Power and Analog I/O

### 5.5.1 Overview

22 GPIO pads have low power capabilities (RTC domain) and analog functions which are handled by the RTC subsystem of ESP32-S2. IO MUX and GPIO matrix are not used for these functions, rather, RTC IO MUX is used to redirect input/output signals to the RTC subsystem.

When configured as RTC GPIOs, the output pads can still retain the output level value when the chip is in Deep-sleep mode, and the input pads can wake up the chip from Deep-sleep.

Section [5.11](#) lists the RTC\_MUX pins and their functions.

### 5.5.2 Functional Description

Each pad with analog and RTC functions is controlled by [RTCIO\\_TOUCH\\_PAD \$n\$ \\_MUX\\_SEL](#) bit in register [RTCIO\\_TOUCH\\_PAD \$n\$ \\_REG](#). By default all bits in these registers are set to 0, routing all input/output signals via IO MUX.

If [RTCIO\\_TOUCH\\_PAD \$n\$ \\_MUX\\_SEL](#) is set to 1, then input/output signals to and from that pad is routed to the RTC subsystem. In this mode, [RTCIO\\_TOUCH\\_PAD \$n\$ \\_REG](#) is used for digital input/output and the analog features of the pad are also available.

Please refer to Section 5.11 for the list of RTC pin functions and the mapping table of GPIO pads to their analog functions. Note that [RTCIO\\_TOUCH\\_PAD \$n\$ \\_REG](#) applies the RTC GPIO pin numbering, not the GPIO pad numbering.

## 5.6 Pin Functions in Light-sleep

Pins may provide different functions when ESP32-S2 is in Light-sleep mode. If [IO\\_MUX\\_SLP\\_SEL](#) in register [IO\\_MUX\\_n\\_REG](#) for a GPIO pad is set to 1, a different set of bits will be used to control the pad when the chip is in Light-sleep mode.

**Table 21: Pin Function Register for IO MUX Light-sleep Mode**

IO MUX Function	Normal Execution OR <a href="#">IO_MUX_SLP_SEL</a> = 0	Light-sleep Mode AND <a href="#">IO_MUX_SLP_SEL</a> = 1
Output Drive Strength	<a href="#">IO_MUX_FUN_DRV</a>	<a href="#">IO_MUX_FUN_DRV</a>
Pullup Resistor	<a href="#">IO_MUX_FUN_WPU</a>	<a href="#">IO_MUX_MCU_WPU</a>
Pulldown Resistor	<a href="#">IO_MUX_FUN_WPD</a>	<a href="#">IO_MUX_MCU_WPD</a>
Output Enable	(From GPIO Matrix _OEN field) <sup>1</sup>	<a href="#">IO_MUX_MCU_OE</a>

If [IO\\_MUX\\_SLP\\_SEL](#) is set to 0, pin functions remain the same in both normal execution and Light-sleep mode.

**Note:**

Please refer to Section 5.3.2 for how to enable output in normal execution (when [IO\\_MUX\\_SLP\\_SEL](#) = 0).

## 5.7 Pad Hold Feature

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

**Note:**

- For digital pads, to maintain pad input/output status in Deep-sleep mode, users can set RTC\_CNTL\_DG\_PAD\_FORCE\_UNHOLD to 0 before powering down. For RTC pads, the input and output values are controlled by the corresponding bits of register RTC\_PAD\_HOLD\_REG, and users can set it to 1 to hold the value or set it to 0 to unhold the value.
- For digital pads, to disable the hold function after the chip is woken up, users can set RTC\_CNTL\_DG\_PAD\_FORCE\_UNHOLD to 1. To maintain the hold function of the pad, users can change the corresponding bit in register RTC\_PAD\_HOLD\_REG to 1.

## 5.8 I/O Pad Power Supplies

For more information on the power supply for IO pads, please refer to Pin Definition in [ESP32-S2 Datasheet](#).

### 5.8.1 Power Supply Management

Each ESP32-S2 digital pin is connected to one of the four different power domains.

- VDD3P3\_RTC\_IO: the input power supply for both RTC and CPU
- VDD3P3\_CPU: the input power supply for CPU
- VDD3P3\_RTC: the input power supply for RTC analog part
- VDD\_SPI: configurable power supply

VDD\_SPI can be configured to use an internal LDO. The LDO input is VDD3P3\_RTC\_IO and the output is 1.8 V. If the LDO is not enabled, VDD\_SPI is connected directly to the same power supply as VDD3P3\_RTC\_IO.

The VDD\_SPI configuration is determined by the value of strapping pin GPIO45, or can be overridden by eFuse and/or register settings. See [ESP32-S2 Datasheet](#) sections Power Scheme and Strapping Pins for more details.

## 5.9 Peripheral Signal List

Table 22 shows the peripheral input/output signals via GPIO matrix.

**Table 22: GPIO Matrix**

Signal No.	Input signals	Default value if unassigned *	Same input signal from IO MUX core	Output signals	Output enable of output signals
0	SPIQ_in	0	yes	SPIQ_out	SPIQ_oe
1	SPID_in	0	yes	SPID_out	SPID_oe
2	SPIHD_in	0	yes	SPIHD_out	SPIHD_oe
3	SPIWP_in	0	yes	SPIWP_out	SPIWP_oe
4	-	-	-	SPICLK_out_mux	SPICLK_oe
5	-	-	-	SPICS0_out	SPICS0_oe
6	-	-	-	SPICS1_out	SPICS1_oe
7	SPID4_in	0	yes	SPID4_out	SPID4_oe

Signal No.	Input signals	Default value if unassigned *	Same input signal from IO MUX core	Output signals	Output enable of output signals
8	SPID5_in	0	yes	SPID5_out	SPID5_oe
9	SPID6_in	0	yes	SPID6_out	SPID6_oe
10	SPID7_in	0	yes	SPID7_out	SPID7_oe
11	SPIDQS_in	0	yes	SPIDQS_out	SPIDQS_oe
14	U0RXD_in	0	yes	U0TXD_out	1'd1
15	U0CTS_in	0	yes	U0RTS_out	1'd1
16	U0DSR_in	0	no	U0DTR_out	1'd1
17	U1RXD_in	0	yes	U1TXD_out	1'd1
18	U1CTS_in	0	yes	U1RTS_out	1'd1
21	U1DSR_in	0	no	U1DTR_out	1'd1
23	I2S0O_BCK_in	0	no	I2S0O_BCK_out	1'd1
25	I2S0O_WS_in	0	no	I2S0O_WS_out	1'd1
27	I2S0I_BCK_in	0	no	I2S0I_BCK_out	1'd1
28	I2S0I_WS_in	0	no	I2S0I_WS_out	1'd1
29	I2CEXT0_SCL_in	1	no	I2CEXT0_SCL_out	I2CEXT0_SCL_oe
30	I2CEXT0_SDA_in	1	no	I2CEXT0_SDA_out	I2CEXT0_SDA_oe
39	pcnt_sig_ch0_in0	0	no	gpio_wlan_prio	1'd1
40	pcnt_sig_ch1_in0	0	no	gpio_wlan_active	1'd1
41	pcnt_ctrl_ch0_in0	0	no	-	1'd1
42	pcnt_ctrl_ch1_in0	0	no	-	1'd1
43	pcnt_sig_ch0_in1	0	no	-	1'd1
44	pcnt_sig_ch1_in1	0	no	-	1'd1
45	pcnt_ctrl_ch0_in1	0	no	-	1'd1
46	pcnt_ctrl_ch1_in1	0	no	-	1'd1
47	pcnt_sig_ch0_in2	0	no	-	1'd1
48	pcnt_sig_ch1_in2	0	no	-	1'd1
49	pcnt_ctrl_ch0_in2	0	no	-	1'd1
50	pcnt_ctrl_ch1_in2	0	no	-	1'd1
51	pcnt_sig_ch0_in3	0	no	-	1'd1
52	pcnt_sig_ch1_in3	0	no	-	1'd1
53	pcnt_ctrl_ch0_in3	0	no	-	1'd1
54	pcnt_ctrl_ch1_in3	0	no	-	1'd1
64	usb_otg_iddig_in	0	no	-	1'd1
65	usb_otg_avalid_in	0	no	-	1'd1
66	usb_srp_bvalid_in	0	no	usb_otg_idpullup	1'd1
67	usb_otg_vbusvalid_in	0	no	usb_otg_dppulldown	1'd1
68	usb_srp_sessend_in	0	no	usb_otg_dmpulldown	1'd1
69	-	-	-	usb_otg_drvvbus	1'd1
70	-	-	-	usb_srp_chrvbus	1'd1
71	-	-	-	usb_srp_dischrvbus	1'd1
72	SPI3_CLK_in	0	no	SPI3_CLK_out_mux	SPI3_CLK_oe

Signal No.	Input signals	Default value if unassigned *	Same input signal from IO MUX core	Output signals	Output enable of output signals
73	SPI3_Q_in	0	no	SPI3_Q_out	SPI3_Q_oe
74	SPI3_D_in	0	no	SPI3_D_out	SPI3_D_oe
75	SPI3_HD_in	0	no	SPI3_HD_out	SPI3_HD_oe
76	SPI3_CS0_in	0	no	SPI3_CS0_out	SPI3_CS0_oe
77	-	-	-	SPI3_CS1_out	SPI3_CS1_oe
78	-	-	-	SPI3_CS2_out	SPI3_CS2_oe
79	-	-	-	ledc_ls_sig_out0	1'd1
80	-	-	-	ledc_ls_sig_out1	1'd1
81	-	-	-	ledc_ls_sig_out2	1'd1
82	-	-	-	ledc_ls_sig_out3	1'd1
83	rmt_sig_in0	0	no	ledc_ls_sig_out4	1'd1
84	rmt_sig_in1	0	no	ledc_ls_sig_out5	1'd1
85	rmt_sig_in2	0	no	ledc_ls_sig_out6	1'd1
86	rmt_sig_in3	0	no	ledc_ls_sig_out7	1'd1
87	-	-	-	rmt_sig_out0	1'd1
88	-	-	-	rmt_sig_out1	1'd1
89	-	-	-	rmt_sig_out2	1'd1
90	-	-	-	rmt_sig_out3	1'd1
95	I2CEXT1_SCL_in	1	no	I2CEXT1_SCL_out	I2CEXT1_SCL_oe
96	I2CEXT1_SDA_in	1	no	I2CEXT1_SDA_out	I2CEXT1_SDA_oe
100	-	-	-	gpio_sd0_out	1'd1
101	-	-	-	gpio_sd1_out	1'd1
102	-	-	-	gpio_sd2_out	1'd1
103	-	-	-	gpio_sd3_out	1'd1
104	-	-	-	gpio_sd4_out	1'd1
105	-	-	-	gpio_sd5_out	1'd1
106	-	-	-	gpio_sd6_out	1'd1
107	-	-	-	gpio_sd7_out	1'd1
108	FSPICLK_in	0	yes	FSPICLK_out_mux	FSPICLK_oe
109	FSPIQ_in	0	yes	FSPIQ_out	FSPIQ_oe
110	FSPID_in	0	yes	FSPID_out	FSPID_oe
111	FSPIHD_in	0	yes	FSPIHD_out	FSPIHD_oe
112	FSPIWP_in	0	yes	FSPIWP_out	FSPIWP_oe
113	FSPIIO4_in	0	yes	FSPIIO4_out	FSPIIO4_oe
114	FSPIIO5_in	0	yes	FSPIIO5_out	FSPIIO5_oe
115	FSPIIO6_in	0	yes	FSPIIO6_out	FSPIIO6_oe
116	FSPIIO7_in	0	yes	FSPIIO7_out	FSPIIO7_oe
117	FSPICS0_in	0	yes	FSPICS0_out	FSPICS0_oe
118	-	-	-	FSPICS1_out	FSPICS1_oe
119	-	-	-	FSPICS2_out	FSPICS2_oe
120	-	-	-	FSPICS3_out	FSPICS3_oe

Signal No.	Input signals	Default value if unassigned *	Same input signal from IO MUX core	Output signals	Output enable of output signals
121	-	-	-	FSPICS4_out	FSPICS4_oe
122	-	-	-	FSPICS5_out	FSPICS5_oe
123	can_rx	1	no	can_tx	1'd1
124	-	-	-	can_bus_off_on	1'd1
125	-	-	-	can_clkout	1'd1
126	-	-	-	SUBSPICLK_out_mux	SUBSPICLK_oe
127	SUBSPIQ_in	0	yes	SUBSPIQ_out	SUBSPIQ_oe
128	SUBSPID_in	0	yes	SUBSPID_out	SUBSPID_oe
129	SUBSPIHD_in	0	yes	SUBSPIHD_out	SUBSPIHD_oe
130	SUBSPIWP_in	0	yes	SUBSPIWP_out	SUBSPIWP_oe
131	-	-	-	SUBSPICS0_out	SUBSPICS0_oe
132	-	-	-	SUBSPICS1_out	SUBSPICS1_oe
133	-	-	-	FSPIDQS_out	FSPIDQS_oe
134	-	-	-	FSPI_HSYNC_out	FSPI_HSYNC_oe
135	-	-	-	FSPI_VSYNC_out	FSPI_VSYNC_oe
136	-	-	-	FSPI_DE_out	FSPI_DE_oe
137	-	-	-	FSPICD_out	FSPICD_oe
139	-	-	-	SPI3_CD_out	SPI3_CD_oe
140	-	-	-	SPI3_DQS_out	SPI3_DQS_oe
143	I2S0I_DATA_in0	0	no	I2S0O_DATA_out0	1'd1
144	I2S0I_DATA_in1	0	no	I2S0O_DATA_out1	1'd1
145	I2S0I_DATA_in2	0	no	I2S0O_DATA_out2	1'd1
146	I2S0I_DATA_in3	0	no	I2S0O_DATA_out3	1'd1
147	I2S0I_DATA_in4	0	no	I2S0O_DATA_out4	1'd1
148	I2S0I_DATA_in5	0	no	I2S0O_DATA_out5	1'd1
149	I2S0I_DATA_in6	0	no	I2S0O_DATA_out6	1'd1
150	I2S0I_DATA_in7	0	no	I2S0O_DATA_out7	1'd1
151	I2S0I_DATA_in8	0	no	I2S0O_DATA_out8	1'd1
152	I2S0I_DATA_in9	0	no	I2S0O_DATA_out9	1'd1
153	I2S0I_DATA_in10	0	no	I2S0O_DATA_out10	1'd1
154	I2S0I_DATA_in11	0	no	I2S0O_DATA_out11	1'd1
155	I2S0I_DATA_in12	0	no	I2S0O_DATA_out12	1'd1
156	I2S0I_DATA_in13	0	no	I2S0O_DATA_out13	1'd1
157	I2S0I_DATA_in14	0	no	I2S0O_DATA_out14	1'd1
158	I2S0I_DATA_in15	0	no	I2S0O_DATA_out15	1'd1
159	-	-	-	I2S0O_DATA_out16	1'd1
160	-	-	-	I2S0O_DATA_out17	1'd1
161	-	-	-	I2S0O_DATA_out18	1'd1
162	-	-	-	I2S0O_DATA_out19	1'd1
163	-	-	-	I2S0O_DATA_out20	1'd1
164	-	-	-	I2S0O_DATA_out21	1'd1

Signal No.	Input signals	Default value if unassigned *	Same input signal from IO MUX core	Output signals	Output enable of output signals
165	-	-	-	I2S0O_DATA_out22	1'd1
166	-	-	-	I2S0O_DATA_out23	1'd1
167	SUBSPID4_in	0	yes	SUBSPID4_out	SUBSPID4_oe
168	SUBSPID5_in	0	yes	SUBSPID5_out	SUBSPID5_oe
169	SUBSPID6_in	0	yes	SUBSPID6_out	SUBSPID6_oe
170	SUBSPID7_in	0	yes	SUBSPID7_out	SUBSPID7_oe
171	SUBSPIDQS_in	0	yes	SUBSPIDQS_out	SUBSPIDQS_oe
193	I2S0I_H_SYNC	0	no	-	1'd1
194	I2S0I_V_SYNC	0	no	-	1'd1
195	I2S0I_H_ENABLE	0	no	-	1'd1
215	-	-	-	ant_sel0	1'd1
216	-	-	-	ant_sel1	1'd1
217	-	-	-	ant_sel2	1'd1
218	-	-	-	ant_sel3	1'd1
219	-	-	-	ant_sel4	1'd1
220	-	-	-	ant_sel5	1'd1
221	-	-	-	ant_sel6	1'd1
222	-	-	-	ant_sel7	1'd1
223	sig_in_func_223	0	no	sig_in_func223	1'd1
224	sig_in_func_224	0	no	sig_in_func224	1'd1
225	sig_in_func_225	0	no	sig_in_func225	1'd1
226	sig_in_func_226	0	no	sig_in_func226	1'd1
227	sig_in_func_227	0	no	sig_in_func227	1'd1
235	pro_alonegpio_in0	0	no	pro_alonegpio_out0	1'd1
236	pro_alonegpio_in1	0	no	pro_alonegpio_out1	1'd1
237	pro_alonegpio_in2	0	no	pro_alonegpio_out2	1'd1
238	pro_alonegpio_in3	0	no	pro_alonegpio_out3	1'd1
239	pro_alonegpio_in4	0	no	pro_alonegpio_out4	1'd1
240	pro_alonegpio_in5	0	no	pro_alonegpio_out5	1'd1
241	pro_alonegpio_in6	0	no	pro_alonegpio_out6	1'd1
242	pro_alonegpio_in7	0	no	pro_alonegpio_out7	1'd1
251	-	-	-	clk_i2s_mux	1'd1

## 5.10 IO MUX Pad List

Table 23 shows the IO MUX functions of each I/O pad:

Table 23: IO MUX Pad List

GPIO	Pad Name	Function 0	Function 1	Function 2	Function 3	Function 4	Reset	Notes
0	GPIO0	GPIO0	GPIO0	-	-	-	3	R
1	GPIO1	GPIO1	GPIO1	-	-	-	1	R

GPIO	Pad Name	Function 0	Function 1	Function 2	Function 3	Function 4	Reset	Notes
2	GPIO2	GPIO2	GPIO2	-	-	-	1	R
3	GPIO3	GPIO3	GPIO3	-	-	-	0	R
4	GPIO4	GPIO4	GPIO4	-	-	-	0	R
5	GPIO5	GPIO5	GPIO5	-	-	-	0	R
6	GPIO6	GPIO6	GPIO6	-	-	-	0	R
7	GPIO7	GPIO7	GPIO7	-	-	-	0	R
8	GPIO8	GPIO8	GPIO8	-	SUBSPICS1	-	0	R
9	GPIO9	GPIO9	GPIO9	-	SUBSPIHD	FSPIHD	1	R
10	GPIO10	GPIO10	GPIO10	FSPII04	SUBSPICS0	FSPICS0	1	R
11	GPIO11	GPIO11	GPIO11	FSPII05	SUBSPID	FSPID	1	R
12	GPIO12	GPIO12	GPIO12	FSPII06	SUBSPICLK	FSPICLK	1	R
13	GPIO13	GPIO13	GPIO13	FSPII07	SUBSPIQ	FSPIQ	1	R
14	GPIO14	GPIO14	GPIO14	FSPIDQS	SUBSPIWP	FSPIWP	1	R
15	XTAL_32K_P	XTAL_32K_P	GPIO15	U0RTS	-	-	0	R
16	XTAL_32K_N	XTAL_32K_N	GPIO16	U0CTS	-	-	0	R
17	DAC_1	DAC_1	GPIO17	U1TXD	-	-	1	R
18	DAC_2	DAC_2	GPIO18	U1RXD	CLK_OUT3	-	1	R
19	GPIO19	GPIO19	GPIO19	U1RTS	CLK_OUT2	-	0	R
20	GPIO20	GPIO20	GPIO20	U1CTS	CLK_OUT1	-	0	R
21	GPIO21	GPIO21	GPIO21	-	-	-	0	R
26	SPICS1	SPICS1	GPIO26	-	-	-	3	-
27	SPIHD	SPIHD	GPIO27	-	-	-	3	-
28	SPIWP	SPIWP	GPIO28	-	-	-	3	-
29	SPICS0	SPICS0	GPIO29	-	-	-	3	-
30	SPICLK	SPICLK	GPIO30	-	-	-	3	-
31	SPIQ	SPIQ	GPIO31	-	-	-	3	-
32	SPIID	SPIID	GPIO32	-		-	3	-
33	GPIO33	GPIO33	GPIO33	FSPIHD	SUBSPIHD	SPII04	1	-
34	GPIO34	GPIO34	GPIO34	FSPICS0	SUBSPICS0	SPII05	1	-
35	GPIO35	GPIO35	GPIO35	FSPID	SUBSPID	SPII06	1	-
36	GPIO36	GPIO36	GPIO36	FSPICLK	SUBSPICLK	SPII07	1	-
37	GPIO37	GPIO37	GPIO37	FSPIQ	SUBSPIQ	SPIDQS	1	-
38	GPIO38	GPIO38	GPIO38	FSPIWP	SUBSPIWP	-	1	-
39	MTCK	MTCK	GPIO39	CLK_OUT3	SUBSPICS1	-	1	-
40	MTDO	MTDO	GPIO40	CLK_OUT2	-	-	1	-
41	MTDI	MTDI	GPIO41	CLK_OUT1	-	-	1	-
42	MTMS	MTMS	GPIO42	-	-	-	1	-
43	U0TXD	U0TXD	GPIO43	CLK_OUT1	-	-	3	-
44	U0RXD	U0RXD	GPIO44	CLK_OUT2	-	-	3	-
45	GPIO45	GPIO45	GPIO45	-	-	-	2	-
46	GPIO46	GPIO46	GPIO46	-	-	-	2	I

### Reset Configurations

“Reset” column shows the default configuration of each pad after reset:

- **0** - IE=0 (input disabled)
- **1** - IE=1 (input enabled)
- **2** - IE=1, WPD=1 (input enabled, pull-down resistor enabled)
- **3** - IE=1, WPU=1 (input enabled, pull-up resistor enabled)

**Note:**

- **R** - Pad has RTC/analog functions via RTC IO MUX.
- **I** - Pad can only be configured as input GPIO.

Please refer to Appendix A – ESP32-S2 Pin Lists in [ESP32-S2 Datasheet](#) for more details.

## 5.11 RTC IO MUX Pin List

Table 24 shows the RTC pins and how they correspond to GPIO pads.

**Table 24: RTC IO MUX Pin Summary**

RTC GPIO Num	GPIO Num	Pad Name	Analog Function			
			1	2	3	4
0	0	TOUCH_PAD0*	RTC_GPIO0	-	-	sar_i2c_scl_0
1	1	TOUCH_PAD1	RTC_GPIO1	-	-	sar_i2c_sda_0
2	2	TOUCH_PAD2	RTC_GPIO2	-	-	sar_i2c_scl_1
3	3	TOUCH_PAD3	RTC_GPIO3	-	-	sar_i2c_sda_1
4	4	TOUCH_PAD4	RTC_GPIO4	-	-	-
5	5	TOUCH_PAD5	RTC_GPIO5	-	-	-
6	6	TOUCH_PAD6	RTC_GPIO6	-	-	-
7	7	TOUCH_PAD7	RTC_GPIO7	-	-	-
8	8	TOUCH_PAD8	RTC_GPIO8	-	-	-
9	9	TOUCH_PAD9	RTC_GPIO9	-	-	-
10	10	TOUCH_PAD10	RTC_GPIO10	-	-	-
11	11	TOUCH_PAD11	RTC_GPIO11	-	-	-
12	12	TOUCH_PAD12	RTC_GPIO12	-	-	-
13	13	TOUCH_PAD13	RTC_GPIO13	-	-	-
14	14	TOUCH_PAD14	RTC_GPIO14	-	-	-
15	15	X32P	RTC_GPIO15	-	-	-
16	16	X32N	RTC_GPIO16	-	-	-
17	17	PDAC1	RTC_GPIO17	-	-	-
18	18	PDAC2	RTC_GPIO18	-	-	-
19	19	RTC_PAD19	RTC_GPIO19	-	-	-
20	20	RTC_PAD20	RTC_GPIO20	-	-	-
21	21	RTC_PAD21	RTC_GPIO21	-	-	-

**Note:** TOUCH\_PAD0 is an internal channel and its analog functions are not lead to a corresponding external

GPIO.

## 5.12 Base Address

Users can access GPIO and RTCIO respectively with two register base addresses shown in the following table.

For more information about accessing peripherals from different buses, please see Chapter 1 *System and Memory*.

**Table 25: GPIO and RTCIO Base Address**

Module Name	Access to Access Peripheral	Base Address
GPIO	PeriBUS1	0x3F404000
	PeriBUS2	0x60004000
IO MUX	PeriBUS2	0x60004000
GPIOSD	PeriBUS2	0x60004F00
RTCIO	PeriBUS1	0x3F408400
	PeriBUS2	0x60008400

## 5.13 Register Summary

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 base address, please refer to Section 5.12.

### 5.13.1 GPIO Matrix Register Summary

Name	Description	Address	Access
<b>GPIO Configuration Registers</b>			
GPIO_BT_SELECT_REG	GPIO bit selection register	0x0000	R/W
GPIO_OUT_REG	GPIO0 ~ 31 output register	0x0004	R/W
GPIO_OUT_W1TS_REG	GPIO0 ~ 31 output bit set register	0x0008	WO
GPIO_OUT_W1TC_REG	GPIO0 ~ 31 output bit clear register	0x000C	WO
GPIO_OUT1_REG	GPIO32 ~ 53 output register	0x0010	R/W
GPIO_OUT1_W1TS_REG	GPIO32 ~ 53 output bit set register	0x0014	WO
GPIO_OUT1_W1TC_REG	GPIO32 ~ 53 output bit clear register	0x0018	WO
GPIO_SDIO_SELECT_REG	GPIO SDIO selection register	0x001C	R/W
GPIO_ENABLE_REG	GPIO0 ~ 31 output enable register	0x0020	R/W
GPIO_ENABLE_W1TS_REG	GPIO0 ~ 31 output enable bit set register	0x0024	WO
GPIO_ENABLE_W1TC_REG	GPIO0 ~ 31 output enable bit clear register	0x0028	WO
GPIO_ENABLE1_REG	GPIO32 ~ 53 output enable register	0x002C	R/W
GPIO_ENABLE1_W1TS_REG	GPIO32 ~ 53 output enable bit set register	0x0030	WO
GPIO_ENABLE1_W1TC_REG	GPIO32 ~ 53 output enable bit clear register	0x0034	WO
GPIO_STRAP_REG	Bootstrap pin value register	0x0038	RO
GPIO_IN_REG	GPIO0 ~ 31 input register	0x003C	RO
GPIO_IN1_REG	GPIO32 ~ 53 input register	0x0040	RO

Name	Description	Address	Access
GPIO_PIN0_REG	Configuration for GPIO pin 0	0x0074	R/W
GPIO_PIN1_REG	Configuration for GPIO pin 1	0x0078	R/W
GPIO_PIN2_REG	Configuration for GPIO pin 2	0x007C	R/W
...	...	...	...
GPIO_PIN51_REG	Configuration for GPIO pin 51	0x0140	R/W
GPIO_PIN52_REG	Configuration for GPIO pin 52	0x0144	R/W
GPIO_PIN53_REG	Configuration for GPIO pin 53	0x0148	R/W
GPIO_FUNC0_IN_SEL_CFG_REG	Peripheral function 0 input selection register	0x0154	R/W
GPIO_FUNC1_IN_SEL_CFG_REG	Peripheral function 1 input selection register	0x0158	R/W
GPIO_FUNC2_IN_SEL_CFG_REG	Peripheral function 2 input selection register	0x015C	R/W
...	...	...	...
GPIO_FUNC253_IN_SEL_CFG_REG	Peripheral function 253 input selection register	0x0548	R/W
GPIO_FUNC254_IN_SEL_CFG_REG	Peripheral function 254 input selection register	0x054C	R/W
GPIO_FUNC255_IN_SEL_CFG_REG	Peripheral function 255 input selection register	0x0550	R/W
GPIO_FUNC0_OUT_SEL_CFG_REG	Peripheral output selection for GPIO0	0x0554	R/W
GPIO_FUNC1_OUT_SEL_CFG_REG	Peripheral output selection for GPIO1	0x0558	R/W
GPIO_FUNC2_OUT_SEL_CFG_REG	Peripheral output selection for GPIO2	0x055C	R/W
..	...	...	...
GPIO_FUNC51_OUT_SEL_CFG_REG	Peripheral output selection for GPIO51	0x0620	R/W
GPIO_FUNC52_OUT_SEL_CFG_REG	Peripheral output selection for GPIO52	0x0624	R/W
GPIO_FUNC53_OUT_SEL_CFG_REG	Peripheral output selection for GPIO53	0x0628	R/W
GPIO_CLOCK_GATE_REG	GPIO clock gating register	0x062C	R/W
<b>Interrupt Configuration Registers</b>			
GPIO_STATUS_W1TS_REG	GPIO0 ~ 31 interrupt status bit set register	0x0048	WO
GPIO_STATUS_W1TC_REG	GPIO0 ~ 31 interrupt status bit clear register	0x004C	WO
GPIO_STATUS1_W1TS_REG	GPIO32 ~ 53 interrupt status bit set register	0x0054	WO
GPIO_STATUS1_W1TC_REG	GPIO32 ~ 53 interrupt status bit clear register	0x0058	WO
<b>GPIO Interrupt Source Registers</b>			
GPIO_STATUS_NEXT_REG	GPIO0 ~ 31 interrupt source register	0x014C	RO
GPIO_STATUS_NEXT1_REG	GPIO32 ~ 53 interrupt source register	0x0150	RO
<b>Interrupt Status Registers</b>			
GPIO_STATUS_REG	GPIO0 ~ 31 interrupt status register	0x0044	R/W
GPIO_STATUS1_REG	GPIO32 ~ 53 interrupt status register	0x0050	R/W
GPIO_PCPU_INT_REG	GPIO0 ~ 31 PRO_CPU interrupt status register	0x005C	RO
GPIO_PCPU_NMI_INT_REG	GPIO0 ~ 31 PRO_CPU non-maskable interrupt status register	0x0060	RO
GPIO_PCPU_INT1_REG	GPIO32 ~ 53 PRO_CPU interrupt status register	0x0068	RO
GPIO_PCPU_NMI_INT1_REG	GPIO32 ~ 53 PRO_CPU non-maskable interrupt status register	0x006C	RO

### 5.13.2 IO MUX Register Summary

Name	Description	Address	Access
IO_MUX_PIN_CTRL	Clock output configuration register	0x0000	R/W
IO_MUX_GPIO0_REG	Configuration register for pad GPIO0	0x0004	R/W
IO_MUX_GPIO1_REG	Configuration register for pad GPIO1	0x0008	R/W
IO_MUX_GPIO2_REG	Configuration register for pad GPIO2	0x000C	R/W
IO_MUX_GPIO3_REG	Configuration register for pad GPIO3	0x0010	R/W
IO_MUX_GPIO4_REG	Configuration register for pad GPIO4	0x0014	R/W
IO_MUX_GPIO5_REG	Configuration register for pad GPIO5	0x0018	R/W
IO_MUX_GPIO6_REG	Configuration register for pad GPIO6	0x001C	R/W
IO_MUX_GPIO7_REG	Configuration register for pad GPIO7	0x0020	R/W
IO_MUX_GPIO8_REG	Configuration register for pad GPIO8	0x0024	R/W
IO_MUX_GPIO9_REG	Configuration register for pad GPIO9	0x0028	R/W
IO_MUX_GPIO10_REG	Configuration register for pad GPIO10	0x002C	R/W
IO_MUX_GPIO11_REG	Configuration register for pad GPIO11	0x0030	R/W
IO_MUX_GPIO12_REG	Configuration register for pad GPIO12	0x0034	R/W
IO_MUX_GPIO13_REG	Configuration register for pad GPIO13	0x0038	R/W
IO_MUX_GPIO14_REG	Configuration register for pad GPIO14	0x003C	R/W
IO_MUX_XTAL_32K_P_REG	Configuration register for pad XTAL_32K_P	0x0040	R/W
IO_MUX_XTAL_32K_N_REG	Configuration register for pad XTAL_32K_N	0x0044	R/W
IO_MUX_DAC_1_REG	Configuration register for pad DAC_1	0x0048	R/W
IO_MUX_DAC_2_REG	Configuration register for pad DAC_2	0x004C	R/W
IO_MUX_GPIO_19_REG	Configuration register for pad GPIO19	0x0050	R/W
IO_MUX_GPIO_20_REG	Configuration register for pad GPIO20	0x0054	R/W
IO_MUX_GPIO_21_REG	Configuration register for pad GPIO21	0x0058	R/W
IO_MUX_SPICS1_REG	Configuration register for pad SPICS1	0x006C	R/W
IO_MUX_SPIHD_REG	Configuration register for pad SPIHD	0x0070	R/W
IO_MUX_SPIWP_REG	Configuration register for pad SPIWP	0x0074	R/W
IO_MUX_SPICS0_REG	Configuration register for pad SPICS0	0x0078	R/W
IO_MUX_SPICLK_REG	Configuration register for pad SPICLK	0x007C	R/W
IO_MUX_SPIQ_REG	Configuration register for pad SPIQ	0x0080	R/W
IO_MUX_SPID_REG	Configuration register for pad SPID	0x0084	R/W
IO_MUX_GPIO_33_REG	Configuration register for pad GPIO33	0x0088	R/W
IO_MUX_GPIO_34_REG	Configuration register for pad GPIO34	0x008C	R/W
IO_MUX_GPIO_35_REG	Configuration register for pad GPIO35	0x0090	R/W
IO_MUX_GPIO_36_REG	Configuration register for pad GPIO36	0x0094	R/W
IO_MUX_GPIO_37_REG	Configuration register for pad GPIO37	0x0098	R/W
IO_MUX_GPIO_38_REG	Configuration register for pad GPIO38	0x009C	R/W
IO_MUX_MTCK_REG	Configuration register for pad MTCK	0x00A0	R/W
IO_MUX_MTDO_REG	Configuration register for pad MTDO	0x00A4	R/W
IO_MUX_MTDI_REG	Configuration register for pad MTDI	0x00A8	R/W
IO_MUX_MTMS_REG	Configuration register for pad MTMS	0x00AC	R/W
IO_MUX_U0TXD_REG	Configuration register for pad U0TXD	0x00B0	R/W
IO_MUX_U0RXD_REG	Configuration register for pad U0RXD	0x00B4	R/W
IO_MUX_GPIO_45_REG	Configuration register for pad GPIO45	0x00B8	R/W

Name	Description	Address	Access
IO_MUX_GPIO_46_REG	Configuration register for pad GPIO46	0x00BC	R/W

### 5.13.3 Sigma Delta Modulated Output Register Summary

Name	Description	Address	Access
<b>Configuration Registers</b>			
GPIOSD_SIGMADELTA0_REG	Duty Cycle Configure Register of SDM0	0x0000	R/W
GPIOSD_SIGMADELTA1_REG	Duty Cycle Configure Register of SDM1	0x0004	R/W
GPIOSD_SIGMADELTA2_REG	Duty Cycle Configure Register of SDM2	0x0008	R/W
GPIOSD_SIGMADELTA3_REG	Duty Cycle Configure Register of SDM3	0x000C	R/W
GPIOSD_SIGMADELTA4_REG	Duty Cycle Configure Register of SDM4	0x0010	R/W
GPIOSD_SIGMADELTA5_REG	Duty Cycle Configure Register of SDM5	0x0014	R/W
GPIOSD_SIGMADELTA6_REG	Duty Cycle Configure Register of SDM6	0x0018	R/W
GPIOSD_SIGMADELTA7_REG	Duty Cycle Configure Register of SDM7	0x001C	R/W
GPIOSD_SIGMADELTA(CG)_REG	Clock Gating Configure Register	0x0020	R/W
GPIOSD_SIGMADELTA_MISC_REG	MISC Register	0x0024	R/W
GPIOSD_SIGMADELTA_VERSION_REG	Version Control Register	0x0028	R/W

### 5.13.4 RTC IO MUX Register Summary

Name	Description	Address	Access
<b>GPIO Configuration and Data Registers</b>			
RTCIO_RTC_GPIO_OUT_REG	RTC GPIO output register	0x0000	R/W
RTCIO_RTC_GPIO_OUT_W1TS_REG	RTC GPIO output bit set register	0x0004	WO
RTCIO_RTC_GPIO_OUT_W1TC_REG	RTC GPIO output bit clear register	0x0008	WO
RTCIO_RTC_GPIO_ENABLE_REG	RTC GPIO output enable register	0x000C	R/W
RTCIO_RTC_GPIO_ENABLE_W1TS_REG	RTC GPIO output enable bit set register	0x0010	WO
RTCIO_RTC_GPIO_ENABLE_W1TC_REG	RTC GPIO output enable bit clear register	0x0014	WO
RTCIO_RTC_GPIO_STATUS_REG	RTC GPIO interrupt status register	0x0018	R/W
RTCIO_RTC_GPIO_STATUS_W1TS_REG	RTC GPIO interrupt status bit set register	0x001C	WO
RTCIO_RTC_GPIO_STATUS_W1TC_REG	RTC GPIO interrupt status bit clear register	0x0020	WO
RTCIO_RTC_GPIO_IN_REG	RTC GPIO input register	0x0024	RO
RTCIO_RTC_GPIO_PIN0_REG	RTC configuration for pin 0	0x0028	R/W
RTCIO_RTC_GPIO_PIN1_REG	RTC configuration for pin 1	0x002C	R/W
RTCIO_RTC_GPIO_PIN2_REG	RTC configuration for pin 2	0x0030	R/W
RTCIO_RTC_GPIO_PIN3_REG	RTC configuration for pin 3	0x0034	R/W
...	...	...	...
RTCIO_RTC_GPIO_PIN19_REG	RTC configuration for pin 19	0x0074	R/W
RTCIO_RTC_GPIO_PIN20_REG	RTC configuration for pin 20	0x0078	R/W
RTCIO_RTC_GPIO_PIN21_REG	RTC configuration for pin 21	0x007C	R/W
<b>GPIO RTC Function Configuration Registers</b>			
RTCIO_TOUCH_PAD0_REG	Touch pad 0 configuration register	0x0084	R/W
RTCIO_TOUCH_PAD1_REG	Touch pad 1 configuration register	0x0088	R/W

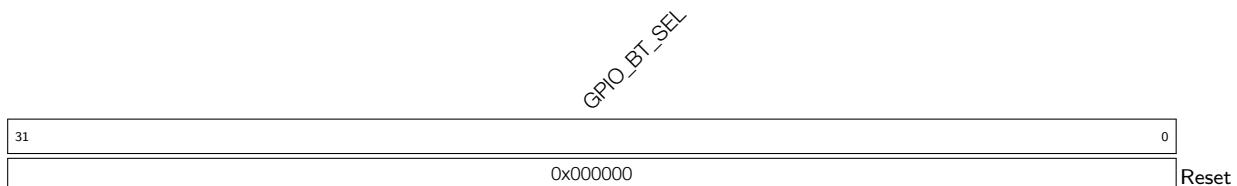
Name	Description	Address	Access
RTCIO_TOUCH_PAD2_REG	Touch pad 2 configuration register	0x008C	R/W
...	...	...	...
RTCIO_TOUCH_PAD13_REG	Touch pad 13 configuration register	0x00B8	R/W
RTCIO_TOUCH_PAD14_REG	Touch pad 14 configuration register	0x00BC	R/W
RTCIO_XTAL_32P_PAD_REG	32KHz crystal P-pad configuration register	0x00C0	R/W
RTCIO_XTAL_32N_PAD_REG	32KHz crystal N-pad configuration register	0x00C4	R/W
RTCIO_PAD_DAC1_REG	DAC1 configuration register	0x00C8	R/W
RTCIO_PAD_DAC2_REG	DAC2 configuration register	0x00CC	R/W
RTCIO_RTC_PAD19_REG	Touch pad 19 configuration register	0x00D0	R/W
RTCIO_RTC_PAD20_REG	Touch pad 20 configuration register	0x00D4	R/W
RTCIO_RTC_PAD21_REG	Touch pad 21 configuration register	0x00D8	R/W
RTCIO_XTL_EXT_CTR_REG	Crystal power down enable GPIO source	0x00E0	R/W
RTCIO_SAR_I2C_IO_REG	RTC I2C pad selection	0x00E4	R/W

## 5.14 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 base address, please refer to Section 5.12.

### 5.14.1 GPIO Matrix Registers

Register 5.1: GPIO\_BT\_SELECT\_REG (0x0000)



**GPIO\_BT\_SEL** Reserved (R/W)

**Register 5.2: GPIO\_OUT\_REG (0x0004)**

GPIO_OUT_DATA_ORIG	
31	0
0x000000	Reset

**GPIO\_OUT\_DATA\_ORIG** GPIO0 ~ 31 output value in simple GPIO output mode. The values of bit0 ~ bit31 correspond to the output value of GPIO0 ~ GPIO31 respectively. Bit22 ~ bit25 are invalid. (R/W)

**Register 5.3: GPIO\_OUT\_W1TS\_REG (0x0008)**

GPIO_OUT_W1TS	
31	0
0x000000	Reset

**GPIO\_OUT\_W1TS** GPIO0 ~ 31 output set register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_OUT\\_REG](#) will be set to 1. Recommended operation: use this register to set [GPIO\\_OUT\\_REG](#). (WO)

**Register 5.4: GPIO\_OUT\_W1TC\_REG (0x000C)**

GPIO_OUT_W1TC	
31	0
0x000000	Reset

**GPIO\_OUT\_W1TC** GPIO0 ~ 31 output clear register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_OUT\\_REG](#) will be cleared. Recommended operation: use this register to clear [GPIO\\_OUT\\_REG](#). (WO)

**Register 5.5: GPIO\_OUT1\_REG (0x0010)**

The diagram shows the memory map for Register 5.5. It consists of a single row of four columns. The first three columns are labeled "31", "22", and "21" respectively, representing bits 31 to 21. The fourth column is labeled "0" and contains the value "0x0000". A label "GPIO\_OUT1\_DATA\_ORIG" is positioned above the "0" column, rotated diagonally upwards. A label "(reserved)" is positioned above the first three columns, rotated diagonally downwards. A "Reset" button is located at the bottom right of the fourth column.

31	22	21	0
0	0	0	0x0000

Reset

**GPIO\_OUT1\_DATA\_ORIG** GPIO32 ~ 53 output value in simple GPIO output mode. The values of bit0 ~ bit13 correspond to GPIO32 ~ GPIO45. Bit14 ~ bit21 are invalid. (R/W)

**Register 5.6: GPIO\_OUT1\_W1TS\_REG (0x0014)**

The diagram shows the memory map for Register 5.6. It consists of a single row of four columns. The first three columns are labeled "31", "22", and "21" respectively, representing bits 31 to 21. The fourth column is labeled "0" and contains the value "0x0000". A label "GPIO\_OUT1\_W1TS" is positioned above the "0" column, rotated diagonally upwards. A label "(reserved)" is positioned above the first three columns, rotated diagonally downwards. A "Reset" button is located at the bottom right of the fourth column.

31	22	21	0
0	0	0	0x0000

Reset

**GPIO\_OUT1\_W1TS** GPIO32 ~ 53 output value set register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_OUT1\\_REG](#) will be set to 1. Recommended operation: use this register to set [GPIO\\_OUT1\\_REG](#). (WO)

**Register 5.7: GPIO\_OUT1\_W1TC\_REG (0x0018)**

The diagram shows the memory map for Register 5.7. It consists of a single row of four columns. The first three columns are labeled "31", "22", and "21" respectively, representing bits 31 to 21. The fourth column is labeled "0" and contains the value "0x0000". A label "GPIO\_OUT1\_W1TC" is positioned above the "0" column, rotated diagonally upwards. A label "(reserved)" is positioned above the first three columns, rotated diagonally downwards. A "Reset" button is located at the bottom right of the fourth column.

31	22	21	0
0	0	0	0x0000

Reset

**GPIO\_OUT1\_W1TC** GPIO32 ~ 53 output value clear register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_OUT1\\_REG](#) will be cleared. Recommended operation: use this register to clear [GPIO\\_OUT1\\_REG](#). (WO)

**Register 5.8: GPIO\_SDIO\_SELECT\_REG (0x001C)**

(reserved)																GPIO_SDIO_SEL	
31 0																8   7	0
0x0																Reset	

**GPIO\_SDIO\_SEL** Reserved (R/W)

**Register 5.9: GPIO\_ENABLE\_REG (0x0020)**

(reserved)																GPIO_ENABLE_DATA	
31 0x000000																0	Reset
0x000000																Reset	

**GPIO\_ENABLE\_DATA** GPIO0 ~ 31 output enable register. (R/W)

**Register 5.10: GPIO\_ENABLE\_W1TS\_REG (0x0024)**

(reserved)																GPIO_ENABLE_W1TS	
31 0x000000																0	Reset
0x000000																Reset	

**GPIO\_ENABLE\_W1TS** GPIO0 ~ 31 output enable set register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_ENABLE\\_REG](#) will be set to 1. Recommended operation: use this register to set [GPIO\\_ENABLE\\_REG](#). (WO)

**Register 5.11: GPIO\_ENABLE\_W1TC\_REG (0x0028)**

31		0
0x000000		Reset

**GPIO\_ENABLE\_W1TC** GPIO0 ~ 31 output enable clear register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_ENABLE\\_REG](#) will be cleared. Recommended operation: use this register to clear [GPIO\\_ENABLE\\_REG](#). (WO)

**Register 5.12: GPIO\_ENABLE1\_REG (0x002C)**

31		22	21	0
0	0	0	0	0x0000

**GPIO\_ENABLE1\_DATA** GPIO32 ~ 53 output enable register. (R/W)

**Register 5.13: GPIO\_ENABLE1\_W1TS\_REG (0x0030)**

31		22	21	0
0	0	0	0	0x0000

**GPIO\_ENABLE1\_W1TS** GPIO32 ~ 53 output enable set register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_ENABLE1\\_REG](#) will be set to 1. Recommended operation: use this register to set [GPIO\\_ENABLE1\\_REG](#). (WO)

**Register 5.14: GPIO\_ENABLE1\_W1TC\_REG (0x0034)**

GPIO_ENABLE1_W1TC											
(reserved)											
31	22	21	0								
0	0	0	0	0	0	0	0	0	0	0x0000	Reset

**GPIO\_ENABLE1\_W1TC** GPIO32 ~ 53 output enable clear register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_ENABLE1\\_REG](#) will be cleared. Recommended operation: use this register to clear [GPIO\\_ENABLE1\\_REG](#). (WO)

**Register 5.15: GPIO\_STRAP\_REG (0x0038)**

GPIO_STRAPPING												
(reserved)												
31	16	15	0									
0	0	0	0	0	0	0	0	0	0	0	0	Reset

**GPIO\_STRAPPING** GPIO strapping values: bit4 ~ bit2 correspond to stripping pins GPIO45, GPIO0, and GPIO46 respectively. (RO)

**Register 5.16: GPIO\_IN\_REG (0x003C)**

GPIO_IN_DATA_NEXT		
(reserved)		
31	0	
0	0	Reset

**GPIO\_IN\_DATA\_NEXT** GPIO0 ~ 31 input value. Each bit represents a pad input value, 1 for high level and 0 for low level. (RO)

## Register 5.17: GPIO\_IN1\_REG (0x0040)

**GPIO\_IN\_DATA1\_NEXT** GPIO32 ~ 53 input value. Each bit represents a pad input value. (RO)

### Register 5.18: GPIO\_PIN $n$ \_REG ( $n$ : 0-53) (0x0074+4\* $n$ )

GPIO Pin Control Register																	
Register Description																	
Bit Field Details																	
31	(reserved)	18	17	13	12	11	10	9	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0x0	0x0	0	0x0	0	0x0	0	0x0	Reset

**GPIO\_PIN $n$ \_SYNC2\_BYPASS** For the second stage synchronization, GPIO input data can be synchronized on either edge of the APB clock. 0: no synchronization; 1: synchronized on falling edge; 2 and 3: synchronized on rising edge. (R/W)

**GPIO\_PIN*n*\_PAD\_DRIVER** Pad driver selection. 0: normal output; 1: open drain output. (R/W)

**GPIO\_PIN<sub>n</sub>\_SYNC1\_BYPASS** For the first stage synchronization, GPIO input data can be synchronized on either edge of the APB clock. 0: no synchronization; 1: synchronized on falling edge; 2 and 3: synchronized on rising edge. (R/W)

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

**GPIO\_PIN<sub>n</sub>\_WAKEUP\_ENABLE** GPIO wake-up enable bit, only wakes up the CPU from Light-sleep  
(R/W)

**GPIO\_PIN*n*\_CONFIG** Reserved (R/W)

**GPIO\_PIN*n*\_INT\_ENA** Interrupt enable bits. bit13: CPU interrupt enabled; bit14: CPU non-maskable interrupt enabled. (R/W)

**Register 5.19: GPIO\_FUNC $n$ \_IN\_SEL\_CFG\_REG ( $n$ : 0-255) (0x0154+4\* $n$ )**

The diagram shows the bit field layout of Register 5.19. It consists of a 32-bit register with the most significant bit (bit 31) labeled '(reserved)' and the least significant bit (bit 0) labeled 'Reset'. Between bit 31 and bit 0, there are four groups of bits: bit 8, bit 7, bit 6, and bit 5. Bit 8 is labeled 'GPIO\_FUNC $n$ \_IN\_SEL', bit 7 is labeled 'GPIO\_FUNC $n$ \_IN\_INV\_SEL', bit 6 is labeled 'GPIO\_SIG $n$ \_IN\_SEL', and bit 5 is labeled 'GPIO\_FUNC $n$ \_IN\_SEL'.

31		8	7	6	5	0
0	0	0	0	0	0	0x0 Reset

**GPIO\_FUNC $n$ \_IN\_SEL** Selection control for peripheral input signal  $m$ , selects a pad from the 54 GPIO matrix pads to connect this input signal. Or selects 0x38 for a constantly high input or 0x3C for a constantly low input. (R/W)

**GPIO\_FUNC $n$ \_IN\_INV\_SEL** Invert the input value. 1: invert enabled; 0: invert disabled. (R/W)

**GPIO\_SIG $n$ \_IN\_SEL** Bypass GPIO matrix. 1: route signals via GPIO matrix, 0: connect signals directly to peripheral configured in IO\_MUX. (R/W)

**Register 5.20: GPIO\_FUNC $n$ \_OUT\_SEL\_CFG\_REG ( $n$ : 0-53) (0x0554+4\* $n$ )**

The diagram shows the bit field layout of Register 5.20. It consists of a 32-bit register with the most significant bit (bit 31) labeled '(reserved)' and the least significant bit (bit 0) labeled 'Reset'. Between bit 31 and bit 0, there are four groups of bits: bit 12, bit 11, bit 10, and bit 9. Bit 12 is labeled 'GPIO\_FUNC $n$ \_OUT\_SEL', bit 11 is labeled 'GPIO\_FUNC $n$ \_OUT\_INV\_SEL', bit 10 is labeled 'GPIO\_FUNC $n$ \_OEN\_SEL', and bit 9 is labeled 'GPIO\_FUNC $n$ \_OEN\_INV\_SEL'.

31		12	11	10	9	8	0
0	0	0	0	0	0	0	0x100 Reset

**GPIO\_FUNC $n$ \_OUT\_SEL** Selection control for GPIO output  $n$ . If a value  $s$  ( $0 \leq s < 256$ ) is written to this field, the peripheral output signal  $s$  will be connected to GPIO output  $n$ . If a value 256 is written to this field, bit  $n$  of [GPIO\\_OUT\\_REG](#)/[GPIO\\_OUT1\\_REG](#) and [GPIO\\_ENABLE\\_REG](#)/[GPIO\\_ENABLE1\\_REG](#) will be selected as the output value and output enable. (R/W)

**GPIO\_FUNC $n$ \_OUT\_INV\_SEL** 0: Do not invert the output value; 1: Invert the output value. (R/W)

**GPIO\_FUNC $n$ \_OEN\_SEL** 0: Use output enable signal from peripheral; 1: Force the output enable signal to be sourced from bit  $n$  of [GPIO\\_ENABLE\\_REG](#). (R/W)

**GPIO\_FUNC $n$ \_OEN\_INV\_SEL** 0: Do not invert the output enable signal; 1: Invert the output enable signal. (R/W)

**Register 5.21: GPIO\_CLOCK\_GATE\_REG (0x062C)**

(reserved)																																	

**GPIO\_CLK\_EN** Clock gating enable bit. If set to 1, the clock is free running. (R/W)

**Register 5.22: GPIO\_STATUS\_W1TS\_REG (0x0048)**

(reserved)																																

**GPIO\_STATUS\_W1TS** GPIO0 ~ 31 interrupt status set register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_STATUS\\_INTERRUPT](#) will be set to 1. Recommended operation: use this register to set [GPIO\\_STATUS\\_INTERRUPT](#). (WO)

**Register 5.23: GPIO\_STATUS\_W1TC\_REG (0x004C)**

(reserved)																																

**GPIO\_STATUS\_W1TC** GPIO0 ~ 31 interrupt status clear register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_STATUS\\_INTERRUPT](#) will be cleared. Recommended operation: use this register to clear [GPIO\\_STATUS\\_INTERRUPT](#). (WO)

**Register 5.24: GPIO\_STATUS1\_W1TS\_REG (0x0054)**

								GPIO_STATUS1_W1TS		
								(reserved)		Reset
31	22	0	0	0	0	0	0	0	0	0x0000

**GPIO\_STATUS1\_W1TS** GPIO32 ~ 53 interrupt status set register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_STATUS1\\_REG](#) will be set to 1. Recommended operation: use this register to set [GPIO\\_STATUS1\\_REG](#). (WO)

**Register 5.25: GPIO\_STATUS1\_W1TC\_REG (0x0058)**

								GPIO_STATUS1_W1TC		
								(reserved)		Reset
31	22	0	0	0	0	0	0	0	0	0x0000

**GPIO\_STATUS1\_W1TC** GPIO32 ~ 53 interrupt status clear register. If the value 1 is written to a bit here, the corresponding bit in [GPIO\\_STATUS1\\_REG](#) will be cleared. Recommended operation: use this register to clear [GPIO\\_STATUS1\\_REG](#). (WO)

**Register 5.26: GPIO\_STATUS\_NEXT\_REG (0x014C)**

								GPIO_STATUS_INTERRUPT_NEXT		
								(reserved)		Reset
31	22	0	0	0	0	0	0	0	0	0x000000

**GPIO\_STATUS\_INTERRUPT\_NEXT** Interrupt source signal of GPIO0 ~ 31, could be rising edge interrupt, falling edge interrupt, level sensitive interrupt and any edge interrupt. (RO)

**Register 5.27: GPIO\_STATUS\_NEXT1\_REG (0x0150)**

(reserved)										GPIO_STATUS1_INTERRUPT_NEXT	
31 0 0 0 0 0 0 0 0 0   22 0x0000										0 Reset	

**GPIO\_STATUS1\_INTERRUPT\_NEXT** Interrupt source signal of GPIO32 ~ 53. (RO)

**Register 5.28: GPIO\_STATUS\_REG (0x0044)**

(reserved)										GPIO_STATUS_INTERRUPT	
31 0 0 0 0 0 0 0 0 0   22 0x000000										0 Reset	

**GPIO\_STATUS\_INTERRUPT** GPIO0 ~ 31 interrupt status register. (R/W)

**Register 5.29: GPIO\_STATUS1\_REG (0x0050)**

(reserved)										GPIO_STATUS1_INTERRUPT	
31 0 0 0 0 0 0 0 0 0   22 0x0000										0 Reset	

**GPIO\_STATUS1\_INTERRUPT** GPIO32 ~ 53 interrupt status register. (R/W)

**Register 5.30: GPIO\_PCPU\_INT\_REG (0x005C)**

GPIO_PCPU_INT	
31	0
0x000000	Reset

**GPIO\_PCPU\_INT** GPIO0 ~ 31 PRO\_CPU interrupt status. This interrupt status is corresponding to the bit in [GPIO\\_STATUS\\_REG](#) when assert (high) enable signal (bit13 of [GPIO\\_PINn\\_REG](#)). (RO)

**Register 5.31: GPIO\_PCPU\_NMI\_INT\_REG (0x0060)**

GPIO_PCPU_NMI_INT	
31	0
0x000000	Reset

**GPIO\_PCPU\_NMI\_INT** GPIO0 ~ 31 PRO\_CPU non-maskable interrupt status. This interrupt status is corresponding to the bit in [GPIO\\_STATUS\\_REG](#) when assert (high) enable signal (bit 14 of [GPIO\\_PINn\\_REG](#)). (RO)

**Register 5.32: GPIO\_PCPU\_INT1\_REG (0x0068)**

GPIO_PCPU1_INT	
(reserved)	0
31	22
0 0 0 0 0 0 0 0 0 0	21 0
0x0000	Reset

**GPIO\_PCPU1\_INT** GPIO32 ~ 53 PRO\_CPU interrupt status. This interrupt status is corresponding to the bit in [GPIO\\_STATUS1\\_REG](#) when assert (high) enable signal (bit 13 of [GPIO\\_PINn\\_REG](#)). (RO)

**Register 5.33: GPIO\_PCPU\_NMI\_INT1\_REG (0x006C)**

31	22	21	0
0	0	0	0
(reserved)			0x0000

Reset

**GPIO\_PROCPU\_NMI1\_INT** GPIO32 ~ 53 PRO\_CPU non-maskable interrupt status. This interrupt status is corresponding to bit in [GPIO\\_STATUS1\\_REG](#) when assert (high) enable signal (bit 14 of [GPIO\\_PINn\\_REG](#)). (RO)

**5.14.2 IO MUX Registers****Register 5.34: IO\_MUX\_PIN\_CTRL (0x0000)**

31	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x0		0x0		0x2		0x0		0x0		0x0		0x0		0x0		Reset										

**IO\_MUX\_PIN\_CTRL\_CLKx** If you want to output clock for I2S0 to:

CLK\_OUT1 then set IO\_MUX\_PIN\_CTRL[3:0] = 0x0

CLK\_OUT2 then set IO\_MUX\_PIN\_CTRL[3:0] = 0x0 and IO\_MUX\_PIN\_CTRL[7:4] = 0x0;

CLK\_OUT3 then set IO\_MUX\_PIN\_CTRL[3:0] = 0x0 and IO\_MUX\_PIN\_CTRL[11:8] = 0x0.

**Note:**

Only the above mentioned combinations of clock source and clock output pins are possible.

The CLK\_OUT1 ~ 3 can be found in [IO\\_MUX Pad List](#).

**IO\_MUX\_SWITCH\_PRT\_NUM** IO pad power switch delay, delay unit is one APB clock.

**IO\_MUX\_PAD\_POWER\_CTRL** Select power voltage for GPIO33 ~ 37. 1: select VDD\_SPI 1.8 V; 0: select VDD3P3\_CPU 3.3 V.

**Register 5.35: IO\_MUX\_<REG> (*n*: GPIO0-GPIO21, GPIO26-GPIO46) (0x0010+4\*<i>n</i>)**

																<i>IO_MUX_FILTER_EN</i>	<i>IO_MUX_MCU_SEL</i>	<i>IO_MUX_FUN_DRV</i>	<i>IO_MUX_FUN_IE</i>	<i>IO_MUX_FUN_WPU</i>	<i>IO_MUX_MCU_WPU</i>	<i>IO_MUX_MCU_WPD</i>	<i>IO_MUX_MCU_IE</i>	<i>IO_MUX_MCU_WPD</i>	<i>IO_MUX_SLP_SEL</i>	<i>IO_MUX_MCU_OF</i>					
31	(reserved)															16	15	14	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0	0x0	0x0	0	0	0	0	0	0	0	0	0	0	0	0	Reset	

**IO\_MUX\_MCU\_OE** Output enable of the pad in sleep mode. 1: Output enabled; 0: Output disabled.  
(R/W)

**IO\_MUX\_SLP\_SEL** Sleep mode selection of this pad. Set to 1 to put the pad in sleep mode. (R/W)

**IO\_MUX\_MCU\_WPD** Pull-down enable of the pad during sleep mode. 1: Internal pull-down enabled;  
0: Internal pull-down disabled. (R/W)

**IO\_MUX\_MCU\_WPU** Pull-up enable of the pad during sleep mode. 1: Internal pull-up enabled; 0:  
Internal pull-up disabled.

**IO\_MUX\_MCU\_IE** Input enable of the pad during sleep mode. 1: Input enabled; 0: Input disabled.  
(R/W)

**IO\_MUX\_FUN\_WPD** Pull-down enable of the pad. 1: Pull-down enabled; 0: Pull-down disabled.  
(R/W)

**IO\_MUX\_FUN\_WPU** Pull-up enable of the pad. 1: Internal pull-up enabled; 0: Internal pull-up dis-  
abled. (R/W)

**IO\_MUX\_FUN\_IE** Input enable of the pad. 1: Input enabled; 0: Input disabled. (R/W)

**IO\_MUX\_FUN\_DRV** Select the drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40  
mA. (R/W)

**IO\_MUX\_MCU\_SEL** Select IO MUX function for this signal. 0: Select Function 1; 1: Select Function  
2, etc. (R/W)

**IO\_MUX\_FILTER\_EN** Enable filter for pin input signals. 1: Filter enabled; 2: Filter disabled. (R/W)

### 5.14.3 Sigma Delta Modulated Output Registers

Register 5.36: GPIOSD\_SIGMADELTA $n$ \_REG ( $n$ : 0-7) (0x0000+4\* $n$ )

31	(reserved)	16	15	8	7	0
0	0	0	0	0	0	0
0xff						0x0
						Reset

**GPIOSD\_SD $n$ \_IN** This field is used to configure the duty cycle of sigma delta modulation output.

(R/W)

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

Register 5.37: GPIOSD\_SIGMADELTA(CG)\_REG (0x0020)

31	30	(reserved)	(reserved)	0
0	0	0	0	0

**GPIOSD\_CLK\_EN** Clock enable bit of configuration registers for sigma delta modulation. (R/W)

Register 5.38: GPIOSD\_SIGMADELTA\_MISC\_REG (0x0024)

31	30	29	(reserved)	0
0	0	0	0	0

**GPIOSD\_FUNCTION\_CLK\_EN** Clock enable bit of sigma delta modulation. (R/W)

**GPIOSD\_SPI\_SWAP** Reserved. (R/W)

Register 5.39: GPIOSD\_SIGMADELTA\_VERSION\_REG (0x0028)

				GPIOSD_GPIO_SD_DATE	
				(reserved)	0
31	28	27			
0	0	0	0	0x1802260	Reset

**GPIOSD\_GPIO\_SD\_DATE** Version control register. (R/W)

#### 5.14.4 RTC IO MUX Registers

Register 5.40: RTCIO\_RTC\_GPIO\_OUT\_REG (0x0000)

				RTCIO_GPIO_OUT_DATA	
				(reserved)	0
31		10	9		
0		0	0	0	0

**RTCIO\_GPIO\_OUT\_DATA** GPIO0 ~ 21 output register. Bit10 corresponds to GPIO0, bit11 corresponds to GPIO1, etc. (R/W)

Register 5.41: RTCIO\_RTC\_GPIO\_OUT\_W1TS\_REG (0x0004)

				RTCIO_GPIO_OUT_DATA_W1TS	
				(reserved)	0
31		10	9		
0		0	0	0	0

**RTCIO\_GPIO\_OUT\_DATA\_W1TS** GPIO0 ~ 21 output set register. If the value 1 is written to a bit here, the corresponding bit in [RTCIO\\_RTC\\_GPIO\\_OUT\\_REG](#) will be set to 1. Recommended operation: use this register to set [RTCIO\\_RTC\\_GPIO\\_OUT\\_REG](#). (WO)

**Register 5.42: RTCIO\_RTC\_GPIO\_OUT\_W1TC\_REG (0x0008)**

		RTCIO_GPIO_OUT_DATA_W1TC										(reserved)	
31	10	9	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0	0	Reset	

**RTCIO\_GPIO\_OUT\_DATA\_W1TC** GPIO0 ~ 21 output clear register. If the value 1 is written to a bit here, the corresponding bit in [RTCIO\\_RTC\\_GPIO\\_OUT\\_REG](#) will be cleared. Recommended operation: use this register to clear [RTCIO\\_RTC\\_GPIO\\_OUT\\_REG](#). (WO)

**Register 5.43: RTCIO\_RTC\_GPIO\_ENABLE\_REG (0x000C)**

		RTCIO_GPIO_ENABLE										(reserved)	
31	10	9	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0	0	Reset	

**RTCIO\_GPIO\_ENABLE** GPIO0 ~ 21 output enable. Bit10 corresponds to GPIO0, bit11 corresponds to GPIO1, etc. If the bit is set to 1, it means this GPIO pad is output. (R/W)

**Register 5.44: RTCIO\_RTC\_GPIO\_ENABLE\_W1TS\_REG (0x0010)**

		RTCIO_GPIO_ENABLE_W1TS										(reserved)	
31	10	9	0	0	0	0	0	0	0	0	0		
0	0	0	0	0	0	0	0	0	0	0	0	Reset	

**RTCIO\_GPIO\_ENABLE\_W1TS** GPIO0 ~ 21 output enable set register. If the value 1 is written to a bit here, the corresponding bit in [RTCIO\\_RTC\\_GPIO\\_ENABLE\\_REG](#) will be set to 1. Recommended operation: use this register to set [RTCIO\\_RTC\\_GPIO\\_ENABLE\\_REG](#). (WO)

#### Register 5.45: RTCIO\_RTC\_GPIO\_ENABLE\_W1TC\_REG (0x0014)

RTCIO_GPIO_ENABLE_WTC		(reserved)								
31	0	10	9	0	0	0	0	0	0	Reset

**RTCIO\_GPIO\_ENABLE\_W1TC** GPIO0 ~ 21 output enable clear register. If the value 1 is written to a bit here, the corresponding bit in [RTCIO\\_RTC\\_GPIO\\_ENABLE\\_REG](#) will be cleared. Recommended operation: use this register to clear [RTCIO\\_RTC\\_GPIO\\_ENABLE\\_REG](#). (WO)

#### Register 5.46: RTCIO\_RTC\_GPIO\_STATUS\_REG (0x0018)

**RTCIO\_GPIO\_STATUS\_INT** GPIO0 ~ 21 interrupt status register. Bit10 corresponds to GPIO0, bit11 corresponds to GPIO1, etc. This register should be used together with [RT-CIO\\_RTC\\_GPIO\\_PINn\\_INT\\_TYPE](#) in [RTCIO\\_RTC\\_GPIO\\_PINn\\_REG](#). 0: no interrupt; 1: corresponding interrupt. (R/W)

#### Register 5.47: RTCIO\_RTC\_GPIO\_STATUS\_W1TS\_REG (0x001C)

**RTCIO\_GPIO\_STATUS\_INT\_W1TS** GPIO0 ~ 21 interrupt set register. If the value 1 is written to a bit here, the corresponding bit in [RTCIO\\_GPIO\\_STATUS\\_INT](#) will be set to 1. Recommended operation: use this register to set [RTCIO\\_GPIO\\_STATUS\\_INT](#). (WO)

**Register 5.48: RTCIO\_RTC\_GPIO\_STATUS\_W1TC\_REG (0x0020)**

RTCIO\_GPIO\_STATUS\_INT\_W1TC

31	10	9	0
0	0	0	0

(reserved)

Reset

**RTCIO\_GPIO\_STATUS\_INT\_W1TC** GPIO0 ~ 21 interrupt clear register. If the value 1 is written to a bit here, the corresponding bit in [RTCIO\\_GPIO\\_STATUS\\_INT](#) will be cleared. Recommended operation: use this register to clear [RTCIO\\_GPIO\\_STATUS\\_INT](#). (WO)

**Register 5.49: RTCIO\_RTC\_GPIO\_IN\_REG (0x0024)**

RTCIO\_GPIO\_IN\_NEXT

31	10	9	0
0	0	0	0

(reserved)

Reset

**RTCIO\_GPIO\_IN\_NEXT** GPIO0 ~ 21 input value. Bit10 corresponds to GPIO0, bit11 corresponds to GPIO1, etc. Each bit represents a pad input value, 1 for high level, and 0 for low level. (RO)

**Register 5.50: RTCIO\_RTC\_GPIO\_PIN $n$ \_REG ( $n$ : 0-21) (0x0028+4\* $n$ )**

The register map shows the layout of the RTCIO\_RTC\_GPIO\_PIN $n$ \_REG register. It consists of a 32-bit wide register divided into several fields:

- Bit 31:** (reserved)
- Bits 11-10:** (reserved)
- Bit 9:** (reserved)
- Bits 7-6:** RTCIO\_GPIO\_PIN $n$ \_WAKEUP\_ENABLE
- Bit 5:** (reserved)
- Bit 4:** RTCIO\_GPIO\_PIN $n$ \_INT\_TYPE
- Bits 3-2:** (reserved)
- Bit 1:** RTCIO\_GPIO\_PIN $n$ \_PAD\_DRIVER
- Bit 0:** (reserved)

**Reset:** The register is reset to 0x00000000.

**RTCIO\_GPIO\_PIN $n$ \_PAD\_DRIVER** Pad driver selection. 0: normal output; 1: open drain. (R/W)

**RTCIO\_GPIO\_PIN $n$ \_INT\_TYPE** GPIO interrupt type selection. 0: GPIO interrupt disabled; 1: rising edge trigger; 2: falling edge trigger; 3: any edge trigger; 4: low level trigger; 5: high level trigger. (R/W)

**RTCIO\_GPIO\_PIN $n$ \_WAKEUP\_ENABLE** GPIO wake-up enable. This will only wake up ESP32-S2 from Light-sleep. (R/W)

**Register 5.51: RTCIO\_TOUCH\_PAD $n$ \_REG ( $n$ : 0-14) (0x0084+4\* $n$ )**

(reserved)	RTCIO_TOUCH_PAD $n$ _DRV	RTCIO_TOUCH_PAD $n$ _RDE	RTCIO_TOUCH_PAD $n$ _RUE	(reserved)	RTCIO_TOUCH_PAD $n$ _DAC	RTCIO_TOUCH_PAD $n$ _START	RTCIO_TOUCH_PAD $n$ _TIE_OPT	RTCIO_TOUCH_PAD $n$ _XPD	RTCIO_TOUCH_PAD $n$ _MUX_SEL	RTCIO_TOUCH_PAD $n$ _SLP_SEL	RTCIO_TOUCH_PAD $n$ _SLP_IE	RTCIO_TOUCH_PAD $n$ _SLP_OF	RTCIO_TOUCH_PAD $n$ _SLP_IF	RTCIO_TOUCH_PAD $n$ _SLP_OE	RTCIO_TOUCH_PAD $n$ _FUN_SEL	RTCIO_TOUCH_PAD $n$ _SLP_SEL	RTCIO_TOUCH_PAD $n$ _SLP_IF	RTCIO_TOUCH_PAD $n$ _SLP_OE	RTCIO_TOUCH_PAD $n$ _FUN_IF	(reserved)
31	30	29	28	27	26	25	23	22	21	20	19	18	17	16	15	14	13	12	0	0
0	2	1	0	0	0x4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Reset

**RTCIO\_TOUCH\_PAD $n$ \_FUN\_IE** Input enable in normal execution. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_SLP\_OE** Output enable in sleep mode. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_SLP\_IE** Input enable in sleep mode. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_SLP\_SEL** 0: no sleep mode; 1: enable sleep mode. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_FUN\_SEL** Function selection. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_MUX\_SEL** Connect the RTC pad input to digital pad input. 0 is available. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_XPD** Touch sensor power on. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_TIE\_OPT** The tie option of touch sensor. 0: tie low; 1: tie high. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_START** Start touch sensor. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_DAC** Touch sensor slope control. 3-bit for each touch pad, defaults to 0x4. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_RUE** Pull-down enable of the pad. 1: internal pull-down enabled, 0: internal pull-down disabled. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_RDE** Pull-up enable of the pad. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)**RTCIO\_TOUCH\_PAD $n$ \_DRV** Select the drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA. (R/W)

## Register 5.52: RTCIO\_XTAL\_32P\_PAD\_REG (0x00C0)

(reserved)	RTCIO_X32P_DRV	RTCIO_X32P_RDE	RTCIO_X32P_RUE	(reserved)	RTCIO_X32P_MUX_SEL	RTCIO_X32P_FUN_SEL	RTCIO_X32P_SLP_SEL	RTCIO_X32P_SLP_IE	RTCIO_X32P_SLP_OE	RTCIO_X32P_FUN_IE	(reserved)	0					
31	30	29	28	27	26	20	19	18	17	16	15	14	13	12	0	Reset	
0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

**RTCIO\_X32P\_FUN\_IE** Input enable in normal execution. (R/W)

**RTCIO\_X32P\_SLP\_OE** Output enable in sleep mode. (R/W)

**RTCIO\_X32P\_SLP\_IE** Input enable in sleep mode. (R/W)

**RTCIO\_X32P\_SLP\_SEL** 1: enable sleep mode; 0: no sleep mode (R/W)

**RTCIO\_X32P\_FUN\_SEL** Function selection (R/W)

**RTCIO\_X32P\_MUX\_SEL** 1: use RTC GPIO, 0: use digital GPIO (R/W)

**RTCIO\_X32P\_RUE** Pull-down enable of the pad. 1: internal pull-down enabled; 0: internal pull-down disabled. (R/W)

**RTCIO\_X32P\_RDE** Pull-up enable of the pad. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

**RTCIO\_X32P\_DRV** Select the drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA. (R/W)

## Register 5.53: RTCIO\_XTAL\_32N\_PAD\_REG (0x00C4)

31	30	29	28	27	26	20	19	18	17	16	15	14	13	12	0
0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

Labels for fields:

- (reserved) RTCIO\_X32N\_DRV
- (reserved) RTCIO\_X32N\_RDE
- (reserved) RTCIO\_X32N\_RUE
- (reserved)
- RTCIO\_X32N\_MUX\_SEL
- RTCIO\_X32N\_FUN\_SEL
- RTCIO\_X32N\_SLP\_SEL
- RTCIO\_X32N\_SLP\_IE
- RTCIO\_X32N\_SLP\_OE
- RTCIO\_X32N\_FUN\_IE
- (reserved)

**RTCIO\_X32N\_FUN\_IE** Input enable in normal execution. (R/W)

**RTCIO\_X32N\_SLP\_OE** Output enable in sleep mode. (R/W)

**RTCIO\_X32N\_SLP\_IE** Input enable in sleep mode. (R/W)

**RTCIO\_X32N\_SLP\_SEL** 1: enable sleep mode; 0: no sleep mode (R/W)

**RTCIO\_X32N\_FUN\_SEL** Function selection (R/W)

**RTCIO\_X32N\_MUX\_SEL** 1: use RTC GPIO, 0: use digital GPIO (R/W)

**RTCIO\_X32N\_RUE** Pull-down enable of the pad. 1: internal pull-down enabled; 0: internal pull-down disabled. (R/W)

**RTCIO\_X32N\_RDE** Pull-up enable of the pad. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

**RTCIO\_X32N\_DRV** Select drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA. (R/W)

## Register 5.54: RTCIO\_PAD\_DAC1\_REG (0x00C8)

(reserved)	RTClO_PDAC1_DRV	RTClO_PDAC1_RDE	RTClO_PDAC1_RUE	(reserved)	RTClO_PDAC1_MUX_SEL	RTClO_PDAC1_FUN_SEL	RTClO_PDAC1_SLP_SEL	RTClO_PDAC1_SLP_IE	RTClO_PDAC1_SLP_OE	RTClO_PDAC1_FUN_IE	RTClO_PDAC1_DAC_XPD_FORCE	RTClO_PDAC1_XPD_DAC	(reserved)						
31	30	29	28	27	26	20	19	18	17	16	15	14	13	12	11	10	3	2	0
0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**RTClO\_PDAC1\_DAC** Configure DAC\_1 output when [RTClO\\_PDAC1\\_DAC\\_XPD\\_FORCE](#) is set to 1.

(R/W)

**RTClO\_PDAC1\_XPD\_DAC** When [RTClO\\_PDAC1\\_DAC\\_XPD\\_FORCE](#) is set to 1, 1: enable DAC\_1 output; 0: disable DAC\_1 output. (R/W)

**RTClO\_PDAC1\_DAC\_XPD\_FORCE** 1: use [RTClO\\_PDAC1\\_XPD\\_DAC](#) to control DAC\_1 output; 0: use SAR ADC FSM to control DAC\_1 output. (R/W)

**RTClO\_PDAC1\_FUN\_IE** Input enable in normal execution. (R/W)

**RTClO\_PDAC1\_SLP\_OE** Output enable in sleep mode. (R/W)

**RTClO\_PDAC1\_SLP\_IE** Input enable in sleep mode. (R/W)

**RTClO\_PDAC1\_SLP\_SEL** 1: enable sleep mode; 0: no sleep mode. (R/W)

**RTClO\_PDAC1\_FUN\_SEL** DAC\_1 function selection. (R/W)

**RTClO\_PDAC1\_MUX\_SEL** 1: use RTC GPIO, 0: use digital GPIO (R/W)

**RTClO\_PDAC1\_RUE** Pull-down enable of the pad. 1: internal pull-down enabled, 0: internal pull-down disabled. (R/W)

**RTClO\_PDAC1\_RDE** Pull-up enable of the pad. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

**RTClO\_PDAC1\_DRV** Select drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA. (R/W)

## Register 5.55: RTCIO\_PAD\_DAC2\_REG (0x00CC)

(reserved)	RTClO_PDAC2_DRV	RTClO_PDAC2_RDE	RTClO_PDAC2_RUE	(reserved)	RTClO_PDAC2_MUX_SEL	RTClO_PDAC2_FUN_SEL	RTClO_PDAC2_SLP_SEL	RTClO_PDAC2_SLP_IE	RTClO_PDAC2_SLP_OE	RTClO_PDAC2_FUN_IE	RTClO_PDAC2_DAC_XPD_FORCE	RTClO_PDAC2_XPD_DAC	(reserved)						
31	30	29	28	27	26	20	19	18	17	16	15	14	13	12	11	10	3	2	0
0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**RTClO\_PDAC2\_DAC** Configure DAC\_2 output when [RTClO\\_PDAC2\\_DAC\\_XPD\\_FORCE](#) is set to 1.  
(R/W)

**RTClO\_PDAC2\_XPD\_DAC** When [RTClO\\_PDAC2\\_DAC\\_XPD\\_FORCE](#) is set to 1, 1: enable DAC\_2 output; 0: disable DAC\_2 output. (R/W)

**RTClO\_PDAC2\_DAC\_XPD\_FORCE** 1: use [RTClO\\_PDAC2\\_XPD\\_DAC](#) to control DAC\_2 output; 0: use SAR ADC FSM to control DAC\_2 output. (R/W)

**RTClO\_PDAC2\_FUN\_IE** Input enable in normal execution. (R/W)

**RTClO\_PDAC2\_SLP\_OE** Output enable in sleep mode. (R/W)

**RTClO\_PDAC2\_SLP\_IE** Input enable in sleep mode. (R/W)

**RTClO\_PDAC2\_SLP\_SEL** 1: enable sleep mode; 0: no sleep mode (R/W)

**RTClO\_PDAC2\_FUN\_SEL** DAC\_2 function selection. (R/W)

**RTClO\_PDAC2\_MUX\_SEL** 1: use RTC GPIO, 0: use digital GPIO. (R/W)

**RTClO\_PDAC2\_RUE** Pull-down enable of the pad. 1: internal pull-down enabled, 0: internal pull-down disabled. (R/W)

**RTClO\_PDAC2\_RDE** Pull-up enable of the pad. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

**RTClO\_PDAC2\_DRV** Select drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA. (R/W)

## Register 5.56: RTCIO\_RTC\_PAD19\_REG (0x00D0)

**RTCIO\_RTC\_PAD19\_FUN\_IE** Input enable in normal execution. (R/W)

**RTCIO\_RTC\_PAD19\_SLP\_OE** Output enable in sleep mode. (R/W)

**RTCIO\_RTC\_PAD19\_SLP\_IE** Input enable in sleep mode. (R/W)

**RTCIO\_RTC\_PAD19\_SLP\_SEL** 1: enable sleep mode; 0: no sleep mode. (R/W)

**RTCIO\_RTC\_PAD19\_FUN\_SEL** Function selection (R/W)

**RTCIO\_RTC\_PAD19\_MUX\_SEL** 1: use RTC GPIO, 0: use digital GPIO (R/W)

**RTCIO\_RTC\_PAD19\_RUE** Pull-down enable of the pad. 1: internal pull-down enabled, 0: internal pull-down disabled. (R/W)

**RTCIO\_RTC\_PAD19\_RDE** Pull-up enable of the pad. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

**RTCIO\_RTC\_PAD19\_DRV** Select drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA. (R/W)

## Register 5.57: RTCIO\_RTC\_PAD20\_REG (0x00D4)

**RTCIO\_RTC\_PAD20\_FUN\_IE** Input enable in normal execution. (R/W)

**RTCIO\_RTC\_PAD20\_SLP\_OE** Output enable in sleep mode. (R/W)

**RTCIO\_RTC\_PAD20\_SLP\_IE** Input enable in sleep mode. (R/W)

**RTCIO\_RTC\_PAD20\_SLP\_SEL** 1: enable sleep mode; 0: no sleep mode. (R/W)

**RTCIO\_RTC\_PAD20\_FUN\_SEL** Function selection. (R/W)

**RTCIO\_RTC\_PAD20\_MUX\_SEL** 1: use RTC GPIO, 0: use digital GPIO. (R/W)

**RTCIO\_RTC\_PAD20\_RUE** Pull-down enable of the pad. 1: internal pull-down enabled; 0: internal pull-down disabled. (R/W)

**RTCIO\_RTC\_PAD20\_RDE** Pull-up enable of the pad. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

**RTCIO\_RTC\_PAD20\_DRV** Select drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA. (R/W)

**Register 5.58: RTCIO\_RTC\_PAD21\_REG (0x00D8)**

31	30	29	28	27	26	20	19	18	17	16	15	14	13	12		0	Reset
0	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

**RTCIO\_RTC\_PAD21\_FUN\_IE** Input enable in normal execution. (R/W)

**RTCIO\_RTC\_PAD21\_SLP\_OE** Output enable in sleep mode. (R/W)

**RTCIO\_RTC\_PAD21\_SLP\_IE** Input enable in sleep mode. (R/W)

**RTCIO\_RTC\_PAD21\_SLP\_SEL** 1: enable sleep mode; 0: no sleep mode. (R/W)

**RTCIO\_RTC\_PAD21\_FUN\_SEL** Function selection. (R/W)

**RTCIO\_RTC\_PAD21\_MUX\_SEL** 1: use RTC GPIO,0: use digital GPIO. (R/W)

**RTCIO\_RTC\_PAD21\_RUE** Pull-down enable of the pad. 1: internal pull-down enabled; 0: internal pull-down disabled. (R/W)

**RTCIO\_RTC\_PAD21\_RDE** Pull-up enable of the pad. 1: internal pull-up enabled; 0: internal pull-up disabled. (R/W)

**RTCIO\_RTC\_PAD21\_DRV** Select drive strength of the pad. 0: ~5 mA; 1: ~10 mA; 2: ~20 mA; 3: ~40 mA. (R/W)

**Register 5.59: RTCIO\_XTL\_EXT\_CTR\_REG (0x00E0)**

31	27	26		0	Reset
0	0	0	0	0	0

**RTCIO\_XTL\_EXT\_CTR\_SEL** Select the external crystal power down enable source to get into sleep mode. 0: select GPIO0; 1: select GPIO1, etc. The input value on this pin XOR RTC\_CNTL\_EXT\_XTL\_CONF\_REG[30] is the crystal power down enable signal. (R/W)

**Register 5.60: RTCIO\_SAR\_I2C\_IO\_REG (0x00E4)**

31	30	29	28	27			(reserved)	0	Reset
0	0	0	0	0	0	0	0	0	0

**RTCIO\_SAR\_I2C\_SCL\_SEL** Selects a pad the RTC I2C SCL signal connects to. 0: use TOUCH PAD0; 1: use TOUCH PAD2. (R/W)

**RTCIO\_SAR\_I2C\_SDA\_SEL** Selects a pad the RTC I2C SDA signal connects to. 0: use TOUCH PAD1; 1: use TOUCH PAD3. (R/W)

# 6. System Registers

## 6.1 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 variety 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.

## 6.2 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

## 6.3 Function Description

### 6.3.1 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 30 below.

**Table 30: 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 31 below.

**Table 31: 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

### 6.3.2 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](#)

### 6.3.3 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](#)

### 6.3.4 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

### 6.3.5 Low-power Management Registers

The following registers are used for low-power management.

- SYSTEM\_RTC\_FASTMEM\_CONFIG\_REG
- SYSTEM\_RTC\_FASTMEM\_CRC\_REG

### 6.3.6 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 32.

- 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 32: Peripheral Clock Gating and Reset Bits**

Peripheral	Clock Enabling Bit <sup>1</sup>	Reset Controlling Bit <sup>23</sup>
<b>CPU Peripherals</b>	<b>SYSTEM_CPU_PERI_CLK_EN_REG</b>	<b>SYSTEM_CPU_PERI_RST_EN_REG</b>
DEDICATED GPIO	SYSTEM_CLK_EN_DEDICATED_GPIO	SYSTEM_RST_EN_DEDICATED_GPIO
<b>Peripherals</b>	<b>SYSTEM_PERIP_CLK_EN0_REG</b>	<b>SYSTEM_PERIP_RST_EN0_REG</b>
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 <sup>4</sup>	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_LED_C CLK_EN	SYSTEM_LED_C_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
<b>Accelerators</b>	<b>SYSTEM_PERIP_CLK_EN1_REG</b>	<b>SYSTEM_PERIP_RST_EN1_REG</b>
DMA	SYSTEM_CRYPTO_DMA_CLK_EN	SYSTEM_CRYPTO_DMA_RST <sup>5</sup>
HMAC	SYSTEM_CRYPTO_HMAC_CLK_EN	SYSTEM_CRYPTO_HMAC_RST <sup>6</sup>
Digital Signature	SYSTEM_CRYPTO_DS_CLK_EN	SYSTEM_CRYPTO_DS_RST <sup>7</sup>
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.

## 6.4 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 33: System Register Base Address**

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F4C0000

## 6.5 Register Summary

The addresses in the following table are relative to the system registers base addresses provided in Section 6.4.

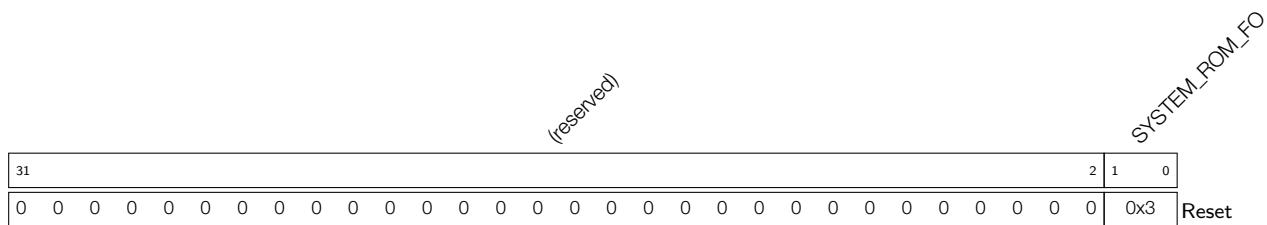
Name	Description	Address	Access
<b>System and Memory Registers</b>			
SYSTEM_ROM_CTRL_0_REG	System ROM configuration register 0	0x0000	R/W
SYSTEM_ROM_CTRL_1_REG	System ROM configuration register 1	0x0004	R/W
SYSTEM_SRAM_CTRL_0_REG	System SRAM configuration register 0	0x0008	R/W
SYSTEM_SRAM_CTRL_1_REG	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
<b>Reset and Clock Registers</b>			
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
<b>Interrupt Matrix Registers</b>			
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
<b>JTAG Software Enable Registers</b>			
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
<b>Low-Power Management Registers</b>			
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
<b>Peripheral Clock Gating and Reset Registers</b>			
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
<b>Version Register</b>			
SYSTEM_REG_DATE_REG	Version control register	0xFFFF	R/W

## 6.6 Registers

The addresses below are relative to the system registers base addresses provided in Section 6.4.

## Register 6.1: 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 30. (R/W)

## Register 6.2: SYSTEM\_ROM\_CTRL\_1\_REG (0x0004)

The diagram illustrates the memory map for the ROM area. It shows two registers: `SYSTEM_ROM_FORCE_PU` at address 31 and `SYSTEM_ROM_FORCE_PD` at address 0. The address range from 0 to 30 is labeled `(reserved)`.

31									4	3	2	1	0								
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	Reset

**SYSTEM\_ROM\_FORCE\_PD** This field is used to power down internal ROM. For details, please refer to Table 30. (R/W)

**SYSTEM\_ROM\_FORCE\_PU** This field is used to power up internal ROM. For details, please refer to Table 30. (R/W)

### Register 6.3: SYSTEM\_SRAM\_CTRL\_0\_REG (0x0008)

Register: SYSTEM\_SRAM\_FO

Address: 0x3fffff

Reset Value: 0

Bit Range: 31-22

Bit Name: (reserved)

**SYSTEM\_SRAM\_FO** This field is used to force on clock gate of internal SRAM. For details, please refer to Table 31. (R/W)

#### Register 6.4: 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 31. (R/W)

**Register 6.5: SYSTEM\_CPU\_PERI\_CLK\_EN\_REG (0x0010)**

The diagram shows the bit layout of Register 6.5. It consists of a 32-bit register with the most significant bit at index 31. The bits are grouped into four sections: a 24-bit field from index 8 to 0 labeled '(reserved)', a single bit labeled 'SYSTEM\_CLK\_EN\_DEDICATED\_GPIO' at index 5, another 24-bit field from index 0 to 1 labeled '(reserved)', and a single bit labeled 'Reset' at index 0.

31								8	7	6	5					0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**SYSTEM\_CLK\_EN\_DEDICATED\_GPIO** Set this bit to enable clock of DEDICATED GPIO module.

**Register 6.6: SYSTEM\_CPU\_PERI\_RST\_EN\_REG (0x0014)**

The diagram shows the bit layout of Register 6.6. It consists of a 32-bit register with the most significant bit at index 31. The bits are grouped into four sections: a 24-bit field from index 8 to 0 labeled '(reserved)', a single bit labeled 'SYSTEM\_RST\_EN\_DEDICATED\_GPIO' at index 5, another 24-bit field from index 0 to 1 labeled '(reserved)', and a single bit labeled 'Reset' at index 0.

31								8	7	6	5					0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**SYSTEM\_RST\_EN\_DEDICATED\_GPIO** Set this bit to reset DEDICATED GPIO module. (R/W)

**Register 6.7: SYSTEM\_CPU\_PER\_CONF\_REG (0x0018)**

The diagram shows the bit field layout of Register 6.7. The register is 32 bits wide, with bit 31 labeled '(reserved)' and bit 0 being the least significant bit. Bit 0 is also labeled 'Reset'. The bit fields are:

- SYSTEM\_CPU\_WAITI\_DELAY\_NUM**: Bits 7:4
- SYSTEM\_CPU\_WAIT\_MODE\_FORCE\_ON**: Bit 3
- SYSTEM\_PLL\_FREQ\_SEL**: Bit 2
- SYSTEM\_CPUPERIOD\_SEL**: Bit 1
- Reset**: Bit 0

31								8	7	4	3	2	1	0					
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0	1	1	0	Reset

**SYSTEM\_CPUPERIOD\_SEL** This field is used to select the clock frequency of CPU or CPU period.

For details, please refer to Table 8 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 8 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 forced 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 6.8: SYSTEM\_JTAG\_CTRL\_0\_REG (0x001C)**

The diagram shows the bit field layout of Register 6.8. The register is 32 bits wide, with bit 31 labeled '(reserved)' and bit 0 being the least significant bit. Bit 0 is also labeled 'Reset'. The bit field is:

- SYSTEM\_CANCEL\_EFUSE\_DISABLE\_JTAG\_TEMPORARY\_0**: Bits 31:0

31																																0
0																																Reset

**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 6.9: SYSTEM\_JTAG\_CTRL\_1\_REG (0x0020)**

SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_1	
31	0
0	Reset

**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 6.10: SYSTEM\_JTAG\_CTRL\_2\_REG (0x0024)**

SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_2	
31	0
0	Reset

**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 6.11: SYSTEM\_JTAG\_CTRL\_3\_REG (0x0028)**

SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_3	
31	0
0	Reset

**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 6.12: SYSTEM\_JTAG\_CTRL\_4\_REG (0x002C)**

SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_4	
31	0
0	Reset

**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 6.13: SYSTEM\_JTAG\_CTRL\_5\_REG (0x0030)**

SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_5	
31	0
0	Reset

**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 6.14: SYSTEM\_JTAG\_CTRL\_6\_REG (0x0034)**

SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_6	
31	0
0	Reset

**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 6.15: SYSTEM\_JTAG\_CTRL\_7\_REG (0x0038)

	31	0	0	Reset
SYSTEM_CANCEL_EFUSE_DISABLE_JTAG_TEMPORARY_7				

**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 6.16: SYSTEM\_MEM\_PD\_MASK\_REG (0x003C)

**SYSTEM\_LSPL\_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 6.17: 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 32.

## Register 6.18: SYSTEM\_PERIP\_CLK\_EN1\_REG (0x0044)

(reserved)			SYSTEM_CRYPTO_DMA_CLK_EN																																
(reserved)			SYSTEM_CRYPTO_HMAC_CLK_EN																																
(reserved)			SYSTEM_CRYPTO_DS_CLK_EN																																
(reserved)			SYSTEM_CRYPTO_RSA_CLK_EN																																
(reserved)			SYSTEM_CRYPTO_SHA_CLK_EN																																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
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	0	0	0	0	0	0	0	0	0	0	Reset

**SYSTEM\_PERIP\_CLK\_EN1\_REG** Configures this register to enable different accelerator clocks. For details, please refer to Table 32.

## Register 6.19: SYSTEM\_PERIP\_RST\_EN0\_REG (0x0048)

(reserved)																																				
(reserved)																																				
(reserved)																																				
(reserved)																																				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
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	0	0	0	0	0	0	0	0	0	0	Reset	

**SYSTEM\_PERIP\_RST\_EN0\_REG** Configures this register to reset different peripherals. For details, please refer to Table 32.

## Register 6.20: SYSTEM\_PERIP\_RST\_EN1\_REG (0x004C)

(reserved)																																				
(reserved)																																				
(reserved)																																				
(reserved)																																				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
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	0	0	0	0	0	0	0	0	0	0	Reset	

**SYSTEM\_PERIP\_RST\_EN1\_REG** Configures this register to reset different accelerators. For details, please refer to Table 32.

## Register 6.21: SYSTEM\_BT\_LPCK\_DIV\_FRAC\_REG (0x0054)

SYSTEM_BT_LPCK_DIV_FRAC_REG (0x0054)											
Bit Descriptions											
31	29	28	27	26	25	24	23	12	11		
0	0	0	0	0	0	1	0	1		1	Reset

**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 6.22: SYSTEM\_CPU\_INTR\_FROM\_CPU\_0\_REG (0x0058)

SYSTEM_CPU_INTR_FROM_CPU_0_REG (0x0058)																														
Bit Descriptions																														
31																														
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	0	0	0	0	0	0

**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 6.23: SYSTEM\_CPU\_INTR\_FROM\_CPU\_1\_REG (0x005C)

SYSTEM_CPU_INTR_FROM_CPU_1_REG (0x005C)																														
Bit Descriptions																														
31																														
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	0	0	0	0	0	0

**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 6.24: 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 6.25: 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 6.26: SYSTEM\_RSA\_PD\_CTRL\_REG (0x0068)

31					3	2	1	0
0	0	0	0	0	0	0	0	1

(reserved)

**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 6.27: SYSTEM\_BUSTOEXTMEM\_ENA\_REG (0x006C)

**SYSTEM\_BUSTOEXTMEM\_ENA** Set this bit to enable bus to EDMA. (R/W)

## Register 6.28: SYSTEM\_CACHE\_CONTROL\_REG (0x0070)

The diagram shows the bit field layout of the SYSTEM\_PRO\_DCACHE\_RESET register. It consists of a 32-bit register with the most significant bit (MSB) at the top. The bits are labeled from 31 to 0. Bit 31 is labeled "31". Bit 0 is labeled "(reserved)". Bits 30 to 1 are grouped under the label "Reset". Bits 31, 29, 28, and 27 are labeled "0". Bits 26 to 24 are labeled "1". Bits 23 to 21 are labeled "1". Bits 20 to 17 are labeled "0". Bits 16 to 14 are labeled "0". Bits 13 to 11 are labeled "0". Bits 10 to 8 are labeled "0". Bits 7 to 5 are labeled "0". Bits 4 to 2 are labeled "0". Bits 1 to 0 are labeled "0". To the right of the register, the bit names are listed diagonally: SYSTEM\_PRO\_DCACHE\_RESET, SYSTEM\_PRO\_ICACHE\_CLK\_ON, SYSTEM\_PRO\_DCACHE\_CLK\_ON, and SYSTEM\_PRO\_CACHE\_RESET.

**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 6.29: SYSTEM\_EXTERNAL\_DEVICE\_ENCRYPT\_DECRYPT\_CONTROL\_REG (0x0074)**

The diagram shows the bit field layout of Register 6.29. It consists of a 32-bit register with the following bit descriptions from left to right:

- Bit 31: (reserved)
- Bits 24-20: (reserved)
- Bit 19: SYSTEM\_ENABLE\_SPI\_MANUAL\_ENCRYPT
- Bit 18: SYSTEM\_ENABLE\_DOWNLOAD\_DB\_ENCRYPT
- Bit 17: SYSTEM\_ENABLE\_DOWNLOAD\_G0CB\_DECRYPT
- Bit 16: SYSTEM\_ENABLE\_DOWNLOAD\_MANUAL\_ENCRYPT
- Bit 15: (reserved)
- Bit 4: SYSTEM\_ENABLE\_SPL\_MANUAL\_ENCRYPT
- Bit 3: (reserved)
- Bit 2: (reserved)
- Bit 1: (reserved)
- Bit 0: Reset

**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 6.30: SYSTEM\_RTC\_FASTMEM\_CONFIG\_REG (0x0078)**

The diagram shows the bit field layout of Register 6.30. It consists of a 32-bit register with the following bit descriptions from left to right:

- Bit 31: SYSTEM\_RTC\_MEM\_CRC\_FINISH
- Bit 30: SYSTEM\_RTC\_MEM\_CRC\_LEN
- Bit 20: SYSTEM\_RTC\_MEM\_CRC\_ADDR
- Bit 19: SYSTEM\_RTC\_MEM\_CRC\_START
- Bit 9: (reserved)
- Bit 8: (reserved)
- Bit 7: (reserved)
- Bit 0: Reset

**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 6.31: SYSTEM\_RTC\_FASTMEM\_CRC\_REG (0x007C)**

		SYSTEM_RTC_MEM_CRC_RES	
31	0	0	Reset

**SYSTEM\_RTC\_MEM\_CRC\_RES** This field stores the CRC result of RTC memory. (RO)

**Register 6.32: SYSTEM\_SRAM\_CTRL\_2\_REG (0x0088)**

		SYSTEM_SRAM_FORCE_PU	
(reserved)		21	0
31	22		Reset
0 0 0 0 0 0 0 0 0 0		0x3fffff	

**SYSTEM\_SRAM\_FORCE\_PU** This field is used to power up internal SRAM. For details, please refer to Table 31. (R/W)

**Register 6.33: SYSTEM\_SYSCLK\_CONF\_REG (0x008C)**

		SYSTEM_CLK_XTAL_FREQ		SYSTEM_SOC_CLK_SEL		SYSTEM_PRE_DIV_CNT		0
(reserved)		18	19	12	11	10	9	0
31				0		0		0x1
0 0 0 0 0 0 0 0 0 0								Reset

**SYSTEM\_PRE\_DIV\_CNT** This field is used to set the count of prescaler of XTAL\_CLK. For details, please refer to Table 10 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 8 in Chapter 2 *Reset and Clock*. (R/W)

**SYSTEM\_CLK\_XTAL\_FREQ** This field is used to read XTAL frequency in MHz. (RO)

**Register 6.34: SYSTEM\_REG\_DATE\_REG (0x0FFC)**

				SYSTEM_REG_DATE
31	28	27		0
0	0	0	0	0x1908020 Reset

**SYSTEM\_DATE** Version control register. (R/W)

## 7. DMA Controller

### 7.1 Overview

Direct Memory Access (DMA) is a feature that allows peripheral-to-memory and memory-to-memory data transfer at a high speed. The CPU is not involved in the DMA transfer, and therefore it becomes more efficient.

ESP32-S2 has three types of DMA, namely Internal DMA, EDMA and Copy DMA. Internal DMA can only access internal RAM and is used for data transfer between internal RAM and peripherals. EDMA can access both internal RAM and external RAM and is used for data transfer between internal RAM, external RAM and peripherals. Copy DMA can only access internal RAM and is used for data transfer from one location in internal RAM to another.

Eight peripherals on ESP32-S2 have DMA features. As shown in figure 7-1, UART0 and UART1 share one Internal DMA; SPI3 and ADC Controller share one Internal DMA; AES Accelerator and SHA Accelerator share one EDMA; SPI2 and I<sup>2</sup>S0 have their individual EDMA. Besides, the CPU Peripheral module on ESP32-S2 also has one Copy DMA.

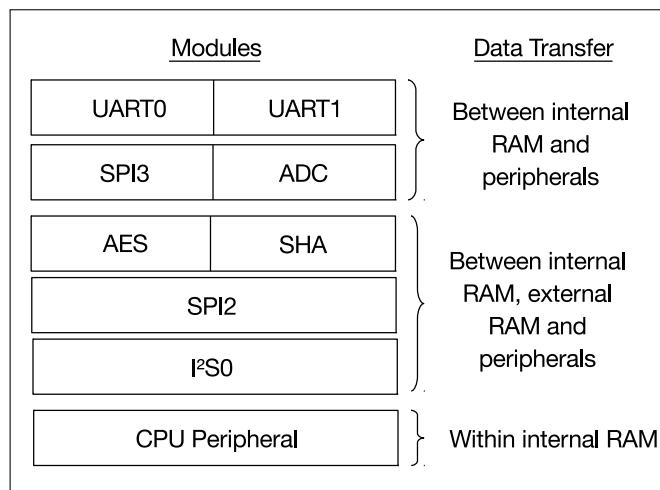


Figure 7-1. Modules with DMA and Supported Data Transfers

### 7.2 Features

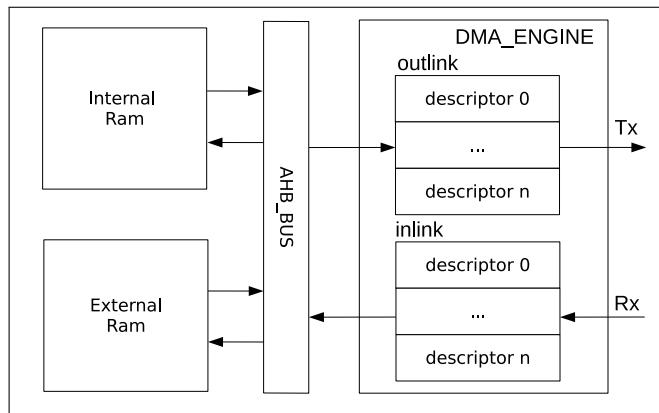
The DMA controller has the following features:

- AHB bus architecture
- Half-duplex and full-duplex mode
- Programmable length of data to be transferred in bytes
- INCR burst transfer when accessing internal RAM
- Access to an address space of 320 KB at most in internal RAM
- Access to an address space of 10.5 MB at most in external RAM
- High-speed data transfer using DMA

## 7.3 Functional Description

In ESP32-S2, all modules that need high-speed data transfer support DMA. The DMA controller and CPU data bus have access to the same address space in internal RAM and external RAM. DMA controllers for different modules vary in functions according to needs, but their architecture is identical.

### 7.3.1 DMA Engine Architecture

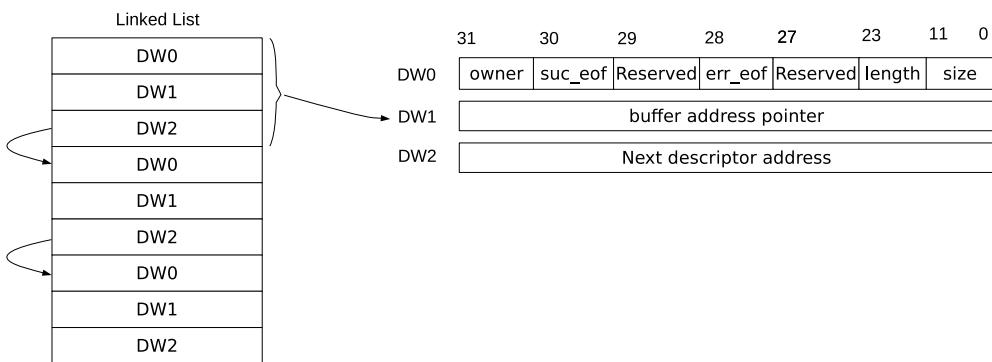


**Figure 7-2. DMA Engine Architecture**

A DMA engine reads/writes data to/from external RAM or internal RAM via the AHB\_BUS. Figure 7-2 shows the basic architecture of a DMA engine. For how to access RAM, please see Chapter 1 [System and Memory](#).

Software can use the DMA engine through linked lists. The DMA\_ENGINE transmits data in corresponding RAM according to the outlink (i.e. a linked list of transmit descriptors), and stores received data into specific address space in RAM according to the inlink (i.e. a linked list of receive descriptors).

### 7.3.2 Linked List



**Figure 7-3. Structure of a Linked List**

Figure 7-3 shows the structure of a linked list. An outlink and an inlink have the same structure. A linked list is formed by one or more descriptors, and each descriptor consists of three words. Linked lists should be in internal RAM for the DMA engine to be able to use them. The meaning of each field is as follows:

- Owner (DW0) [31]: Specifies who is allowed to access the buffer that this descriptor points to.
  - 1'b0: CPU can access the buffer;
  - 1'b1: The DMA controller can access the buffer.

When the DMA controller stops using the buffer, this bit is cleared by hardware. You can set *PERI\_IN\_LOOP\_TEST* bit to disable automatic clearing by hardware. When software loads a linked list, this bit should be set to 1.

**Note:** *PERI* refers to modules that support DMA transfers, e.g. I<sup>2</sup>S, SPI, UHCl, etc.

- **suc\_eof** (DW0) [30]: Specifies whether this descriptor is the last descriptor in the list.

1'b0: This descriptor is not the last one;

1'b1: This descriptor is the last one.

When a packet has been received, this bit in the last receive descriptor is set to 1 by hardware, and this bit in the last transmit descriptor is set to 1 by software.

- **Reserved** (DW0) [29]: Reserved.

- **err\_eof** (DW0) [28]: Specifies whether the received data has errors.

This bit is used only when UART DMA receives data. When an error is detected in the received packet, this bit in the receive descriptor is set to 1 by hardware.

- **Reserved** (DW0) [27:24]: Reserved.

- **Length** (DW0) [23:12]: Specifies the number of valid bytes in the buffer that this descriptor points to. This field in a transmit descriptor is written by software and indicates how many bytes can be read from the buffer; this field in a receive descriptor is written by hardware automatically and indicates how many bytes have been stored into the buffer.

When the DMA controller accesses external RAM, this field must be a multiple of 16/32/64 bytes. Please see more details in Section [7.3.6 Accessing External RAM](#).

- **Size** (DW0) [11:0]: Specifies the size of the buffer that this descriptor points to.

When the DMA controller accesses external RAM, this field must be a multiple of 16/32/64 bytes. Please see more details in Section [7.3.6 Accessing External RAM](#).

- **Buffer address pointer** (DW1): Pointer to the buffer.

When the DMA controller accesses external RAM, the destination address must be aligned with *PERI\_EXT\_MEM\_BK\_SIZE* field. Please see more details in Section [7.3.6 Accessing External RAM](#).

- **Next descriptor address** (DW2): Pointer to the next descriptor. If the current descriptor is the last one (**suc\_eof** = 1), this value is 0. This field can only point to internal RAM.

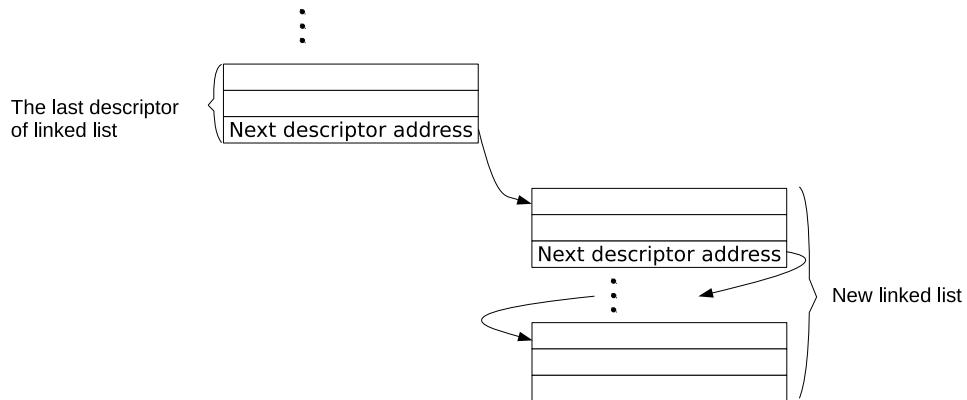
If the length of data received is smaller than the size of the buffer, the DMA controller will not use available space of the buffer in the next transaction.

### 7.3.3 Enabling DMA

Software uses the DMA controller through linked lists. When the DMA controller receives data, software loads an inlink, configures *PERI\_INLINK\_ADDR* field with address of the first receive descriptor, and sets *PERI\_INLINK\_START* bit to enable DMA. When the DMA controller transmits data, software loads an outlink, prepares data to be transmitted, configures *PERI\_OUTLINK\_ADDR* field with address of the first transmit descriptor, and sets *PERI\_OUTLINK\_START* bit to enable DMA. *PERI\_INLINK\_START* bit and *PERI\_OUTLINK\_START* bit are cleared automatically by hardware.

The DMA controller can be restarted. If you are not sure whether the loaded linked list has been used up or not and want to load a new linked list, you can use this Restart function without affecting the loaded linked list. When using the Restart function, software needs to rewrite address of the first descriptor in the new list to DW2 of the

last descriptor in the loaded list, loads the new list as shown in Figure 7-4, and set `PERI_INLINK_RESTART` bit or `PERI_OUTLINK_RESTART` bit (these two bits are cleared automatically by hardware). By doing so, hardware can obtain the address of the first descriptor in the new list when reading the last descriptor in the loaded list, and then read the new list.



**Figure 7-4.** Relationship among Linked Lists

### 7.3.4 Linked List reading process

Once configured and enabled by software, the DMA controller starts to read the linked list from internal RAM. *PERI\_IN\_DSCR\_ERR\_INT\_ENA* bit or *PERI\_OUT\_DSCR\_ERR\_INT\_ENA* bit can be set to enable descriptor error interrupt. If the buffer address pointer (DW1) does not point to 0x3FFB0000 ~ 0x3FFFFFF when the DMA controller accesses internal RAM, or does not point to 0x3F500000 ~ 0x3FF7FFFF when the DMA controller accesses external RAM, a descriptor error interrupt is generated.

**Note:** The third word (DW2) in a descriptor can only point to internal RAM; it points to the next descriptor to use and descriptors must be in internal memory.

### 7.3.5 EOF

The DMA controller uses EOF (end of file) flags to indicate the completion of data transfer.

Before the DMA controller transmits data, *PERI\_OUT\_TOTAL\_EOF\_INT\_ENA* bit should be set. If data in the buffer pointed by the last descriptor has been transmitted, a *PERI\_OUT\_TOTAL\_EOF\_INT* interrupt is generated.

Before the DMA controller receives data, *PERI\_IN\_SUC\_EOF\_INT\_ENA* bit should be set. If data has been received successfully, a *PERI\_IN\_SUC\_EOF\_INT* interrupt is generated. In addition to *PERI\_IN\_SUC\_EOF\_INT* interrupt, UART DMA also supports *UHCI\_IN\_ERR\_EOF\_INT*. This interrupt is enabled by setting *UHCI\_IN\_ERR\_EOF\_INT\_ENA* bit, and it indicates that a data packet has been received with errors.

When a *PERI\_OUT\_TOTAL\_EOF\_INT* or a *PERI\_IN\_SUC\_EOF\_INT* interrupt is detected, software can record the value of field *PERI\_OUTLINK\_DSCR\_ADDR* or *PERI\_INLINK\_DSCR\_ADDR*, i.e. address of the last descriptor (right shifted 2 bits). Therefore, software can tell which descriptors have been used and reclaim them.

### 7.3.6 Accessing External RAM

Of all DMA controllers for ESP32-S2 peripherals, only I<sup>2</sup>S0, SPI2, AES and SHA DMA have access to external RAM. The address space of external RAM that the DMA controller can access is 0x3E500000 ~ 0x3FF7FFFF.

Please note that destination addresses (in this case addresses for writing data to external RAM) must be 16-byte, 32-byte or 64-byte aligned. Table 35 illustrates the value of *PERI\_EXT\_MEM\_BK\_SIZE* bit when destination address is 16-byte, 32-byte and 64-byte aligned respectively.

**Note:** Source addresses (in this case, namely addresses for reading data from external RAM) do not need to be aligned.

**Table 35: Relationship Between Configuration Register and Destination Address**

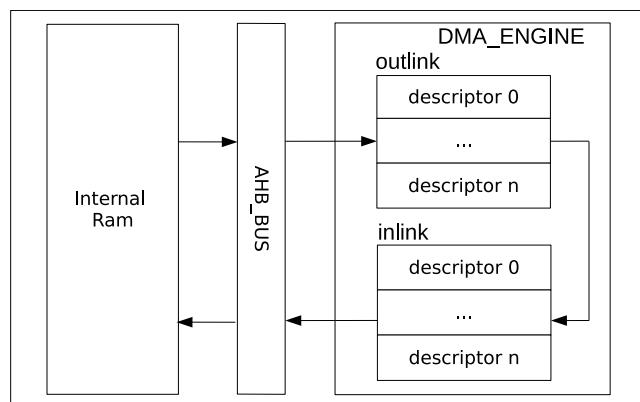
<i>PERI_EXT_MEM_BK_SIZE</i>	Destination address alignment
0	16-bit aligned
1	32-bit aligned
2	64-bit aligned

## 7.4 Copy DMA Controller

Copy DMA is used for data transfer from one location in internal RAM to another. Figure 7-5 shows the architecture of a Copy DMA engine. Unlike Internal DMA and EDMA, Copy DMA first reads data to be transferred from internal RAM, stores the data into the DMA FIFO via an outlink, and then writes the data to the target internal RAM via an inlink.

Copy DMA should be configured by software as follows:

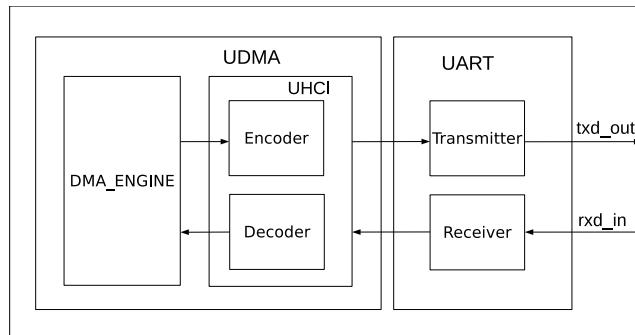
1. Set CP\_DMA\_IN\_RST, CP\_DMA\_OUT\_RST, CP\_DMA\_FIFO\_RST and CP\_DMA\_CMD FIFO\_RST bit first to 1 and then to 0, to reset Copy DMA state machine and FIFO pointer;
2. Set CP\_DMA\_FIFO\_RST bit first to 1 and then to 0, to reset FIFO pointer;
3. Load an outlink, and configure CP\_DMA\_OUTLINK\_ADDR with address of the first transmit descriptor;
4. Load an inlink, and configure CP\_DMA\_INLINK\_ADDR with address of the first receive descriptor;
5. Set CP\_DMA\_OUTLINK\_START to enable DMA transmission;
6. Set CP\_DMA\_INLINK\_START to enable DMA reception.



**Figure 7-5. Copy DMA Engine Architecture**

## 7.5 UART DMA (UDMA) Controller

ESP32-S2 has two UART controllers. They share one UDMA controller. UHCI\_UART\_CE specifies which UART controller gets access to UDMA.



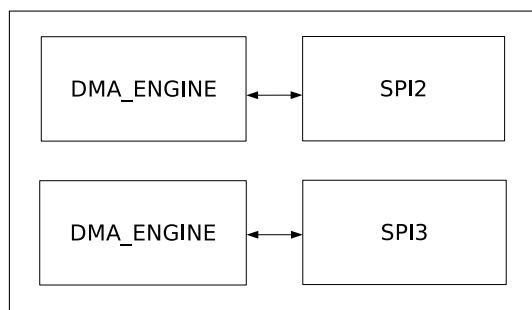
**Figure 7-6. Data Transfer in UDMA Mode**

Figure 7-6 shows how data is transferred using UDMA. Before UDMA receives data, software prepares an inlink. [UHCI\\_INLINK\\_ADDR](#) points to the first receive descriptor in the inlink. After [UHCI\\_INLINK\\_START](#) is set, UHCI sends data that UART has received to the Decoder. The decoded data is then stored into the RAM pointed by the inlink under the control of UDMA.

Before UDMA sends data, software prepares an outlink and data to be sent. [UHCI\\_OUTLINK\\_ADDR](#) points to the first transmit descriptor in the outlink. After [UHCI\\_OUTLINK\\_START](#) is set, UDMA reads data from the RAM pointed by outlink. The data is then encoded by the Encoder, and sent sequentially by the UART transmitter.

Data packets of UDMA have separators at the beginning and the end, with data bits in the middle. The encoder inserts separators in front of and after data bits, and replaces data bits identical to separators with special characters. The decoder removes separators in front of and after data bits, and replaces special characters with separators. There can be more than one continuous separator at the beginning and the end of a data packet. The separator is configured by [UHCI\\_SEPER\\_CHAR](#), 0xC0 by default. The special character is configured by [UHCI\\_ESC\\_SEQ0\\_CHAR0](#) (0xDB by default) and [UHCI\\_ESC\\_SEQ0\\_CHAR1](#) (0xDD by default). When all data has been sent, a [UHCI\\_OUT\\_TOTAL\\_EOF\\_INT](#) interrupt is generated. When all data has been received, a [UHCI\\_IN\\_SUC\\_EOF\\_INT](#) is generated.

## 7.6 SPI DMA Controller



**Figure 7-7. SPI DMA**

As shown in Figure 7-7, SPI2 and SPI3 have separate DMA controllers.

SPI DMA receives and transmits data through descriptors at least one byte at a time. The transmission of data can be done in bursts.

SPI\_OUTLINK\_START bit of SPI\_DMA\_OUT\_LINK\_REG register and SPI\_INLINK\_START bit of SPI\_DMA\_IN\_LINK\_REG register are used to enable the DMA engine and are cleared by hardware. When SPI\_OUTLINK\_START is set to 1, the DMA engine loads an outlink and prepares data to be transferred; when SPI\_INLINK\_START is set to 1, the DMA engine loads an inlink and prepares to receive data.

When receiving data, SPI DMA should be configured by software as follows:

1. Set SPI\_DMA\_IN\_RST, SPI\_AHBM\_FIFO\_RST and SPI\_AHBM\_RST bit first to 1 and then to 0, to reset DMA state machine and FIFO pointer;
2. Load an inlink, and configure SPI\_DMA\_INLINK\_ADDR with address of the first receive descriptor;
3. Set SPI\_DMA\_INLINK\_START to enable DMA reception.

When transmitting data, SPI DMA should be configured by software as follows:

1. Set SPI\_DMA\_OUT\_RST, SPI\_AHBM\_FIFO\_RST and SPI\_AHBM\_RST bit first to 1 and then to 0, to reset DMA state machine and FIFO pointer;
2. Load an outlink, and configure SPI\_DMA\_OUTLINK\_ADDR with address of the first transmit descriptor;
3. Set SPI\_DMA\_OUTLINK\_START to enable DMA transmission.

**Note:** When SPI DMA transfers data between internal RAM and external RAM, SPI\_MEM\_TRANS\_EN should be set.

SPI DMA also supports data transfer in segments.

## 7.7 I<sup>2</sup>S DMA Controller

ESP32-S2 I<sup>2</sup>S has an individual DMA. I2S\_DSCR\_EN bit of I2S\_FIFO\_CONF\_REG register is used to enable DMA transfer of I<sup>2</sup>S. I<sup>2</sup>S DMA receives and transmits data through linked lists. The transmission of data can be done in bursts. I2S\_RX\_EOF\_NUM[31:0] bit of I2S\_RXEOF\_NUM\_REG register is used to configure how many words of data to be received at a time.

I2S\_OUTLINK\_START bit of I2S\_OUT\_LINK\_REG and I2S\_INLINK\_START bit of I2S\_IN\_LINK\_REG register are used to enable the DMA engine and are cleared by hardware. When I2S\_OUTLINK\_START bit is set to 1, the DMA engine loads an outlink and prepares data to be transferred; when I2S\_INLINK\_START is set to 1, the DMA engine loads an inlink and prepares to receive data.

When receiving data, I<sup>2</sup>S DMA should be configured by software as follows:

1. Set I2S\_IN\_RST, I2S\_AHBM\_FIFO\_RST and I2S\_AHBM\_RST bit first to 1 and then to 0, to reset DMA state machine and FIFO pointer;
2. Load an inlink, and configure I2S\_INLINK\_ADDR with address of the first receive descriptor;
3. Set I2S\_INLINK\_START to enable DMA reception.

When transmitting data, I<sup>2</sup>S DMA should be configured by software as follows:

1. Set I2S\_OUT\_RST, I2S\_AHBM\_FIFO\_RST and I2S\_AHBM\_RST bit first to 1 and then to 0, to reset DMA state machine and FIFO pointer;

2. Load an outlink, and configure I2S\_OUTLINK\_ADDR with address of the first transmit descriptor;
3. Set I2S\_OUTLINK\_START to enable DMA transmission.

**Note:** When I<sup>2</sup>S DMA transfers data between internal RAM and external RAM using I<sup>2</sup>S DMA, I2S\_MEM\_TRANS\_EN should be set.

## 8. UART Controller

### 8.1 Overview

In embedded system applications, data is required to be transferred in a simple way with minimal system resources. This can be achieved by a Universal Asynchronous Receiver/Transmitter (UART), which flexibly exchanges data with other peripheral devices in full duplex mode. ESP32-S2 has two UART controllers compatible with various UART devices. They support Infrared Data Association (IrDA) and RS-485 transmission.

ESP32-S2 has two UART controllers. Each has a group of registers that function identically. In this chapter, the two UART controllers are referred to as `UART $n$` , in which  $n$  denotes 0 or 1.

### 8.2 Features

Each UART controller has the following features:

- Programmable baud rate
- 512 × 8-bit RAM shared by TX FIFOs and RX FIFOs of two UART controllers
- Full duplex asynchronous communication
- Automatic baud rate detection
- Data bits ranging from 5 to 8
- Stop bits whose length can be 1, 1.5, 2 or 3 bits
- Parity bits
- Special character AT\_CMD detection
- RS-485 protocol
- IrDA protocol
- High-speed data communication using DMA
- UART as wake-up source
- Software and hardware flow control

### 8.3 Functional Description

#### 8.3.1 UART Introduction

A UART is a character-oriented data link for asynchronous communication between devices. Such communication does not add clock signals to data sent. Therefore, in order to communicate successfully, the transmitter and the receiver must operate at the same baud rate with the same stop bit and parity bit.

A UART data packet usually begins with one start bit, followed by data bits, one parity bit (optional) and one or more stop bits. UART controllers on ESP32-S2 support various lengths of data bits and stop bits. These

controllers also support software and hardware flow control as well as DMA for seamless high-speed data transfer. This allows developers to use multiple UART ports at minimal software cost.

### 8.3.2 UART Structure

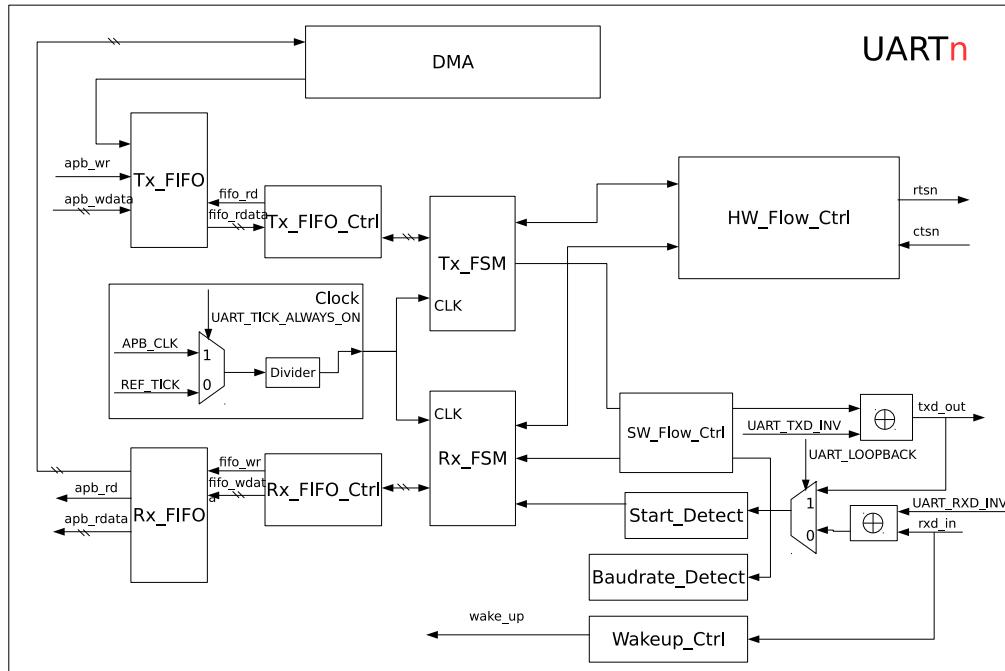


Figure 8-1. UART Structure

Figure 8-1 shows the basic structure of a UART controller. It has two possible clock sources: a 80 MHz APB\_CLK and a reference clock REF\_TICK (for details, please refer to Chapter 2 Reset and Clock), which are selected by configuring [UART\\_TICK\\_REF\\_ALWAYS\\_ON](#). The selected clock source is divided by a divider to generate clock signals that drive the UART controller. The divisor is configured by [UART\\_CLKDIV\\_REG](#): [UART\\_CLKDIV](#) for the integral part, and [UART\\_CLKDIV\\_FRAG](#) for the fractional part.

A UART controller is broken down into two parts according to functions: a transmitter and a receiver.

The transmitter contains a TX FIFO, which buffers data to be sent. Software can write data to Tx\_FIFO via the APB bus, or move data to Tx\_FIFO using DMA. Tx\_FIFO\_Ctrl controls writing and reading Tx\_FIFO. When Tx\_FIFO is not empty, Tx\_FSM reads bytes via Tx\_FIFO\_Ctrl, and converts them into a bitstream. The levels of output signal txd\_out can be inverted by configuring [UART\\_TXD\\_INV](#) register.

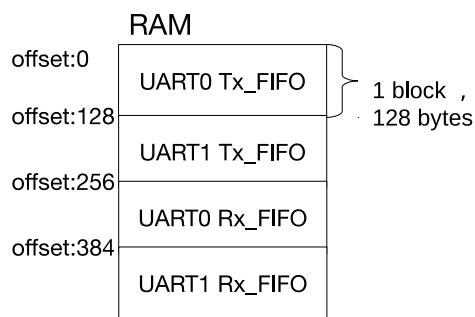
The receiver contains a RX FIFO, which buffers data to be processed. Software can read data from Rx\_FIFO via the APB bus, or receive data using DMA. The levels of input signal rxd\_in can be inverted by configuring [UART\\_RXD\\_INV](#) register, and the signal is then input to the Rx components of the UART Controller:

Baudrate\_Detect measures the baud rate of input signal rxd\_in by detecting its minimum pulse width.

Start\_Detect detects the start bit in a data frame. If the start bit is detected, Rx\_FSM stores data bits in the data frame into Rx\_FIFO by Rx\_FIFO\_Ctrl.

HW\_Flow\_Ctrl controls rxd\_in and txd\_out data flows by standard UART RTS and CTS flow control signals (rtsn\_out and ctsn\_in). SW\_Flow\_Ctrl controls data flows by automatically adding special characters to outgoing data and detecting special characters in incoming data. When a UART controller is in the Light-sleep mode, Wakeup\_Ctrl counts up rising edges of rxd\_in. When the number reaches ([UART\\_ACTIVE\\_THRESHOLD](#) + 2), a wake\_up signal is generated and sent to RTC, which then wakes up the ESP32-S2 chip.

### 8.3.3 UART RAM



**Figure 8-2. UART Controllers Sharing RAM**

The two UART controllers on ESP32-S2 share  $512 \times 8$  bits of FIFO RAM. As figure 8-2 illustrates, RAM is divided into 4 blocks, each has  $128 \times 8$  bits. Figure 8-2 shows by default how many RAM blocks are allocated to TX FIFOs and RX FIFOs of the two UART controllers.  $\text{UART}_n \text{ Tx\_FIFO}$  can be expanded by configuring [UART\\_TX\\_SIZE](#), while  $\text{UART}_n \text{ Rx\_FIFO}$  can be expanded by configuring [UART\\_RX\\_SIZE](#). The size of UART0 Tx\_FIFO can be increased to 4 blocks (the whole RAM), the size of UART1 Tx\_FIFO can be increased to 3 blocks (from offset 128 to the end address), the size of UART0 Rx\_FIFO can be increased to 2 blocks (from offset 256 to the end address), but the size of UART1 Rx\_FIFO cannot be increased. Please note that expanding one FIFO may take up the default space of other FIFOs. For example, by setting [UART\\_TX\\_SIZE](#) of UART0 to 2, the size of UART0 Tx\_FIFO is increased by 128 bytes (from offset 0 to offset 255). In this case, UART0 Tx\_FIFO takes up the default space for UART1 Tx\_FIFO, and UART1's transmitting function cannot be used as a result.

When neither of the two UART controllers is active, RAM could enter low-power mode by setting [UART\\_MEM\\_FORCE\\_PD](#).

UART0 Tx\_FIFO and UART1 Tx\_FIFO are reset by setting [UART\\_TXFIFO\\_RST](#). UART0 Rx\_FIFO and UART1 Rx\_FIFO are reset by setting [UART\\_RXFIFO\\_RST](#).

Data to be sent is written to TX FIFO via the APB bus or using DMA, read automatically and converted from a frame into a bitstream by hardware Tx\_FSM; data received is converted from a bitstream into a frame by hardware Rx\_FSM, written into RX FIFO, and then stored into RAM via the APB bus or using DMA. The two UART controllers share one DMA controller.

The empty signal threshold for Tx\_FIFO is configured by setting [UART\\_TXFIFO\\_EMPTY\\_THRESHOLD](#). When data stored in Tx\_FIFO is less than [UART\\_TXFIFO\\_EMPTY\\_THRESHOLD](#), a [UART\\_TXFIFO\\_EMPTY\\_INTERRUPT](#) interrupt is generated.

The full signal threshold for Rx\_FIFO is configured by setting [UART\\_RXFIFO\\_FULL\\_THRESHOLD](#). When data stored in Rx\_FIFO is greater than [UART\\_RXFIFO\\_FULL\\_THRESHOLD](#), a [UART\\_RXFIFO\\_FULL\\_INTERRUPT](#) interrupt is generated. In addition, when Rx\_FIFO receives more data than its capacity, a [UART\\_RXFIFO\\_OVF\\_INTERRUPT](#) interrupt is generated.

### 8.3.4 Baud Rate Generation and Detection

#### 8.3.4.1 Baud Rate Generation

Before a UART controller sends or receives data, the baud rate should be configured by setting corresponding registers. A UART Controller baud rate generator functions by dividing the input clock source. It can divide the

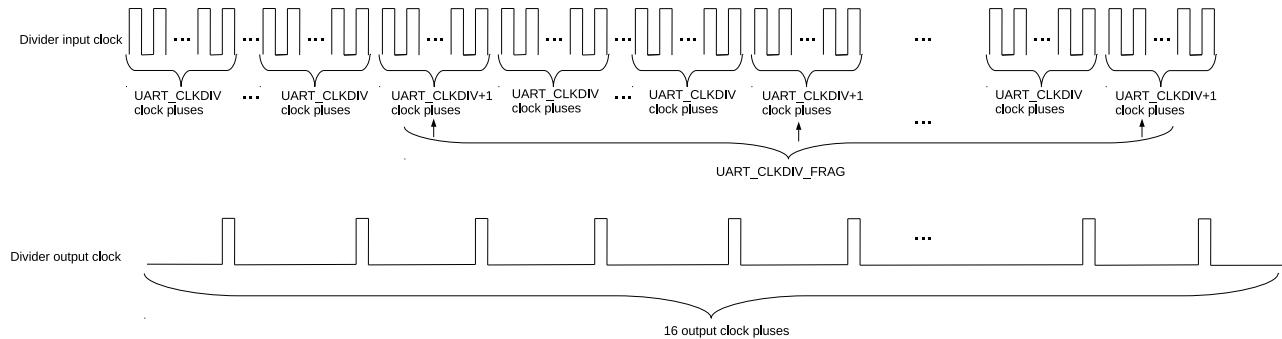
clock source by a fractional amount. The divisor is configured by `UART_CLKDIV_REG`: `UART_CLKDIV` for the integral part, and `UART_CLKDIV_FRAG` for the fractional part. When using an 80 MHz input clock, the UART controller supports a maximum baud rate of 5 Mbaud.

The divisor of the baud rate divider is equal to `UART_CLKDIV` + (`UART_CLKDIV_FRAG` / 16), meaning that the final baud rate is equal to `INPUT_FREQ` / (`UART_CLKDIV` + (`UART_CLKDIV_FRAG` / 16)). For example, if `UART_CLKDIV` = 694 and `UART_CLKDIV_FRAG` = 7 then the divisor value is  $(694 + 7/16) = 694.4375$ . If the input clock frequency is 80MHz APB\_CLK, the baud rate will be  $(80\text{MHz} / 694.4375) = 115201$ .

When `UART_CLKDIV_FRAG` is zero, the baud rate generator is an integer clock divider where an output pulse is generated every `UART_CLKDIV` input pulses.

When `UART_CLKDIV_FRAG` is not zero, the divider is fractional and the output baud rate clock pulses are not strictly uniform. As shown in figure 8-3, for every 16 output pulses, the generator divides either (`UART_CLKDIV` + 1) input pulses or `UART_CLKDIV` input pulses per output pulse. A total of `UART_CLKDIV_FRAG` output pulses are generated by dividing (`UART_CLKDIV` + 1) input pulses, and the remaining  $(16 - \text{UART\_CLKDIV\_FRAG})$  output pulses are generated by dividing `UART_CLKDIV` input pulses.

The output pulses are interleaved as shown in figure 8-3 below, to make the output timing more uniform:



**Figure 8-3. UART Controllers Division**

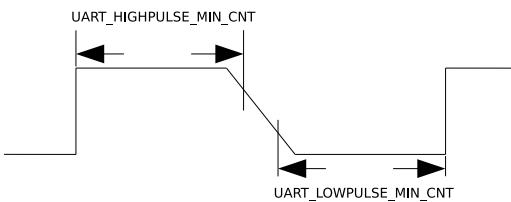
To support IrDA (see Section 8.3.7 [IrDA](#) for details), the fractional clock divider for IrDA data transmission generates clock signals divided by  $16 \times \text{UART\_CLKDIV\_REG}$ . This divider works similarly as the one elaborated above: it takes `UART_CLKDIV`/16 as the integer value and the lowest four bits of `UART_CLKDIV` as the fractional value.

### 8.3.4.2 Baud Rate Detection

Automatic baud rate detection (Autobaud) on UARTs is enabled by setting `UART_AUTOBAUD_EN`. The Baudrate\_Detect module shown in figure 8-1 will measure pulse widths while filtering any noise whose pulse width is shorter than `UART_GLITCH_FILTER`.

Before communication starts, the transmitter could send random data to the receiver for baud rate detection. `UART_LOWPULSE_MIN_CNT` stores the minimum low pulse width, `UART_HIGHPULSE_MIN_CNT` stores the minimum high pulse width, `UART_POSEDGE_MIN_CNT` stores the minimum pulse width between two rising edges, and `UART_NEGEDGE_MIN_CNT` stores the minimum pulse width between two falling edges. These four registers are read by software to determine the transmitter's baud rate.

Baud rate can be determined in the following three ways:



**Figure 8-4. The Timing Diagram of Weak UART Signals Along Falling Edges**

1. Normally, to avoid sampling erroneous data along rising or falling edges in semi-stable state, which results in inaccuracy of [UART\\_LOWPULSE\\_MIN\\_CNT](#) or [UART\\_HIGHPULSE\\_MIN\\_CNT](#), use a weighted average of these two values to eliminate errors. In this case, baud rate is calculated as follows:

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{(\text{UART\_LOWPULSE\_MIN\_CNT} + \text{UART\_HIGHPULSE\_MIN\_CNT})/2}$$

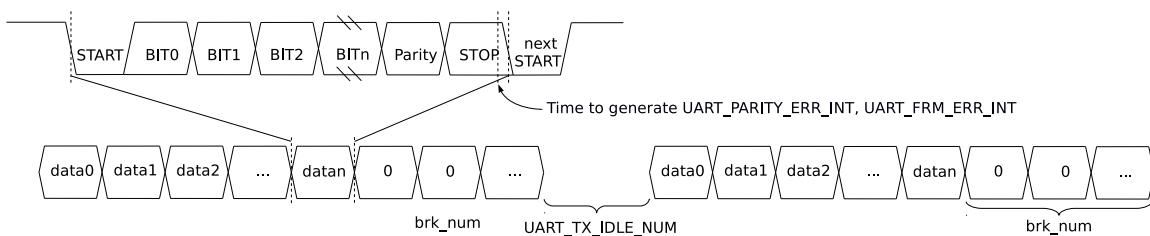
2. If UART signals are weak along falling edges as shown in figure 8-4, which leads to inaccurate average of [UART\\_LOWPULSE\\_MIN\\_CNT](#) and [UART\\_HIGHPULSE\\_MIN\\_CNT](#), use [UART\\_POSEDGE\\_MIN\\_CNT](#) to determine the transmitter's baud rate as follows:

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{\text{UART\_POSEDGE\_MIN\_CNT}/2}$$

3. If UART signals are weak along rising edges, use [UART\\_NEGEDGE\\_MIN\\_CNT](#) to determine the transmitter's baud rate as follows:

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{\text{UART\_NEGEDGE\_MIN\_CNT}/2}$$

### 8.3.5 UART Data Frame



**Figure 8-5. Structure of UART Data Frame**

Figure 8-5 shows the basic structure of a data frame. A frame starts with one START bit, and ends with STOP bits which can be 1, 1.5, 2 or 3 bits long, configured by [UART\\_STOP\\_BIT\\_NUM](#), [UART\\_DL1\\_EN](#) and [UART\\_DL0\\_EN](#). The START bit is logical low, whereas STOP bits are logical high.

The actual data length can be anywhere between 5 ~ 8 bit, configured by [UART\\_BIT\\_NUM](#). When [UART\\_PARITY\\_EN](#) is set, a parity bit is added after data bits. [UART\\_PARITY](#) is used to choose even parity or odd parity. When the receiver detects a parity bit error in data received, a [UART\\_PARITY\\_ERR\\_INT](#) interrupt is generated and data received is still stored into RX FIFO. When the receiver detects a data frame error, a [UART\\_FRM\\_ERR\\_INT](#) interrupt is generated, and data received by default is stored into RX FIFO.

If all data in Tx\_FIFO has been sent, a [UART\\_TX\\_DONE\\_INT](#) interrupt is generated. After this, if the [UART\\_TXD\\_BRK](#) bit is set then the transmitter will send several NULL characters in which the TX data line is logical low. The number of NULL characters is configured by [UART\\_TX\\_BRK\\_NUM](#). Once the transmitter has sent all NULL characters, a [UART\\_TX\\_BRK\\_DONE\\_INT](#) interrupt is generated. The minimum interval between

data frames can be configured using `UART_TX_IDLE_NUM`. If the transmitter stays idle for `UART_TX_IDLE_NUM` or more time, a `UART_TX_BRK_IDLE_DONE_INT` interrupt is generated.

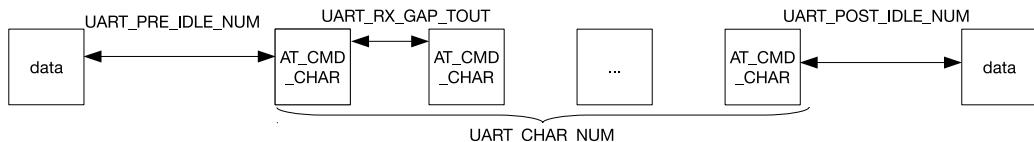


Figure 8-6. AT\_CMD Character Structure

Figure 8-6 is the structure of a special character AT\_CMD. If the receiver constantly receives AT\_CMD\_CHAR and the following conditions are met, a UART\_AT\_CMD\_CHAR\_DET\_INT interrupt is generated.

- The interval between the first AT\_CMD\_CHAR and the last non AT\_CMD\_CHAR character is at least `UART_PRE_IDLE_NUM` single-bit cycles.
  - The interval between two AT\_CMD\_CHAR characters is less than `UART_RX_GAP_TOUT` single-bit cycles.
  - The number of AT\_CMD\_CHAR characters is equal to or greater than `UART_CHAR_NUM`.
  - The interval between the last AT\_CMD\_CHAR character and next non AT\_CMD\_CHAR character is at least `UART_POST_IDLE_NUM` single-bit cycles.

### 8.3.6 RS485

The two UART controllers support RS485 standard. This standard uses differential signals to transmit data, so it can communicate over longer distances at higher bit rates than RS232. RS485 has two-wire half-duplex mode and four-wire full-duplex mode. UART controllers support two-wire half-duplex transmission and bus snooping. In a two-wire RS485 multidrop network, there can be 32 slaves at most.

### 8.3.6.1 Driver Control

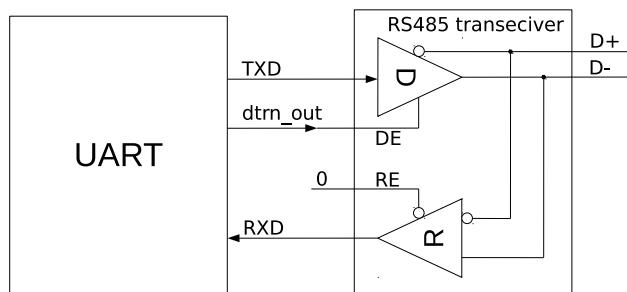
As shown in figure 8-7, in a two-wire multidrop network, an external RS485 transceiver is needed for differential to single-ended conversion. A RS485 transceiver contains a driver and a receiver. When a UART controller is not in transmitter mode, the connection to the differential line can be broken by disabling the driver. When DE is 1, the driver is enabled; when DE is 0, the driver is disabled.

The receiving UART converts differential signals to single-ended signals via an external receiver. RE is the enable control signal for the receiver. When RE is 0, the receiver is enabled; when RE is 1, the receiver is disabled. If RE is configured as 0, the UART controller is allowed to snoop data on the bus, including data sent by itself.

DE can be controlled by either software or hardware. To reduce cost of software, in our design DE is controlled by hardware. As shown in figure 8-7, DE is connected to dtrn\_out of UART (please refer to Section [8.3.9.1 Hardware Flow Control](#) for more details).

### 8.3.6.2 Turnaround Delay

By default, the two UART controllers work in receiver mode. When a UART controller is switched from transmitter mode to receiver mode, the RS485 protocol requires a turnaround delay of at least one cycle after the stop bit.



**Figure 8-7. Driver Control Diagram in RS485 Mode**

The transmitter supports turnaround delay of two cycles added after the stop bit. When [UART\\_DL1\\_EN](#) is set, turnaround delay of one single-bit cycle is added; when [UART\\_DL0\\_EN](#) is set, turnaround delay of a second cycle is added.

### 8.3.6.3 Bus Snooping

By default, an RS485 device is not allowed to transmit and receive data simultaneously. However, the UART controller peripheral supports snooping this bus by receiving while transmitting. If [UART\\_RS485TX\\_RX\\_EN](#) is set and the external RS485 transceiver is configured as in figure 8-7, a UART controller may receive data in transmitter mode and snoop the bus. If [UART\\_RS485RXBY\\_TX\\_EN](#) is set, a UART controller may transmit data in receiver mode.

The two UART controllers can snoop data sent by themselves. In transmitter mode, when a UART controller monitors a collision between data sent and data received, a [UART\\_RS485\\_CLASH\\_INT](#) is generated; when a UART controller monitors a data frame error, a [UART\\_RS485\\_FRM\\_ERR\\_INT](#) interrupt is generated; when a UART controller monitors a polarity error, a [UART\\_RS485\\_PARITY\\_ERR\\_INT](#) is generated.

### 8.3.7 IrDA

IrDA protocol consists of three layers, namely the physical layer, the link access protocol and the link management protocol. The two UART controllers implement IrDA physical layer. In IrDA encoding, a UART controller supports data rates up to 115.2 kbit/s (SIR, or serial infrared mode). As shown in figure 8-8, the IrDA encoder converts a NRZ (non-return to zero code) signal to a RZI (return to zero code) signal and sends it to the external driver and infrared LED. This encoder uses modulated signals whose pulse width is 3/16 bits to indicate logic “0”, and low levels to indicate logic “1”. The IrDA decoder receives signals from the infrared receiver and converts them to NRZ signals. In most cases, the receiver is high when it is idle, and the encoder output polarity is the opposite of the decoder input polarity. If a low pulse is detected, it indicates that a start bit has been received.

When IrDA function is enabled, one bit is divided into 16 clock cycles. If the bit to be sent is zero, then the 9th, 10th and 11th clock cycle is high.

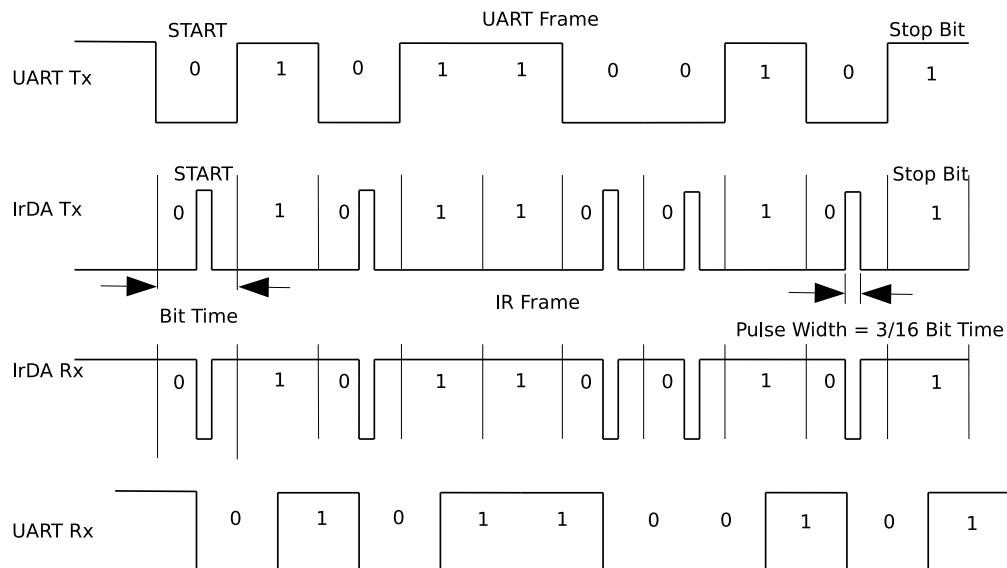


Figure 8-8. The Timing Diagram of Encoding and Decoding in SIR mode

The IrDA transceiver is half-duplex, meaning that it cannot send and receive data simultaneously. As shown in figure 8-9, IrDA function is enabled by setting `UART_IRDA_EN`. When `UART_IRDA_TX_EN` is set (high), the IrDA transceiver is enabled to send data and not allowed to receive data; when `UART_IRDA_TX_EN` is reset (low), the IrDA transceiver is enabled to receive data and not allowed to send data.

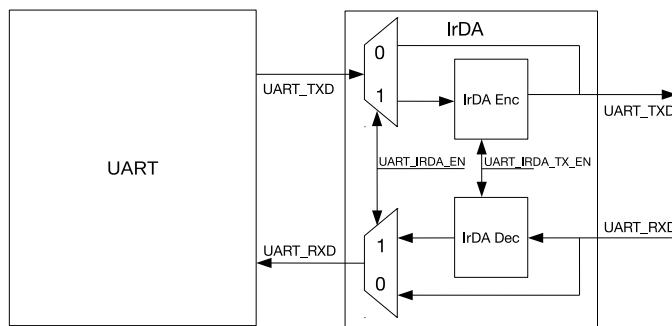


Figure 8-9. IrDA Encoding and Decoding Diagram

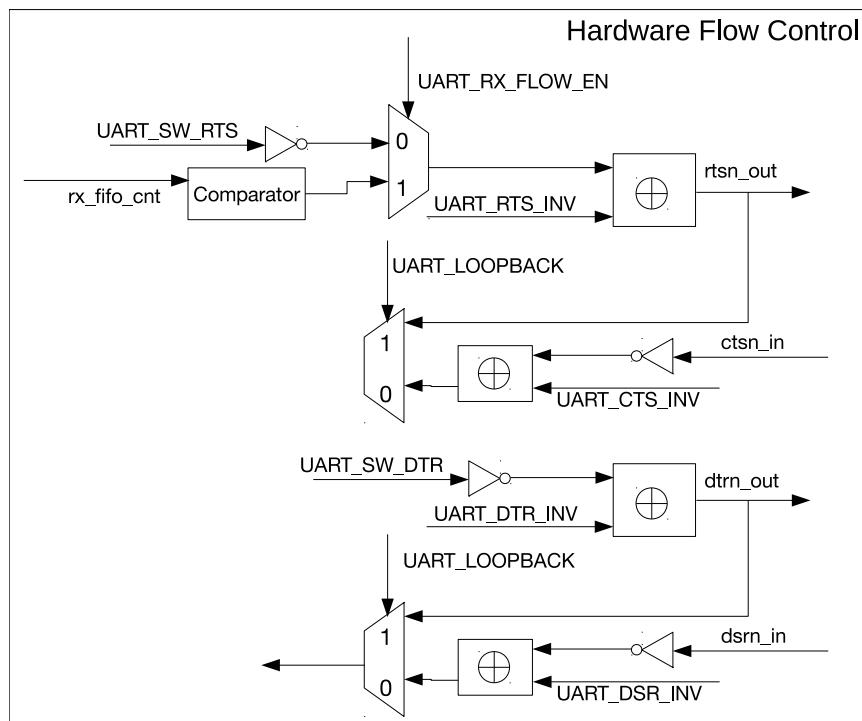
### 8.3.8 Wake-up

UART0 and UART1 can be set as wake-up source. When a UART controller is in Light-sleep mode, `Wakeup_Ctrl` counts up the rising edges of `rxn_in`. When the number of rising edges is greater than `(UART_ACTIVE_THRESHOLD + 2)`, a `wake_up` signal is generated and sent to RTC, which then wakes up ESP32-S2.

### 8.3.9 Flow Control

UART controllers have two ways to control data flow, namely hardware flow control and software flow control. Hardware flow control is achieved using output signal `rtn_out` and input signal `dsrn_in`. Software flow control is achieved by inserting special characters in data flow sent and detecting special characters in data flow received.

### 8.3.9.1 Hardware Flow Control

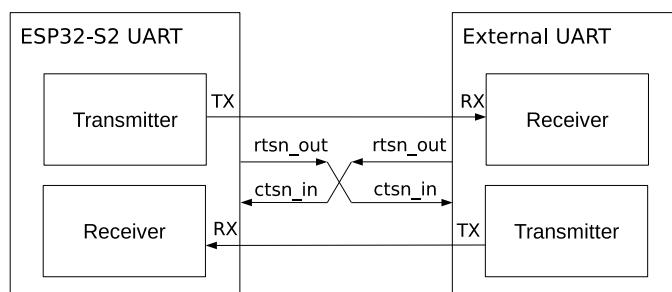


**Figure 8-10. Hardware Flow Control Diagram**

Figure 8-10 shows hardware flow control of a UART controller. Hardware flow control uses output signal `rtsn_out` and input signal `dsrn_in`. Figure 8-11 illustrates how these signals are connected between ESP32-S2 UART (hereinafter referred to as IU0) and the external UART (hereinafter referred to as EU0).

When `rtsn_out` of IU0 is low, EU0 is allowed to send data; when `rtsn_out` of IU0 is high, EU0 is notified to stop sending data until `rtsn_out` of IU0 returns to low. Output signal `rtsn_out` can be controlled in two ways.

- Software control: Enter this mode by setting `UART_RX_FLOW_EN` to 0. In this mode, the level of `rtsn_out` is changed by configuring `UART_SW_RTS`.
- Hardware control: Enter this mode by setting `UART_RX_FLOW_EN` to 1. In this mode, `rtsn_out` is pulled high when data in Rx\_FIFO exceeds `UART_RX_FLOW_THRHD`.



**Figure 8-11. Connection between Hardware Flow Control Signals**

When `ctsn_in` of IU0 is low, IU0 is allowed to send data; when `ctsn_in` is high, IU0 is not allowed to send data. When IU0 detects an edge change of `ctsn_in`, a `UART_CTS_CHG_INT` interrupt is generated.

If `dtrn_out` of IU0 is high, it indicates that IU0 is ready to transmit data. `dtrn_out` is generated by configuring register `UART_SW_DTR`. When the IU0 transmitter detects a edge change of `dsrn_in`, a `UART_DSR_CHG_INT`

interrupt is generated. After this interrupt is detected, software can obtain the level of input signal dsrn\_in by reading [UART\\_DSRN](#). If dsrn\_in is high, it indicates that EU0 is ready to transmit data.

In a two-wire RS485 multidrop network enabled by setting [UART\\_RS485\\_EN](#), dtrn\_out is used for transmit/receive turnaround. In this case, dtrn\_out is generated by hardware. When data transmission starts, dtrn\_out is pulled high and the external driver is enabled; when data transmission completes, dtrn\_out is pulled low and the external driver is disabled. Please note that when there is turnaround delay of one cycle added after the stop bit, dtrn\_out is pulled low after the delay.

UART loopback test is enabled by setting [UART\\_LOOPBACK](#). In the test, UART output signal txd\_out is connected to its input signal rxd\_in, rtsn\_out is connected to ctsn\_in, and dtrn\_out is connected to dsrn\_out. If data sent matches data received, it indicates that UART controllers are working properly.

### 8.3.9.2 Software Flow Control

Instead of CTS/RTS lines, software flow control uses XON/XOFF characters to start or stop data transmission. Such flow control is enabled by setting [UART\\_SW\\_FLOW\\_CON\\_EN](#) to 1.

When using software flow control, hardware automatically detects if there are XON/XOFF characters in data flow received, and generate a [UART\\_SW\\_XOFF\\_INT](#) or a [UART\\_SW\\_XON\\_INT](#) interrupt accordingly. If an XOFF character is detected, the transmitter stops data transmission once the current byte has been transmitted; if an XON character is detected, the transmitter starts data transmission. In addition, software can force the transmitter to stop sending data by setting [UART\\_FORCE\\_XOFF](#), or to start sending data by setting [UART\\_FORCE\\_XON](#).

Software determines whether to insert flow control characters according to the remaining room in RX FIFO. When [UART\\_SEND\\_XOFF](#) is set, the transmitter sends an XOFF character configured by [UART\\_XOFF\\_CHAR](#) after the current byte in transmission; when [UART\\_SEND\\_XON](#) is set, the transmitter sends an XON character configured by [UART\\_XON\\_CHAR](#) after the current byte in transmission. If the RX FIFO of a UART controller stores more data than [UART\\_XOFF\\_THRESHOLD](#), [UART\\_SEND\\_XOFF](#) is set by hardware. As a result, the transmitter sends an XOFF character after the current byte in transmission. If the RX FIFO of a UART controller stores less data than [UART\\_XON\\_THRESHOLD](#), [UART\\_SEND\\_XON](#) is set by hardware. As a result, the transmitter sends an XON character after the current byte in transmission.

### 8.3.10 UDMA

The two UART controllers on ESP32-S2 share one UDMA (UART DMA), which supports the decoding and encoding of HCI data packets. For more information, please refer to Chapter 7: [DMA Controller](#).

### 8.3.11 UART Interrupts

- [UART\\_AT\\_CMD\\_CHAR\\_DET\\_INT](#): Triggered when the receiver detects an AT\_CMD character.
- [UART\\_RS485\\_CLASH\\_INT](#): Triggered when a collision is detected between the transmitter and the receiver in RS485 mode.
- [UART\\_RS485\\_FRM\\_ERR\\_INT](#): Triggered when an error is detected in the data frame sent by the transmitter in RS485 mode.

- `UART_RS485_PARITY_ERR_INT`: Triggered when an error is detected in the parity bit sent by the transmitter in RS485 mode.
- `UART_TX_DONE_INT`: Triggered when all data in the transmitter's TX FIFO has been sent.
- `UART_TX_BRK_IDLE_DONE_INT`: Triggered when the transmitter stays idle for the minimum interval (threshold) after sending the last data bit.
- `UART_TX_BRK_DONE_INT`: Triggered when the transmitter sends a NULL character after all data in TX FIFO has been sent.
- `UART_GLITCH_DET_INT`: Triggered when the receiver detects a glitch in the middle of the start bit.
- `UART_SW_XOFF_INT`: Triggered when `UART_SW_FLOW_CON_EN` is set and the receiver receives a Xoff character.
- `UART_SW_XON_INT`: Triggered when `UART_SW_FLOW_CON_EN` is set and the receiver receives a Xon character.
- `UART_RXFIFO_TOUT_INT`: Triggered when the receiver takes more time than `UART_RX_TOUT_THRHD` to receive one byte.
- `UART_BRK_DET_INT`: Triggered when the receiver detects a NULL character after stop bits.
- `UART_CTS_CHG_INT`: Triggered when the receiver detects an edge change of CTSn signals.
- `UART_DSR_CHG_INT`: Triggered when the receiver detects an edge change of DSRn signals.
- `UART_RXFIFO_OVF_INT`: Triggered when the receiver receives more data than the capacity of RX FIFO.
- `UART_FRM_ERR_INT`: Triggered when the receiver detects a data frame error.
- `UART_PARITY_ERR_INT`: Triggered when the receiver detects a parity error.
- `UART_TXFIFO_EMPTY_INT`: Triggered when TX FIFO stores less data than what `UART_TXFIFO_EMPTY_THRHD` specifies.
- `UART_RXFIFO_FULL_INT`: Triggered when the receiver receives more data than what `UART_RXFIFO_FULL_THRHD` specifies.
- `UART_WAKEUP_INT`: Triggered when UART is woken up.

### 8.3.12 UHCI Interrupts

- `UHCI_DMA_INFIFO_FULL_WM_INT`: Triggered when the counter value of DMA RX FIFO exceeds `UHCI_DMA_INFIFO_FULL_THRS`.
- `UHCI_SEND_A_REG_Q_INT`: Triggered when DMA has sent a series of short packets using `always_send`.
- `UHCI_SEND_S_REG_Q_INT`: Triggered when DMA has sent a series of short packets using `single_send`.
- `UHCI_OUT_TOTAL_EOF_INT`: Triggered when all data has been sent.
- `UHCI_OUTLINK_EOF_ERR_INT`: Triggered when an EOF error is detected in an transmit descriptor.
- `UHCI_IN_DSCR_EMPTY_INT`: Triggered when there are not enough receive descriptors for DMA.
- `UHCI_OUT_DSCR_ERR_INT`: Triggered when an error is detected in an transmit descriptor.
- `UHCI_IN_DSCR_ERR_INT`: Triggered when an error is detected in an receive descriptor.

- UHCl\_OUT\_EOF\_INT: Triggered when the EOF bit in a descriptor is 1.
- UHCl\_OUT\_DONE\_INT: Triggered when an transmit descriptor is completed.
- UHCl\_IN\_ERR\_EOF\_INT: Triggered when an EOF error is detected in an receive descriptor.
- UHCl\_IN\_SUC\_EOF\_INT: Triggered when a data packet has been received.
- UHCl\_IN\_DONE\_INT: Triggered when an receive descriptor is completed.
- UHCl\_TX\_HUNG\_INT: Triggered when DMA spends too much time on reading RAM.
- UHCl\_RX\_HUNG\_INT: Triggered when DMA spends too much time on receiving data.
- UHCl\_TX\_START\_INT: Triggered when DMA detects a separator character.
- UHCl\_RX\_START\_INT: Triggered when a separator character has been sent.

## 8.4 Base Address

Users can access UART0, UART1 and UHCIO respectively with two register base addresses shown in the following table. For more information about accessing peripherals from different buses please see Chapter 1: System and Memory.

**Table 36: Base addresses of UART0, UART1 and UHCIO**

Name	Accessed by	Base Address
UART0	PeriBUS1	0x3F400000
	PeriBUS2	0x60000000
UART1	PeriBUS1	0x3F410000
	PeriBUS2	0x60010000
UHCIO	PeriBUS1	0x3F414000
	PeriBUS2	0x60014000

## 8.5 Register Summary

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

Name	Description	Address	Access
<b>FIFO Configuration</b>			
UART_FIFO_REG	FIFO data register	0x0000	RO
UART_MEM_CONF_REG	UART threshold and allocation configuration	0x005C	R/W
<b>Interrupt registers</b>			
UART_INT_RAW_REG	Raw interrupt status	0x0004	RO
<b>Interrupt Register</b>			
UART_INT_ST_REG	Masked interrupt status	0x0008	RO
UART_INT_ENA_REG	Interrupt enable bits	0x000C	R/W
UART_INT_CLR_REG	Interrupt clear bits	0x0010	WO
<b>Configuration Register</b>			
UART_CLKDIV_REG	Clock divider configuration	0x0014	R/W

Name	Description	Address	Access
UART_CONF0_REG	Configuration register 0	0x0020	R/W
UART_CONF1_REG	Configuration register 1	0x0024	R/W
UART_FLOW_CONF_REG	Software flow control configuration	0x0034	R/W
UART_SLEEP_CONF_REG	Sleeping mode configuration	0x0038	R/W
UART_SWFC_CONF0_REG	Software flow control character configuration	0x003C	R/W
UART_SWFC_CONF1_REG	Software flow-control character configuration	0x0040	R/W
UART_IDLE_CONF_REG	Frame-end idle configuration	0x0044	R/W
UART_RS485_CONF_REG	RS485 mode configuration	0x0048	R/W
<b>Autobaud Register</b>			
UART_AUTOBAUD_REG	Autobaud configuration register	0x0018	R/W
UART_LOWPULSE_REG	Autobaud minimum low pulse duration register	0x0028	RO
UART_HIGHPULSE_REG	Autobaud minimum high pulse duration register	0x002C	RO
UART_RXD_CNT_REG	Autobaud edge change count register	0x0030	RO
UART_POSPULSE_REG	Autobaud high pulse register	0x006C	RO
UART_NEGPULSE_REG	Autobaud low pulse register	0x0070	RO
<b>Status Register</b>			
UART_STATUS_REG	UART status register	0x001C	RO
UART_MEM_TX_STATUS_REG	TX FIFO write and read offset address	0x0060	RO
UART_MEM_RX_STATUS_REG	RX FIFO write and read offset address	0x0064	RO
UART_FSM_STATUS_REG	UART transmit and receive status	0x0068	RO
<b>AT Escape Sequence Selection Configuration</b>			
UART_AT_CMD_PRECNT_REG	Pre-sequence timing configuration	0x004C	R/W
UART_AT_CMD_POSTCNT_REG	Post-sequence timing configuration	0x0050	R/W
UART_AT_CMD_GAPTOUT_REG	Timeout configuration	0x0054	R/W
UART_AT_CMD_CHAR_REG	AT Escape Sequence Selection Configuration	0x0058	R/W
<b>Version Register</b>			
UART_DATE_REG	UART version control register	0x0074	R/W

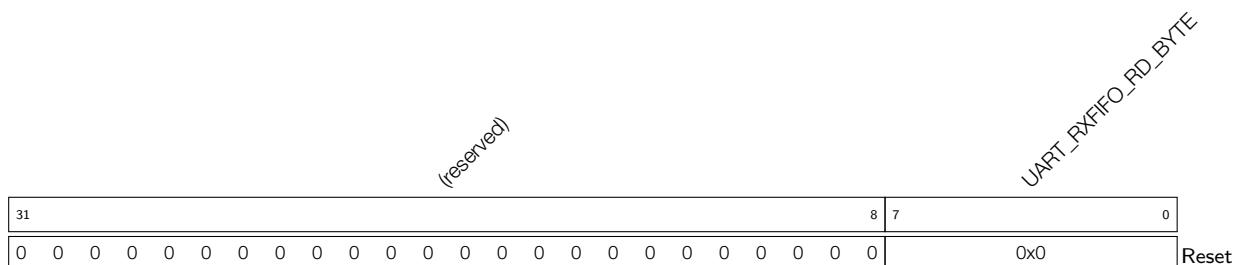
Name	Description	Address	Access
<b>Configuration Register</b>			
UHCI_CONF0_REG	UHCI configuration register	0x0000	R/W
UHCI_CONF1_REG	UHCI configuration register	0x002C	R/W
UHCI_AHB_TEST_REG	AHB test register	0x0048	R/W
UHCI_ESCAPE_CONF_REG	Escape characters configuration	0x0064	R/W
UHCI_HUNG_CONF_REG	Timeout configuration	0x0068	R/W
UHCI_QUICK_SENT_REG	UHCI quick send configuration register	0x0074	R/W
UHCI_REG_Q0_WORD0_REG	Q0_WORD0 quick_sent register	0x0078	R/W
UHCI_REG_Q0_WORD1_REG	Q0_WORD1 quick_sent register	0x007C	R/W
UHCI_REG_Q1_WORD0_REG	Q1_WORD0 quick_sent register	0x0080	R/W
UHCI_REG_Q1_WORD1_REG	Q1_WORD1 quick_sent register	0x0084	R/W
UHCI_REG_Q2_WORD0_REG	Q2_WORD0 quick_sent register	0x0088	R/W
UHCI_REG_Q2_WORD1_REG	Q2_WORD1 quick_sent register	0x008C	R/W
UHCI_REG_Q3_WORD0_REG	Q3_WORD0 quick_sent register	0x0090	R/W

Name	Description	Address	Access
UHCI_REG_Q3_WORD1_REG	Q3_WORD1 quick_sent register	0x0094	R/W
UHCI_REG_Q4_WORD0_REG	Q4_WORD0 quick_sent register	0x0098	R/W
UHCI_REG_Q4_WORD1_REG	Q4_WORD1 quick_sent register	0x009C	R/W
UHCI_REG_Q5_WORD0_REG	Q5_WORD0 quick_sent register	0x00A0	R/W
UHCI_REG_Q5_WORD1_REG	Q5_WORD1 quick_sent register	0x00A4	R/W
UHCI_REG_Q6_WORD0_REG	Q6_WORD0 quick_sent register	0x00A8	R/W
UHCI_REG_Q6_WORD1_REG	Q6_WORD1 quick_sent register	0x00AC	R/W
UHCI_ESC_CONF0_REG	Escape sequence configuration register 0	0x00B0	R/W
UHCI_ESC_CONF1_REG	Escape sequence configuration register 1	0x00B4	R/W
UHCI_ESC_CONF2_REG	Escape sequence configuration register 2	0x00B8	R/W
UHCI_ESC_CONF3_REG	Escape sequence configuration register 3	0x00BC	R/W
UHCI_PKT_THRES_REG	Configure register for packet length	0x00C0	R/W
<b>Interrupt Register</b>			
UHCI_INT_RAW_REG	Raw interrupt status	0x0004	RO
UHCI_INT_ST_REG	Masked interrupt status	0x0008	RO
UHCI_INT_ENA_REG	Interrupt enable bits	0x000C	R/W
UHCI_INT_CLR_REG	Interrupt clear bits	0x0010	WO
<b>DMA Status</b>			
UHCI_DMA_OUT_STATUS_REG	DMA data-output status register	0x0014	RO
UHCI_DMA_IN_STATUS_REG	UHCI data-input status register	0x001C	RO
UHCI_STATE0_REG	UHCI decoder status register	0x0030	RO
UHCI_DMA_OUT_EOF_DES_ADDR_REG	Outlink descriptor address when EOF occurs	0x0038	RO
UHCI_DMA_IN_SUC_EOF_DES_ADDR_REG	Inlink descriptor address when EOF occurs	0x003C	RO
UHCI_DMA_IN_ERR_EOF_DES_ADDR_REG	Inlink descriptor address when errors occur	0x0040	RO
UHCI_DMA_OUT_EOF_BFR_DES_ADDR_REG	Outlink descriptor address before the last transmit descriptor	0x0044	RO
UHCI_DMA_IN_DSCR_REG	The third word of the next receive descriptor	0x004C	RO
UHCI_DMA_IN_DSCR_BF0_REG	The third word of current receive descriptor	0x0050	RO
UHCI_DMA_IN_DSCR_BF1_REG	The second word of current receive descriptor	0x0054	RO
UHCI_DMA_OUT_DSCR_REG	The third word of the next transmit descriptor	0x0058	RO
UHCI_DMA_OUT_DSCR_BF0_REG	The third word of current transmit descriptor	0x005C	RO
UHCI_DMA_OUT_DSCR_BF1_REG	The second word of current transmit descriptor	0x0060	RO
UHCI_RX_HEAD_REG	UHCI packet header register	0x0070	RO
UHCI_STATE1_REG	UHCI encoder status register	0x0034	RO
<b>DMA Configuration</b>			
UHCI_DMA_OUT_PUSH_REG	Push control register of data-output FIFO	0x0018	R/W
UHCI_DMA_IN_POP_REG	Pop control register of data-input FIFO	0x0020	varies
UHCI_DMA_OUT_LINK_REG	Link descriptor address and control	0x0024	varies
UHCI_DMA_IN_LINK_REG	Link descriptor address and control	0x0028	varies
<b>Version Register</b>			

Name	Description	Address	Access
<a href="#">UHCI_DATE_REG</a>	UHCI version control register	0x00FC	R/W

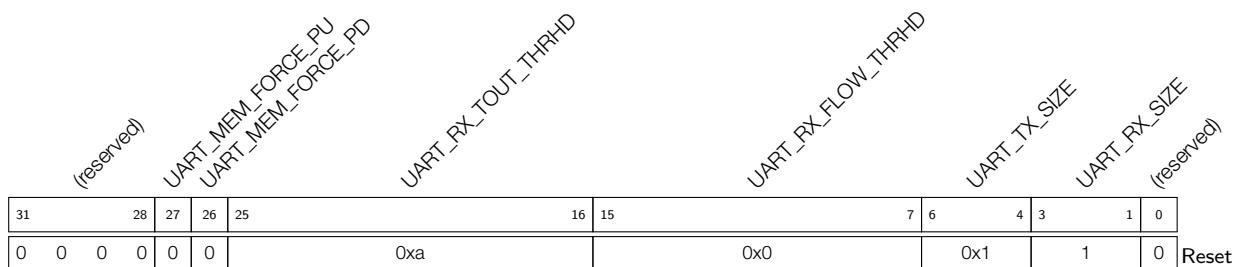
## 8.6 Registers

## Register 8.1: UART\_FIFO\_REG (0x0000)



**UART RXFIFO RD BYTE** This register stores one byte data read from RX FIFO. (RO)

## Register 8.2: UART\_MEM\_CONF\_REG (0x005C)



**UART\_RX\_SIZE** This register is used to configure the amount of mem allocated for RX FIFO. The default number is 128 bytes. (R/W)

**UART\_TX\_SIZE** This register is used to configure the amount of mem allocated for TX FIFO. The default number is 128 bytes. (R/W)

**UART\_RX\_FLOW\_THRHD** This register is used to configure the maximum amount of data that can be received when hardware flow control works. (R/W)

**UART\_RX\_TOUT\_THRHD** This register is used to configure the threshold time that receiver takes to receive one byte. The UART\_RXFIFO\_TOUT\_INT interrupt will be triggered when the receiver takes more time to receive one byte with UART RX TOUT EN set to 1. (R/W)

**UART MEM FORCE PD** Set this bit to force power down UART memory. (R/W)

**UART MEM FORCE PU** Set this bit to force power up UART memory. (R/W)

## Register 8.3: UART\_INT\_RAW\_REG (0x0004)

UART_INT_RAW Register																						
31	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

**UART\_RXFIFO\_FULL\_INT\_RAW** This interrupt raw bit turns to high level when receiver receives more data than what UART\_RXFIFO\_FULL\_THRHD specifies. (RO)

**UART\_TXFIFO\_EMPTY\_INT\_RAW** This interrupt raw bit turns to high level when the amount of data in TX FIFO is less than what UART\_TXFIFO\_EMPTY\_THRHD specifies. (RO)

**UART\_PARITY\_ERR\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects a parity error in the data. (RO)

**UART\_FRM\_ERR\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects a data frame error. (RO)

**UART\_RXFIFO\_OVF\_INT\_RAW** This interrupt raw bit turns to high level when receiver receives more data than the FIFO can store. (RO)

**UART\_DSR\_CHG\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects the edge change of DSRn signal. (RO)

**UART\_CTS\_CHG\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects the edge change of CTSn signal. (RO)

**UART\_BRK\_DET\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects a 0 after the stop bit. (RO)

**UART\_RXFIFO\_TOUT\_INT\_RAW** This interrupt raw bit turns to high level when receiver takes more time than UART\_RX\_TOUT\_THRHD to receive a byte. (RO)

**UART\_SW\_XON\_INT\_RAW** This interrupt raw bit turns to high level when receiver receives XON character when UART\_SW\_FLOW\_CON\_EN is set to 1. (RO)

**UART\_SW\_XOFF\_INT\_RAW** This interrupt raw bit turns to high level when receiver receives XOFF character when UART\_SW\_FLOW\_CON\_EN is set to 1. (RO)

**UART\_GLITCH\_DET\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects a glitch in the middle of a start bit. (RO)

**UART\_TX\_BRK\_DONE\_INT\_RAW** This interrupt raw bit turns to high level when transmitter completes sending NULL characters, after all data in TX FIFO are sent. (RO)

**UART\_TX\_BRK\_IDLE\_DONE\_INT\_RAW** This interrupt raw bit turns to high level when transmitter has kept the shortest duration after sending the last data. (RO)

Continued on the next page...

**Register 8.3: UART\_INT\_RAW\_REG (0x0004)**

Continued from the previous page...

**UART\_TX\_DONE\_INT\_RAW** This interrupt raw bit turns to high level when transmitter has sent out all data in FIFO. (RO)

**UART\_RS485\_PARITY\_ERR\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects a parity error from the echo of transmitter in RS485 mode. (RO)

**UART\_RS485\_FRM\_ERR\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects a data frame error from the echo of transmitter in RS485 mode. (RO)

**UART\_RS485\_CLASH\_INT\_RAW** This interrupt raw bit turns to high level when detects a clash between transmitter and receiver in RS485 mode. (RO)

**UART\_AT\_CMD\_CHAR\_DET\_INT\_RAW** This interrupt raw bit turns to high level when receiver detects the configured UART\_AT\_CMD CHAR. (RO)

**UART\_WAKEUP\_INT\_RAW** This interrupt raw bit turns to high level when input rxd edge changes more times than what UART\_ACTIVE\_THRESHOLD specifies in Light-sleep mode. (RO)

## Register 8.4: UART\_INT\_ST\_REG (0x0008)

31	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset							
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	0	0	0	0	0

**UART\_RXFIFO\_FULL\_INT\_ST** This is the status bit for UART\_RXFIFO\_FULL\_INT\_RAW when UART\_RXFIFO\_FULL\_INT\_ENA is set to 1. (RO)

**UART\_TXFIFO\_EMPTY\_INT\_ST** This is the status bit for UART\_TXFIFO\_EMPTY\_INT\_RAW when UART\_TXFIFO\_EMPTY\_INT\_ENA is set to 1. (RO)

**UART\_PARITY\_ERR\_INT\_ST** This is the status bit for UART\_PARITY\_ERR\_INT\_RAW when UART\_PARITY\_ERR\_INT\_ENA is set to 1. (RO)

**UART\_FRM\_ERR\_INT\_ST** This is the status bit for UART\_FRM\_ERR\_INT\_RAW when UART\_FRM\_ERR\_INT\_ENA is set to 1. (RO)

**UART\_RXFIFO\_OVF\_INT\_ST** This is the status bit for UART\_RXFIFO\_OVF\_INT\_RAW when UART\_RXFIFO\_OVF\_INT\_ENA is set to 1. (RO)

**UART\_DSR\_CHG\_INT\_ST** This is the status bit for UART\_DSR\_CHG\_INT\_RAW when UART\_DSR\_CHG\_INT\_ENA is set to 1. (RO)

**UART\_CTS\_CHG\_INT\_ST** This is the status bit for UART\_CTS\_CHG\_INT\_RAW when UART\_CTS\_CHG\_INT\_ENA is set to 1. (RO)

**UART\_BRK\_DET\_INT\_ST** This is the status bit for UART\_BRK\_DET\_INT\_RAW when UART\_BRK\_DET\_INT\_ENA is set to 1. (RO)

**UART\_RXFIFO\_TOUT\_INT\_ST** This is the status bit for UART\_RXFIFO\_TOUT\_INT\_RAW when UART\_RXFIFO\_TOUT\_INT\_ENA is set to 1. (RO)

**UART\_SW\_XON\_INT\_ST** This is the status bit for UART\_SW\_XON\_INT\_RAW when UART\_SW\_XON\_INT\_ENA is set to 1. (RO)

**UART\_SW\_XOFF\_INT\_ST** This is the status bit for UART\_SW\_XOFF\_INT\_RAW when UART\_SW\_XOFF\_INT\_ENA is set to 1. (RO)

**UART\_GLITCH\_DET\_INT\_ST** This is the status bit for UART\_GLITCH\_DET\_INT\_RAW when UART\_GLITCH\_DET\_INT\_ENA is set to 1. (RO)

**UART\_TX\_BRK\_DONE\_INT\_ST** This is the status bit for UART\_TX\_BRK\_DONE\_INT\_RAW when UART\_TX\_BRK\_DONE\_INT\_ENA is set to 1. (RO)

**UART\_TX\_BRK\_IDLE\_DONE\_INT\_ST** This is the status bit for UART\_TX\_BRK\_IDLE\_DONE\_INT\_RAW when UART\_TX\_BRK\_IDLE\_DONE\_INT\_ENA is set to 1. (RO)

**Continued on the next page...**

**Register 8.4: UART\_INT\_ST\_REG (0x0008)**

Continued from the previous page...

**UART\_TX\_DONE\_INT\_ST** This is the status bit for UART\_TX\_DONE\_INT\_RAW when UART\_TX\_DONE\_INT\_ENA is set to 1. (RO)

**UART\_RS485\_PARITY\_ERR\_INT\_ST** This is the status bit for UART\_RS485\_PARITY\_ERR\_INT\_RAW when UART\_RS485\_PARITY\_INT\_ENA is set to 1. (RO)

**UART\_RS485\_FRM\_ERR\_INT\_ST** This is the status bit for UART\_RS485\_FRM\_ERR\_INT\_RAW when UART\_RS485\_FM\_ERR\_INT\_ENA is set to 1. (RO)

**UART\_RS485\_CLASH\_INT\_ST** This is the status bit for UART\_RS485\_CLASH\_INT\_RAW when UART\_RS485\_CLASH\_INT\_ENA is set to 1. (RO)

**UART\_AT\_CMD\_CHAR\_DET\_INT\_ST** This is the status bit for UART\_AT\_CMD\_DET\_INT\_RAW when UART\_AT\_CMD\_CHAR\_DET\_INT\_ENA is set to 1. (RO)

**UART\_WAKEUP\_INT\_ST** This is the status bit for UART\_WAKEUP\_INT\_RAW when UART\_WAKEUP\_INT\_ENA is set to 1. (RO)

## Register 8.5: UART\_INT\_ENA\_REG (0x000C)

UART_WAKEUP_INT_ENA UART_AT_CMD_CHAR_DET_INT_ENA UART_RS485_CLASH_INT_ENA UART_RS485_FRM_ERR_INT_ENA UART_RS485_PARITY_ERR_INT_ENA UART_RS485_TX_BRK_IDLE_DONE_INT_ENA UART_RX_BRK_DONE_INT_ENA UART_GLITCH_DET_INT_ENA UART_SW_XOFF_INT_ENA UART_SW_XON_INT_ENA UART_RXFIFO_TOUT_INT_ENA UART_DSR_CHG_INT_ENA UART_RXFIFO_OVF_INT_ENA UART_FRM_ERR_INT_ENA UART_RXFIFO_EMPTY_INT_ENA UART_RXFIFO_FULL_INT_ENA																													
(reserved)																													
31	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Reset							
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0							

**UART\_RXFIFO\_FULL\_INT\_ENA** This is the enable bit for UART\_RXFIFO\_FULL\_INT\_ST register.

(R/W)

**UART\_TXFIFO\_EMPTY\_INT\_ENA** This is the enable bit for UART\_TXFIFO\_EMPTY\_INT\_ST register.

(R/W)

**UART\_PARITY\_ERR\_INT\_ENA** This is the enable bit for UART\_PARITY\_ERR\_INT\_ST register. (R/W)

**UART\_FRM\_ERR\_INT\_ENA** This is the enable bit for UART\_FRM\_ERR\_INT\_ST register. (R/W)

**UART\_RXFIFO\_OVF\_INT\_ENA** This is the enable bit for UART\_RXFIFO\_OVF\_INT\_ST register. (R/W)

**UART\_DSR\_CHG\_INT\_ENA** This is the enable bit for UART\_DSR\_CHG\_INT\_ST register. (R/W)

**UART\_CTS\_CHG\_INT\_ENA** This is the enable bit for UART\_CTS\_CHG\_INT\_ST register. (R/W)

**UART\_BRK\_DET\_INT\_ENA** This is the enable bit for UART\_BRK\_DET\_INT\_ST register. (R/W)

**UART\_RXFIFO\_TOUT\_INT\_ENA** This is the enable bit for UART\_RXFIFO\_TOUT\_INT\_ST register.

(R/W)

**UART\_SW\_XON\_INT\_ENA** This is the enable bit for UART\_SW\_XON\_INT\_ST register. (R/W)

**UART\_SW\_XOFF\_INT\_ENA** This is the enable bit for UART\_SW\_XOFF\_INT\_ST register. (R/W)

**UART\_GLITCH\_DET\_INT\_ENA** This is the enable bit for UART\_GLITCH\_DET\_INT\_ST register. (R/W)

**UART\_TX\_BRK\_DONE\_INT\_ENA** This is the enable bit for UART\_TX\_BRK\_DONE\_INT\_ST register. (R/W)

**UART\_TX\_BRK\_IDLE\_DONE\_INT\_ENA** This is the enable bit for UART\_TX\_BRK\_IDLE\_DONE\_INT\_ST register. (R/W)

**UART\_TX\_DONE\_INT\_ENA** This is the enable bit for UART\_TX\_DONE\_INT\_ST register. (R/W)

**UART\_RS485\_PARITY\_ERR\_INT\_ENA** This is the enable bit for UART\_RS485\_PARITY\_ERR\_INT\_ST register. (R/W)

**UART\_RS485\_FRM\_ERR\_INT\_ENA** This is the enable bit for UART\_RS485\_FRM\_ERR\_INT\_ST register. (R/W)

**Continued on the next page...**

**Register 8.5: UART\_INT\_ENA\_REG (0x000C)**

Continued from the previous page...

**UART\_RS485\_CLASH\_INT\_ENA** This is the enable bit for UART\_RS485\_CLASH\_INT\_ST register.  
(R/W)

**UART\_AT\_CMD\_CHAR\_DET\_INT\_ENA** This is the enable bit for  
UART\_AT\_CMD\_CHAR\_DET\_INT\_ST register. (R/W)

**UART\_WAKEUP\_INT\_ENA** This is the enable bit for UART\_WAKEUP\_INT\_ST register. (R/W)

### Register 8.6: UART\_INT\_CLR\_REG (0x0010)

**UART\_RXFIFO\_FULL\_INT\_CLR** Set this bit to clear UART\_THE\_RXFIFO\_FULL\_INT\_RAW interrupt.

(WO)

**UART\_TXFIFO\_EMPTY\_INT\_CLR** Set this bit to clear UART\_TXFIFO\_EMPTY\_INT\_RAW interrupt.

(WO)

**UART\_PARITY\_ERR\_INT\_CLR** Set this bit to clear UART\_PARITY\_ERR\_INT\_RAW interrupt. (WO)

**UART\_FRM\_ERR\_INT\_CLR** Set this bit to clear UART\_FRM\_ERR\_INT\_RAW interrupt. (WO)

**UART\_RXFIFO\_OVF\_INT\_CLR** Set this bit to clear UART\_RXFIFO\_OVF\_INT\_RAW interrupt.  
(WO)

**UART\_DSR\_CHG\_INT\_CLR** Set this bit to clear UART\_DSR\_CHG\_INT\_RAW interrupt. (WO)

**UART\_CTS\_CHG\_INT\_CLR** Set this bit to clear UART\_CTS\_CHG\_INT\_RAW interrupt. (WO)

**UART\_BRK\_DET\_INT\_CLR** Set this bit to clear UART\_BRK\_DET\_INT\_RAW interrupt. (WO)

**UART\_RXFIFO\_TOUT\_INT\_CLR** Set this bit to clear UART\_RXFIFO\_TOUT\_INT\_RAW interrupt.  
(WO)

**UART\_SW\_XON\_INT\_CLR** Set this bit to clear UART\_SW\_XON\_INT\_RAW interrupt. (WO)

**UART\_SW\_XOFF\_INT\_CLR** Set this bit to clear UART\_SW\_XOFF\_INT\_RAW interrupt. (WO)

**UART\_GLITCH\_DET\_INT\_CLR** Set this bit to clear UART\_GLITCH\_DET\_INT\_RAW interrupt. (WO)

**UART\_TX\_BRK\_DONE\_INT\_CLR** Set this bit to clear UART\_TX\_BRK\_DONE\_INT\_RAW interrupt.  
(WO)

**UART\_TX\_BRK\_IDLE\_DONE\_INT\_CLR** Set this bit to clear UART\_TX\_BRK\_IDLE\_DONE\_INT\_RAW interrupt. (WO)

**UART\_TX\_DONE\_INT\_CLR** Set this bit to clear UART\_TX\_DONE\_INT\_RAW interrupt. (WO)

**UART\_RS485\_PARITY\_ERR\_INT\_CLR** Set this bit to clear UART\_RS485\_PARITY\_ERR\_INT\_RAW interrupt. (WO)

**UART\_RS485\_FRM\_ERR\_INT\_CLR** Set this bit to clear UART\_RS485\_FRM\_ERR\_INT\_RAW interrupt. (WO)

**UART\_RS485\_CLASH\_INT\_CLR** Set this bit to clear UART\_RS485\_CLASH\_INT\_RAW interrupt.  
(WO)

**Register 8.6: UART\_INT\_CLR\_REG (0x0010)**

Continued from the previous page...

**UART\_AT\_CMD\_CHAR\_DET\_INT\_CLR** Set this bit to clear UART\_AT\_CMD\_CHAR\_DET\_INT\_RAW interrupt. (WO)

**UART\_WAKEUP\_INT\_CLR** Set this bit to clear UART\_WAKEUP\_INT\_RAW interrupt. (WO)

**Register 8.7: UART\_CLKDIV\_REG (0x0014)**

31	24	23	20	19	0
0	0	0	0	0	0x2b6 Reset

Diagram illustrating the bit fields of Register 8.7: UART\_CLKDIV\_REG (0x0014). The register is 32 bits wide, with fields labeled as follows:

- (reserved) bits 31 to 25
- UART\_CLKDIV\_FRAG bits 24 to 20
- UART\_CLKDIV bits 19 to 0

**UART\_CLKDIV** The integral part of the frequency divisor. (R/W)

**UART\_CLKDIV\_FRAG** The decimal part of the frequency divisor. (R/W)

## Register 8.8: UART\_CONF0\_REG (0x0020)

	UART_MEM_CLK_EN	UART_TICK_REF_ALWAYS_ON	(reserved)	UART_DTR_INV	UART_RTS_INV	UART_RXD_INV	UART_DSR_INV	UART_CTS_INV	UART_RXFIFO_RST	UART_RXFIFO_RST	UART_RXD_EN	UART_RXD_FLOW_EN	UART_RXD_LOOPBACK	UART_RXDA_RX_INV	UART_RXDA_WCTL	UART_RXDA_TX_EN	UART_RXDA_BRK	UART_SW_DTR	UART_STOP_BIT_NUM	UART_BIT_NUM	UART_PARITY_EN	UART_PARITY								
31	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	0	0	Reset

**UART\_PARITY** This register is used to configure the parity check mode. 1'h0: even. 1'h1: odd.

(R/W)

**UART\_PARITY\_EN** Set this bit to enable UART parity check. (R/W)

**UART\_BIT\_NUM** This register is used to set the length of data. 0: 5 bits 1: 6 bits 2: 7 bits 3: 8 bits.

(R/W)

**UART\_STOP\_BIT\_NUM** This register is used to set the length of stop bit. 1: 1 bit 2: 1.5 bits 3: 2 bits. (R/W)

**UART\_SW\_RTS** This register is used to configure the software RTS signal which is used in software flow control. (R/W)

**UART\_SW\_DTR** This register is used to configure the software DTR signal which is used in software flow control. (R/W)

**UART\_TXD\_BRK** Set this bit to enable transmitter to send NULL when the process of sending data is done. (R/W)

**UART\_IRDA\_DPLX** Set this bit to enable IrDA loopback mode. (R/W)

**UART\_IRDA\_TX\_EN** This is the start enable bit for IrDA transmitter. (R/W)

**UART\_IRDA\_WCTL** 1'h1: The IrDA transmitter's 11th bit is the same as 10th bit. 1'h0: Set IrDA transmitter's 11th bit to 0. (R/W)

**UART\_IRDA\_TX\_INV** Set this bit to invert the level of IrDA transmitter. (R/W)

**UART\_IRDA\_RX\_INV** Set this bit to invert the level of IrDA receiver. (R/W)

**UART\_LOOPBACK** Set this bit to enable UART loopback test mode. (R/W)

**UART\_TX\_FLOW\_EN** Set this bit to enable flow control function for transmitter. (R/W)

**UART\_IRDA\_EN** Set this bit to enable IrDA protocol. (R/W)

**UART\_RXFIFO\_RST** Set this bit to reset the UART RX FIFO. (R/W)

**UART\_TXFIFO\_RST** Set this bit to reset the UART TX FIFO. (R/W)

**UART\_RXD\_INV** Set this bit to inverse the level value of UART RXD signal. (R/W)

**UART\_CTS\_INV** Set this bit to inverse the level value of UART CTS signal. (R/W)

**UART\_DSR\_INV** Set this bit to inverse the level value of UART DSR signal. (R/W)

**Register 8.8: UART\_CONF0\_REG (0x0020)**

**Continued from the previous page...**

**UART\_TXD\_INV** Set this bit to inverse the level value of UART TXD signal. (R/W)

**UART\_RTS\_INV** Set this bit to inverse the level value of UART RTS signal. (R/W)

**UART\_DTR\_INV** Set this bit to inverse the level value of UART DTR signal. (R/W)

**UART\_TICK\_REF\_ALWAYS\_ON** This register is used to select the clock. 1'h1: APB\_CLK. 1'h0: REF\_TICK. (R/W)

**UART\_MEM\_CLK\_EN** UART memory clock gate enable signal. 1: UART memory powers on, the data of which can be read and written. 0: UART memory powers down. (R/W)

**Register 8.9: UART\_CONF1\_REG (0x0024)**

31	30	29	28	18	17	9	8	0	Reset
0	0	0	0	0	0	0	0	0	0x60

**UART\_RXFIFO\_FULL\_THRHD** It will produce UART\_RXFIFO\_FULL\_INT interrupt when receiver receives more data than this register value. (R/W)

**UART\_TXFIFO\_EMPTY\_THRHD** It will produce UART\_TXFIFO\_EMPTY\_INT interrupt when the data amount in TX FIFO is less than this register value. (R/W)

**UART\_RX\_TOUT\_FLOW\_DIS** Set this bit to stop accumulating idle\_cnt when hardware flow control works. (R/W)

**UART\_RX\_FLOW\_EN** This is the flow enable bit for UART receiver. 1'h1: Choose software flow control with configuring sw\_rts signal. 1'h0: Disable software flow control. (R/W)

**UART\_RX\_TOUT\_EN** This is the enable bit for UART receiver's timeout function. (R/W)

## Register 8.10: UART\_FLOW\_CONF\_REG (0x0034)

The diagram shows the bit field layout of Register 8.10. The register is 32 bits wide, with bit 31 being the most significant bit. Bit 31 is labeled '(reserved)'. Bits 6 through 0 are labeled from left to right as: 6 (UART\_SEND\_XOFF), 5 (UART\_SEND\_XON), 4 (UART\_FORCE\_XOFF), 3 (UART\_FORCE\_XON), 2 (UART\_XONOFF\_DEL), 1 (UART\_SW\_FLOW\_CON\_EN), and 0 (Reset). The Reset bit is also labeled 'Reset' at the bottom right.

31	6	5	4	3	2	1	0
0 0	0	0	0	0	0	0	Reset

**UART\_SW\_FLOW\_CON\_EN** Set this bit to enable software flow control. It is used with register SW\_XON or SW\_XOFF. (R/W)

**UART\_XONOFF\_DEL** Set this bit to remove flow control character from the received data. (R/W)

**UART\_FORCE\_XON** Set this bit to enable the transmitter to go on sending data. (R/W)

**UART\_FORCE\_XOFF** Set this bit to stop the transmitter from sending data. (R/W)

**UART\_SEND\_XON** Set this bit to send XON character. It is cleared by hardware automatically. (R/W)

**UART\_SEND\_XOFF** Set this bit to send XOFF character. It is cleared by hardware automatically. (R/W)

## Register 8.11: UART\_SLEEP\_CONF\_REG (0x0038)

The diagram shows the bit field layout of Register 8.11. The register is 32 bits wide, with bit 31 being the most significant bit. Bit 31 is labeled '(reserved)'. Bits 10 and 9 are labeled 'UART\_ACTIVE\_THRESHOLD'. Bits 0 through 8 are all labeled '0'. The '0' bits are also labeled 'Reset' at the bottom right.

31	10	9	0
0 0	0xf0	0	Reset

**UART\_ACTIVE\_THRESHOLD** The UART is activated from light sleeping mode when the input rxd edge changes more times than this register value. (R/W)

**Register 8.12: UART\_SWFC\_CONF0\_REG (0x003C)**

31	17	16	9	8	0
0	0	0	0	0	0xe0

Reset

**UART\_XOFF\_THRESHOLD** When the data amount in RX FIFO is more than this register value with UART\_SW\_FLOW\_CON\_EN set to 1, it will send a XOFF character. (R/W)

**UART\_XOFF\_CHAR** This register stores the XOFF flow control character. (R/W)

**Register 8.13: UART\_SWFC\_CONF1\_REG (0x0040)**

31	17	16	9	8	0
0	0	0	0	0	0x0

Reset

**UART\_XON\_THRESHOLD** When the data amount in RX FIFO is less than this register value with UART\_SW\_FLOW\_CON\_EN set to 1, it will send a XON character. (R/W)

**UART\_XON\_CHAR** This register stores the XON flow control character. (R/W)

**Register 8.14: UART\_IDLE\_CONF\_REG (0x0044)**

31	28	27	20	19	10	9	0
0	0	0	0	0xa	0x100	0x100	0x0

Reset

**UART\_RX\_IDLE\_NUM** It will produce frame end signal when receiver takes more time to receive one byte data than this register value. (R/W)

**UART\_TX\_BRK\_NUM** This register is used to configure the duration time between transfers. (R/W)

**UART\_RX\_IDLE\_NUM** This register is used to configure the number of 0 to be sent after the process of sending data is done. It is active when UART\_TXD\_BRK is set to 1. (R/W)

## Register 8.15: UART\_RS485\_CONF\_REG (0x0048)

										UART_RS485_RX_DLY_NUM	UART_RS485_TX_DLY_NUM	
										UART_RS485_RX_BY_TX_EN	UART_RS485_TX_BY_RX_EN	
										UART_DL1_EN	UART_DL0_EN	
31	(reserved)	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Reset

**UART\_RS485\_EN** Set this bit to choose the RS485 mode. (R/W)

**UART\_DL0\_EN** Set this bit to delay the stop bit by 1 bit. (R/W)

**UART\_DL1\_EN** Set this bit to delay the stop bit by 1 bit. (R/W)

**UART\_RS485TX\_RX\_EN** Set this bit to enable receiver could receive data when the transmitter is transmitting data in RS485 mode. (R/W)

**UART\_RS485RXBY\_TX\_EN** 1'h1: enable RS485 transmitter to send data when RS485 receiver line is busy. 1'h0: RS485 transmitter should not send data when its receiver is busy. (R/W)

**UART\_RS485\_RX\_DLY\_NUM** This register is used to delay the receiver's internal data signal. (R/W)

**UART\_RS485\_TX\_DLY\_NUM** This register is used to delay the transmitter's internal data signal. (R/W)

## Register 8.16: UART\_AUTOBAUD\_REG (0x0018)

										UART_GLITCH_FILT	(reserved)	UART_AUTOBAUD_EN
31	(reserved)	16	15	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Reset

**UART\_AUTOBAUD\_EN** This is the enable bit for detecting baudrate. (R/W)

**UART\_GLITCH\_FILT** when input pulse width is lower than this value, the pulse is ignored. This register is used in autobaud detect process. (R/W)

**Register 8.17: UART\_LOWPULSE\_REG (0x0028)**

31	20	19	0
0	0	0	0
0xfffff			Reset

**UART\_LOWPULSE\_MIN\_CNT** This register stores the value of the minimum duration time of the low level pulse. It is used in baud rate detection. (RO)

**Register 8.18: UART\_HIGHPULSE\_REG (0x002C)**

31	20	19	0
0	0	0	0
0xfffff			Reset

**UART\_HIGHPULSE\_MIN\_CNT** This register stores the value of the maximum duration time for the high level pulse. It is used in baud rate detection. (RO)

**Register 8.19: UART\_RXD\_CNT\_REG (0x0030)**

31	20	19	0
0	0	0	0
0x0			Reset

**UART\_RXD\_EDGE\_CNT** This register stores the count of rxd edge change. It is used in baud rate detection. (RO)

**Register 8.20: UART\_POSPULSE\_REG (0x006C)**

31	20	19	0
0	0	0	0

UART\_POSEDGE\_MIN\_CNT

Reset

0xfffff

**UART\_POSEDGE\_MIN\_CNT** This register stores the minimal input clock count between two positive edges. It is used in baud rate detection. (RO)

**Register 8.21: UART\_NEGPULSE\_REG (0x0070)**

31	20	19	0
0	0	0	0

UART\_NEGEDGE\_MIN\_CNT

Reset

0xfffff

**UART\_NEGEDGE\_MIN\_CNT** This register stores the minimal input clock count between two negative edges. It is used in baud rate detection. (RO)

## Register 8.22: UART\_STATUS\_REG (0x001C)

31	30	29	28	26	25		16	15	14	13	12	10	9		0
0x0	0	0	0	0	0		0x0	0	0	0	0	0	0		0x0

Reset

**UART\_RXFIFO\_CNT** Stores the byte number of valid data in RX FIFO. (RO)

**UART\_DSRN** This register represent the level value of the internal UART DSR signal. (RO)

**UART\_CTSN** This register represent the level value of the internal UART CTS signal. (RO)

**UART\_RXD** This register represent the level value of the internal UART RXD signal. (RO)

**UART\_TXFIFO\_CNT** Stores the byte number of data in TX FIFO. (RO)

**UART\_DTRN** This bit represents the level of the internal UART DTR signal. (RO)

**UART\_RTSN** This bit represents the level of the internal UART RTS signal. (RO)

**UART\_TXD** This bit represents the level of the internal UART TXD signal. (RO)

## Register 8.23: UART\_MEM\_TX\_STATUS\_REG (0x0060)

31	21	20		11	10	9		0
0	0	0	0	0	0	0		0x0

Reset

**UART\_APP\_TX\_WADDR** This register stores the offset address in TX FIFO when software writes TX FIFO via APB. (RO)

**UART\_TX\_RADDR** This register stores the offset address in TX FIFO when TX FSM reads data via Tx\_FIFO\_Ctrl. (RO)

**Register 8.24: UART\_MEM\_RX\_STATUS\_REG (0x0064)**

										UART_RX_WADDR		(reserved)		UART_APB_RX_RADDR	
31	20	11	10	9	0										
0	0	0	0	0	0	0x0		0		0x0		0		Reset	

**UART\_APP\_RX\_RADDR** This register stores the offset address in RX\_FIFO when software reads data from RX FIFO via APB. (RO)

**UART\_RX\_WADDR** This register stores the offset address in RX FIFO when Rx\_FIFO\_Ctrl writes RX FIFO. (RO)

**Register 8.25: UART\_FSM\_STATUS\_REG (0x0068)**

										UART_ST_UTX_OUT		UART_ST_URX_OUT			
31	8	7	4	3	0										
0	0	0	0	0	0	0x0		0x0		0x0		0		Reset	

**UART\_ST\_URX\_OUT** This is the status register of receiver. (RO)

**UART\_ST\_UTX\_OUT** This is the status register of transmitter. (RO)

**Register 8.26: UART\_AT\_CMD\_PRECNT\_REG (0x004C)**

										UART_PRE_IDLE_NUM				
31	16	15	0											
0	0	0	0	0	0	0	0	0	0	0x901		0	Reset	

**UART\_PRE\_IDLE\_NUM** This register is used to configure the idle duration time before the first AT\_CMD is received by receiver. It will not take the next data received as AT\_CMD character when the duration is less than this register value. (R/W)

**Register 8.27: UART\_AT\_CMD\_POSTCNT\_REG (0x0050)**

31	16	15	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		0x901	Reset

**UART\_POST\_IDLE\_NUM** This register is used to configure the duration time between the last AT\_CMD and the next data. It will not take the previous data as AT\_CMD character when the duration is less than this register value. (R/W)

**Register 8.28: UART\_AT\_CMD\_GAPTOUT\_REG (0x0054)**

31	16	15	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		11	Reset

**UART\_RX\_GAP\_TOUT** This register is used to configure the duration time between the AT\_CMD chars. It will not take the data as continuous AT\_CMD chars when the duration time is less than this register value. (R/W)

**Register 8.29: UART\_AT\_CMD\_CHAR\_REG (0x0058)**

31	16	15	8	7	0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		0x3		0x2b	Reset

**UART\_AT\_CMD\_CHAR** This register is used to configure the content of AT\_CMD character. (R/W)

**UART\_CHAR\_NUM** This register is used to configure the number of continuous AT\_CMD chars received by receiver. (R/W)

**Register 8.30: UART\_DATE\_REG (0x0074)**

UART_DATE	
31	0
0x18082800	Reset

**UART\_DATE** This is the version control register. (R/W)

## Register 8.31: UHCl\_CONF0\_REG (0x0000)

(reserved)	UHCl_UART_RX_BRK_EOF_EN	UHCl_CLK_EN	UHCl_ENCODE_CRC_EN	UHCl_LEN_EOF_EN	UHCl_UART_IDLE_EOF_EN	UHCl_CRC_REC_EN	UHCl_HEAD_EN	UHCl_SEPER_EN	UHCl_MEM_TRANS_EN	UHCl_OUT_DATA_BURST_EN	UHCl_INDSCR_BURST_EN	UHCl_OUTDSCR_BURST_EN	UHCl_UART1_CE	UHCl_UART0_CE	UHCl_OUT_EOF_MODE	UHCl_OUT_AUTO_WRBACK	UHCl_IN_LOOP_TEST	UHCl_AHBM_RST	UHCl_OUT_AHBM_RST	UHCl_IN_RST					
31	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	Reset

**UHCl\_IN\_RST** Set this bit to reset in DMA FSM. (R/W)**UHCl\_OUT\_RST** Set this bit to reset out DMA FSM. (R/W)**UHCl\_AHBM\_FIFO\_RST** Set this bit to reset AHB interface cmdFIFO of DMA. (R/W)**UHCl\_AHBM\_RST** Set this bit to reset AHB interface of DMA. (R/W)**UHCl\_IN\_LOOP\_TEST** Reserved. (R/W)**UHCl\_OUT\_LOOP\_TEST** Reserved. (R/W)**UHCl\_OUT\_AUTO\_WRBACK** Set this bit to enable automatic outlink-writeback when all the data in TX Buffer has been transmitted. (R/W)**UHCl\_OUT\_NO\_RESTART\_CLR** Reserved. (R/W)**UHCl\_OUT\_EOF\_MODE** This register is used to specify the generation mode of UHCl\_OUT\_EOF\_INT interrupt. 1: When DMA has popped all data from FIFO. 0: When AHB has pushed all data to FIFO. (R/W)**UHCl\_UART0\_CE** Set this bit to link up HCI and UART0. (R/W)**UHCl\_UART1\_CE** Set this bit to link up HCI and UART1. (R/W)**UHCl\_OUTDSCR\_BURST\_EN** This register is used to specify DMA transmit descriptor transfer mode. 1: burst mode. 0: byte mode. (R/W)**UHCl\_INDSCR\_BURST\_EN** This register is used to specify DMA receive descriptor transfer mode. 1: burst mode. 0: byte mode. (R/W)**UHCl\_OUT\_DATA\_BURST\_EN** This register is used to specify data transfer mode. 1: Transmit data in burst mode. 0: Transmit data in byte mode. (R/W)**UHCl\_MEM\_TRANS\_EN** 1: UHCl transmitted data would be write back into DMA INFIFO. (R/W)**UHCl\_SEPER\_EN** Set this bit to separate the data frame using a special char. (R/W)**UHCl\_HEAD\_EN** Set this bit to encode the data packet with a formatting header. (R/W)**UHCl\_CRC\_REC\_EN** Set this bit to enable UHCl to receive the 16 bit CRC. (R/W)**UHCl\_UART\_IDLE\_EOF\_EN** If this bit is set to 1, UHCl will end the payload receiving process when UART has been in idle state. (R/W)**Continued on the next page...**

**Register 8.31: UHCI\_CONF0\_REG (0x0000)**

**Continued from the previous page...**

**UHCI\_LEN\_EOF\_EN** If this bit is set to 1, UHCI decoder receiving payload data is end when the receiving byte count has reached the specified value. The value is payload length indicated by UCHI packet header when UHCI\_HEAD\_EN is 1 or the value is a configuration value when UHCI\_HEAD\_EN is 0. If this bit is set to 0, UHCI decoder receiving payload data is end when 0xc0 is received. (R/W)

**UHCI\_ENCODE\_CRC\_EN** Set this bit to enable data integrity checking by appending a 16 bit CCITT-CRC to the end of the payload. (R/W)

**UHCI\_CLK\_EN** 1'b1: Force clock on for register. 1'b0: Support clock only when application writes registers. (R/W)

**UHCI\_UART\_RX\_BRK\_EOF\_EN** if this bit is set to 1, UHCI will end payload\_rec process when NULL frame is received by UART. (R/W)

## Register 8.32: UHCI\_CONF1\_REG (0x002C)

										UHCI_DMA_INFIFO_FULL_THRS									
(reserved)										9	8	7	6	5	4	3	2	1	0
31	21	20	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1

Reset

**UHCI\_CHECK\_SUM\_EN** This is the enable bit to check header checksum when UHCI receives a data packet. (R/W)

**UHCI\_CHECK\_SEQ\_EN** This is the enable bit to check sequence number when UHCI receives a data packet. (R/W)

**UHCI\_CRC\_DISABLE** Set this bit to support CRC calculation. Data Integrity Check Present bit in UHCI packet frame should be 1. (R/W)

**UHCI\_SAVE\_HEAD** Set this bit to save the packet header when HCI receives a data packet. (R/W)

**UHCI\_TX\_CHECK\_SUM\_RE** Set this bit to encode the data packet with a checksum. (R/W)

**UHCI\_TX\_ACK\_NUM\_RE** Set this bit to encode the data packet with an acknowledgement when a reliable packet is to be transmit. (R/W)

**UHCI\_CHECK\_OWNER** 1: Check the link list descriptor when link list owner is DMA controller; 0: Always check link list descriptor. (R/W)

**UHCI\_WAIT\_SW\_START** The uhci-encoder will jump to ST\_SW\_WAIT status if this register is set to 1. (R/W)

**UHCI\_SW\_START** If current UHCI\_ENCODE\_STATE is ST\_SW\_WAIT, the UHCI will start to send data packet out when this bit is set to 1. (R/W)

**UHCI\_DMA\_INFIFO\_FULL\_THRS** This field is used to generate the UHCI\_DMA\_INFIFO\_FULL\_WM\_INT interrupt when the counter value of DMA RX FIFO exceeds the value of the register. (R/W)

## Register 8.33: UHCI\_AHB\_TEST\_REG (0x0048)

(reserved)										UHCI_AHB_TESTADDR		(reserved)		UHCI_AHB_TESTMODE		
31											6	5	4	3	2	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**UHCI\_AHB\_TESTMODE** Reserved. (R/W)**UHCI\_AHB\_TESTADDR** Reserved. (R/W)

## Register 8.34: UHCI\_ESCAPE\_CONF\_REG (0x0064)

(reserved)										UHCI_RX_13_ESC_EN		UHCI_RX_11_ESC_EN		UHCI_RX_DB_ESC_EN		UHCI_RX_C0_ESC_EN		UHCI_RX_13_ESC_EN		UHCI_RX_11_ESC_EN		UHCI_RX_DB_ESC_EN		UHCI_RX_C0_ESC_EN			
31											8	7	6	5	4	3	2	1	0								
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	Reset

**UHCI\_TX\_C0\_ESC\_EN** Set this bit to enable decoding char 0xc0 when DMA receives data. (R/W)**UHCI\_TX\_DB\_ESC\_EN** Set this bit to enable decoding char 0xdb when DMA receives data. (R/W)**UHCI\_TX\_11\_ESC\_EN** Set this bit to enable decoding flow control char 0x11 when DMA receives data. (R/W)**UHCI\_TX\_13\_ESC\_EN** Set this bit to enable decoding flow control char 0x13 when DMA receives data. (R/W)**UHCI\_RX\_C0\_ESC\_EN** Set this bit to enable replacing 0xc0 by special char when DMA sends data. (R/W)**UHCI\_RX\_DB\_ESC\_EN** Set this bit to enable replacing 0xdb by special char when DMA sends data. (R/W)**UHCI\_RX\_11\_ESC\_EN** Set this bit to enable replacing flow control char 0x11 by special char when DMA sends data. (R/W)**UHCI\_RX\_13\_ESC\_EN** Set this bit to enable replacing flow control char 0x13 by special char when DMA sends data. (R/W)

**Register 8.35: UHCl\_HUNG\_CONF\_REG (0x0068)**

(reserved)	UHCl_RXFIFO_TIMEOUT_ENA	UHCl_RXFIFO_TIMEOUT_SHIFT	UHCl_RXFIFO_TIMEOUT	UHCl_TXFIFO_TIMEOUT_ENA	UHCl_TXFIFO_TIMEOUT_SHIFT	UHCl_TXFIFO_TIMEOUT					
31	24	23	22	20	19	12	11	10	8	7	0
0 0 0 0 0 0 0 0 1 0				0x10		1	0		0x10		Reset

**UHCl\_TXFIFO\_TIMEOUT** This register stores the timeout value. It will produce the UHCl\_RX\_HUNG\_INT interrupt when DMA takes more time to receive data. (R/W)

**UHCl\_TXFIFO\_TIMEOUT\_SHIFT** This register is used to configure the tick count maximum value. (R/W)

**UHCl\_RXFIFO\_TIMEOUT\_ENA** This is the enable bit for Tx-FIFO receive-data timeout. (R/W)

**UHCl\_RXFIFO\_TIMEOUT** This register stores the timeout value. It will produce the UHCl\_RX\_HUNG\_INT interrupt when DMA takes more time to read data from RAM. (R/W)

**UHCl\_RXFIFO\_TIMEOUT\_SHIFT** This register is used to configure the tick count maximum value. (R/W)

**UHCl\_RXFIFO\_TIMEOUT\_ENA** This is the enable bit for DMA send-data timeout. (R/W)

**Register 8.36: UHCl\_QUICK\_SENT\_REG (0x0074)**

(reserved)	UHCl_ALWAYS_SEND_EN	UHCl_ALWAYS_SEND_NUM	UHCl_SINGLE_SEND_EN	UHCl_SINGLE_SEND_NUM			
31	8	7	6	4	3	2	0
0 0	0	0x0	0	0x0	0	0x0	Reset

**UHCl\_SINGLE\_SEND\_NUM** This register is used to specify the single\_send register. (R/W)

**UHCl\_SINGLE\_SEND\_EN** Set this bit to enable single\_send mode to send short packet. (R/W)

**UHCl\_ALWAYS\_SEND\_NUM** This register is used to specify the always\_send register. (R/W)

**UHCl\_ALWAYS\_SEND\_EN** Set this bit to enable always\_send mode to send short packet. (R/W)

**Register 8.37: UHCI\_REG\_Q0\_WORD0\_REG (0x0078)**

UHCI_SEND_Q0_WORD0	
31	0
0x000000	Reset

**UHCI\_SEND\_Q0\_WORD0** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.38: UHCI\_REG\_Q0\_WORD1\_REG (0x007C)**

UHCI_SEND_Q0_WORD1	
31	0
0x000000	Reset

**UHCI\_SEND\_Q0\_WORD1** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.39: UHCI\_REG\_Q1\_WORD0\_REG (0x0080)**

UHCI_SEND_Q1_WORD0	
31	0
0x000000	Reset

**UHCI\_SEND\_Q1\_WORD0** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.40: UHCI\_REG\_Q1\_WORD1\_REG (0x0084)**

UHCI_SEND_Q1_WORD1	
31	0
0x000000	Reset

**UHCI\_SEND\_Q1\_WORD1** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.41: UHCI\_REG\_Q2\_WORD0\_REG (0x0088)**

UHCI_SEND_Q2_WORD0	
31	0
0x000000	Reset

**UHCI\_SEND\_Q2\_WORD0** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.42: UHCI\_REG\_Q2\_WORD1\_REG (0x008C)**

UHCI_SEND_Q2_WORD1	
31	0
0x000000	Reset

**UHCI\_SEND\_Q2\_WORD1** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.43: UHCI\_REG\_Q3\_WORD0\_REG (0x0090)**

UHCI_SEND_Q3_WORD0	
31	0
0x000000	Reset

**UHCI\_SEND\_Q3\_WORD0** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.44: UHCI\_REG\_Q3\_WORD1\_REG (0x0094)**

UHCI_SEND_Q3_WORD1	
31	0
0x000000	Reset

**UHCI\_SEND\_Q3\_WORD1** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.45: UHCI\_REG\_Q4\_WORD0\_REG (0x0098)**

UHCI_SEND_Q4_WORD0	
31	0
0x000000	Reset

**UHCI\_SEND\_Q4\_WORD0** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.46: UHCI\_REG\_Q4\_WORD1\_REG (0x009C)**

UHCI_SEND_Q4_WORD1	
31	0
0x000000	Reset

**UHCI\_SEND\_Q4\_WORD1** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.47: UHCI\_REG\_Q5\_WORD0\_REG (0x00A0)**

UHCI_SEND_Q5_WORD0	
31	0
0x000000	Reset

**UHCI\_SEND\_Q5\_WORD0** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

**Register 8.48: UHCI\_REG\_Q5\_WORD1\_REG (0x00A4)**

UHCI_SEND_Q5_WORD1	
31	0
0x000000	Reset

**UHCI\_SEND\_Q5\_WORD1** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

## Register 8.49: UHCI\_REG\_Q6\_WORD0\_REG (0x00A8)

UHCI_SEND_Q6_WORD0		UHCI_REG_Q6_WORD0_REG (0x00A8)							
		31	24	23	16	15	8	7	0
0x000000									Reset

**UHCI\_SEND\_Q6\_WORD0** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

## Register 8.50: UHCI\_REG\_Q6\_WORD1\_REG (0x00AC)

UHCI_SEND_Q6_WORD1		UHCI_REG_Q6_WORD1_REG (0x00AC)							
		31	24	23	16	15	8	7	0
0x000000									Reset

**UHCI\_SEND\_Q6\_WORD1** This register is used as a quick\_sent register when specified by UHCI\_ALWAYS\_SEND\_NUM or UHCI\_SINGLE\_SEND\_NUM. (R/W)

## Register 8.51: UHCI\_ESC\_CONF0\_REG (0x00B0)

(reserved)		UHCI_SEPER_ESC_CHAR1				UHCI_SEPER_ESC_CHAR0				UHCI_SEPER_CHAR	
		31	24	23	16	15	8	7	0	0	0
0	0	0	0	0	0	0	0	0xd0	0xdb	0xc0	Reset

**UHCI\_SEPER\_CHAR** This register is used to define the separate char that need to be encoded, default is 0xc0. (R/W)

**UHCI\_SEPER\_ESC\_CHAR0** This register is used to define the first char of slip escape sequence when encoding the separate char, default is 0xdb. (R/W)

**UHCI\_SEPER\_ESC\_CHAR1** This register is used to define the second char of slip escape sequence when encoding the separate char, default is 0xdc. (R/W)

**Register 8.52: UHCI\_ESC\_CONF1\_REG (0x00B4)**

UHCI_ESC_SEQ0							
(reserved)				UHCI_ESC_SEQ0_CHAR1	UHCI_ESC_SEQ0_CHAR0	UHCI_ESC_SEQ0	
31	24	23	16	15	8	7	0
0	0	0	0	0	0xd	0xdb	0xdb

Reset

**UHCI\_ESC\_SEQ0** This register is used to define a char that need to be encoded, default is 0xdb that used as the first char of slip escape sequence. (R/W)

**UHCI\_ESC\_SEQ0\_CHAR0** This register is used to define the first char of slip escape sequence when encoding the UHCI\_ESC\_SEQ0, default is 0xdb. (R/W)

**UHCI\_ESC\_SEQ0\_CHAR1** This register is used to define the second char of slip escape sequence when encoding the UHCI\_ESC\_SEQ0, default is 0xdd. (R/W)

**Register 8.53: UHCI\_ESC\_CONF2\_REG (0x00B8)**

UHCI_ESC_SEQ1							
(reserved)				UHCI_ESC_SEQ1_CHAR1	UHCI_ESC_SEQ1_CHAR0	UHCI_ESC_SEQ1	
31	24	23	16	15	8	7	0
0	0	0	0	0	0xde	0xdb	0x11

Reset

**UHCI\_ESC\_SEQ1** This register is used to define a char that need to be encoded, default is 0x11 that used as flow control char. (R/W)

**UHCI\_ESC\_SEQ1\_CHAR0** This register is used to define the first char of slip escape sequence when encoding the UHCI\_ESC\_SEQ1, default is 0xdb. (R/W)

**UHCI\_ESC\_SEQ1\_CHAR1** This register is used to define the second char of slip escape sequence when encoding the UHCI\_ESC\_SEQ1, default is 0xde. (R/W)

**Register 8.54: UHCI\_ESC\_CONF3\_REG (0x00BC)**

UHCI_ESC_SEQ2_CHAR1								UHCI_ESC_SEQ2_CHAR0								UHCI_ESC_SEQ2								
(reserved)								UHCI_ESC_SEQ2_CHAR1								UHCI_ESC_SEQ2_CHAR0								
31	24	23	16	15	8	7	0	0	0	0	0	0	0	0xdff	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	0	0	0	0	0	0	0	0	0	0x13

**UHCI\_ESC\_SEQ2** This register is used to define a char that need to be decoded, default is 0x13 that used as flow control char. (R/W)

**UHCI\_ESC\_SEQ2\_CHAR0** This register is used to define the first char of slip escape sequence when encoding the UHCI\_ESC\_SEQ2, default is 0xdb. (R/W)

**UHCI\_ESC\_SEQ2\_CHAR1** This register is used to define the second char of slip escape sequence when encoding the UHCI\_ESC\_SEQ2, default is 0xdf. (R/W)

**Register 8.55: UHCI\_PKT\_THRES\_REG (0x00C0)**

UHCI_PKT_THRS															
(reserved)															
31	13	12	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	0	0	0	0x80

**UHCI\_PKT\_THRS** This register is used to configure the maximum value of the packet length when UHCI\_HEAD\_EN is 0. (R/W)

## Register 8.56: UHCI\_INT\_RAW\_REG (0x0004)

UHCI_INT_RAW_REG (0x0004)																														
(reserved)																Bit Descriptions														
																Bit Descriptions														
31	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0												
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset
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	0	0	0	0	0	Reset

**UHCI\_RX\_START\_INT\_RAW** This is the interrupt raw bit. Triggered when a separator char has been sent. (RO)

**UHCI\_TX\_START\_INT\_RAW** This is the interrupt raw bit. Triggered when DMA detects a separator char. (RO)

**UHCI\_RX\_HUNG\_INT\_RAW** This is the interrupt raw bit. Triggered when DMA takes more time to receive data than the configure value. (RO)

**UHCI\_TX\_HUNG\_INT\_RAW** This is the interrupt raw bit. Triggered when DMA takes more time to read data from RAM than the configured value. (RO)

**UHCI\_IN\_DONE\_INT\_RAW** This is the interrupt raw bit. Triggered when an receive descriptor is completed. (RO)

**UHCI\_IN\_SUC\_EOF\_INT\_RAW** This is the interrupt raw bit. Triggered when a data packet has been received successfully. (RO)

**UHCI\_IN\_ERR\_EOF\_INT\_RAW** This is the interrupt raw bit. Triggered when there are some errors in EOF in the receive descriptor. (RO)

**UHCI\_OUT\_DONE\_INT\_RAW** This is the interrupt raw bit. Triggered when an transmit descriptor is completed. (RO)

**UHCI\_OUT\_EOF\_INT\_RAW** This is the interrupt raw bit. Triggered when the current descriptor's EOF bit is 1. (RO)

**UHCI\_IN\_DSCR\_ERR\_INT\_RAW** This is the interrupt raw bit. Triggered when there are some errors in the receive descriptor. (RO)

**UHCI\_OUT\_DSCR\_ERR\_INT\_RAW** This is the interrupt raw bit. Triggered when there are some errors in the transmit descriptor. (RO)

**UHCI\_IN\_DSCR\_EMPTY\_INT\_RAW** This is the interrupt raw bit. Triggered when there are not enough inlinks for DMA. (RO)

**UHCI\_OUTLINK\_EOF\_ERR\_INT\_RAW** This is the interrupt raw bit. Triggered when there are some errors in EOF in the transmit descriptor. (RO)

**UHCI\_OUT\_TOTAL\_EOF\_INT\_RAW** This is the interrupt raw bit. Triggered when all data in the last buffer address has been sent out. (RO)

**Continued on the next page...**

**Register 8.56: UHCI\_INT\_RAW\_REG (0x0004)**

Continued from the previous page...

**UHCI\_SEND\_S\_REG\_Q\_INT\_RAW** This is the interrupt raw bit. Triggered when DMA has sent out a short packet using single\_send registers. (RO)

**UHCI\_SEND\_A\_REG\_Q\_INT\_RAW** This is the interrupt raw bit. Triggered when DMA has sent out a short packet using always\_send registers. (RO)

**UHCI\_DMA\_INFIFO\_FULL\_WM\_INT\_RAW** This is the interrupt raw bit. Triggered when the DMA INFIFO count has reached the configured threshold value. (RO)

## Register 8.57: UHCl\_INT\_ST\_REG (0x0008)

31	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	(reserved)														
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	0	0	0	0	0	0	0	0	Reset

**UHCl\_RX\_START\_INT\_ST** This is the masked interrupt bit for UHCl\_RX\_START\_INT interrupt when UHCl\_RX\_START\_INT\_ENA is set to 1. (RO)

**UHCl\_TX\_START\_INT\_ST** This is the masked interrupt bit for UHCl\_TX\_START\_INT interrupt when UHCl\_TX\_START\_INT\_ENA is set to 1. (RO)

**UHCl\_RX\_HUNG\_INT\_ST** This is the masked interrupt bit for UHCl\_RX\_HUNG\_INT interrupt when UHCl\_RX\_HUNG\_INT\_ENA is set to 1. (RO)

**UHCl\_TX\_HUNG\_INT\_ST** This is the masked interrupt bit for UHCl\_TX\_HUNG\_INT interrupt when UHCl\_TX\_HUNG\_INT\_ENA is set to 1. (RO)

**UHCl\_IN\_DONE\_INT\_ST** This is the masked interrupt bit for UHCl\_IN\_DONE\_INT interrupt when UHCl\_IN\_DONE\_INT\_ENA is set to 1. (RO)

**UHCl\_IN\_SUC\_EOF\_INT\_ST** This is the masked interrupt bit for UHCl\_IN\_SUC\_EOF\_INT interrupt when UHCl\_IN\_SUC\_EOF\_INT\_ENA is set to 1. (RO)

**UHCl\_IN\_ERR\_EOF\_INT\_ST** This is the masked interrupt bit for UHCl\_IN\_ERR\_EOF\_INT interrupt when UHCl\_IN\_ERR\_EOF\_INT\_ENA is set to 1. (RO)

**UHCl\_OUT\_DONE\_INT\_ST** This is the masked interrupt bit for UHCl\_OUT\_DONE\_INT interrupt when UHCl\_OUT\_DONE\_INT\_ENA is set to 1. (RO)

**UHCl\_OUT\_EOF\_INT\_ST** This is the masked interrupt bit for UHCl\_OUT\_EOF\_INT interrupt when UHCl\_OUT\_EOF\_INT\_ENA is set to 1. (RO)

**UHCl\_IN\_DSCR\_ERR\_INT\_ST** This is the masked interrupt bit for UHCl\_IN\_DSCR\_ERR\_INT interrupt when UHCl\_IN\_DSCR\_ERR\_INT\_ENA is set to 1. (RO)

**UHCl\_OUT\_DSCR\_ERR\_INT\_ST** This is the masked interrupt bit for UHCl\_OUT\_DSCR\_ERR\_INT interrupt when UHCl\_OUT\_DSCR\_ERR\_INT\_ENA is set to 1. (RO)

**UHCl\_IN\_DSCR\_EMPTY\_INT\_ST** This is the masked interrupt bit for UHCl\_IN\_DSCR\_EMPTY\_INT interrupt when UHCl\_IN\_DSCR\_EMPTY\_INT\_ENA is set to 1. (RO)

**UHCl\_OUT\_OUTLINK\_EOF\_ERR\_INT\_ST** This is the masked interrupt bit for UHCl\_OUT\_OUTLINK\_EOF\_ERR\_INT interrupt when UHCl\_OUT\_OUTLINK\_EOF\_ERR\_INT\_ENA is set to 1. (RO)

**UHCl\_OUT\_TOTAL\_EOF\_INT\_ST** This is the masked interrupt bit for UHCl\_OUT\_TOTAL\_EOF\_INT interrupt when UHCl\_OUT\_TOTAL\_EOF\_INT\_ENA is set to 1. (RO)

**Register 8.57: UHCI\_INT\_ST\_REG (0x0008)**

Continued from the previous page...

**UHCI\_SEND\_S\_REG\_Q\_INT\_ST** This is the masked interrupt bit for UHCI\_SEND\_S\_REG\_Q\_INT interrupt when UHCI\_SEND\_S\_REG\_Q\_INT\_ENA is set to 1. (RO)

**UHCI\_SEND\_A\_REG\_Q\_INT\_ST** This is the masked interrupt bit for UHCI\_SEND\_A\_REG\_Q\_INT interrupt when UHCI\_SEND\_A\_REG\_Q\_INT\_ENA is set to 1. (RO)

**UHCI\_DMA\_INFIFO\_FULL\_WM\_INT\_ST** This is the masked interrupt bit for UHCI\_DMA\_INFIFO\_FULL\_WM\_INT\_INTERRUPT when UHCI\_DMA\_INFIFO\_FULL\_WM\_INT\_ENA is set to 1. (RO)

## Register 8.58: UHCI\_INT\_ENA\_REG (0x000C)

31	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	(reserved)														
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	0	0	0	0	0	0	0	0	Reset

**UHCI\_RX\_START\_INT\_ENA** This is the interrupt enable bit for UHCI\_RX\_START\_INT interrupt. (R/W)

**UHCI\_TX\_START\_INT\_ENA** This is the interrupt enable bit for UHCI\_TX\_START\_INT interrupt. (R/W)

**UHCI\_RX\_HUNG\_INT\_ENA** This is the interrupt enable bit for UHCI\_RX\_HUNG\_INT interrupt. (R/W)

**UHCI\_TX\_HUNG\_INT\_ENA** This is the interrupt enable bit for UHCI\_TX\_HUNG\_INT interrupt. (R/W)

**UHCI\_IN\_DONE\_INT\_ENA** This is the interrupt enable bit for UHCI\_IN\_DONE\_INT interrupt. (R/W)

**UHCI\_IN\_SUC\_EOF\_INT\_ENA** This is the interrupt enable bit for UHCI\_IN\_SUC\_EOF\_INT interrupt. (R/W)

**UHCI\_IN\_ERR\_EOF\_INT\_ENA** This is the interrupt enable bit for UHCI\_IN\_ERR\_EOF\_INT interrupt. (R/W)

**UHCI\_OUT\_DONE\_INT\_ENA** This is the interrupt enable bit for UHCI\_OUT\_DONE\_INT interrupt. (R/W)

**UHCI\_OUT\_EOF\_INT\_ENA** This is the interrupt enable bit for UHCI\_OUT\_EOF\_INT interrupt. (R/W)

**UHCI\_IN\_DSCR\_ERR\_INT\_ENA** This is the interrupt enable bit for UHCI\_IN\_DSCR\_ERR\_INT interrupt. (R/W)

**UHCI\_OUT\_DSCR\_ERR\_INT\_ENA** This is the interrupt enable bit for UHCI\_OUT\_DSCR\_ERR\_INT interrupt. (R/W)

**UHCI\_IN\_DSCR\_EMPTY\_INT\_ENA** This is the interrupt enable bit for UHCI\_IN\_DSCR\_EMPTY\_INT interrupt. (R/W)

**UHCI\_OUT\_TOTAL\_EOF\_INT\_ENA** This is the interrupt enable bit for UHCI\_OUT\_TOTAL\_EOF\_INT interrupt. (R/W)

**UHCI\_SEND\_S\_REG\_Q\_INT\_ENA** This is the interrupt enable bit for UHCI\_SEND\_S\_REG\_Q\_INT interrupt. (R/W)

**Continued on the next page...**

**Register 8.58: UHCI\_INT\_ENA\_REG (0x000C)**

Continued from the previous page...

**UHCI\_SEND\_A\_REG\_Q\_INT\_ENA** This is the interrupt enable bit for UHCI\_SEND\_A\_REG\_Q\_INT interrupt. (R/W)

**UHCI\_DMA\_INFIFO\_FULL\_WM\_INT\_ENA** This is the interrupt enable bit for UHCI\_DMA\_INFIFO\_FULL\_WM\_INT interrupt. (R/W)

## Register 8.59: UHCI\_INT\_CLR\_REG (0x0010)

31	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	(reserved)														
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	0	0	0	0	0	0	0	0	Reset

**UHCI\_RX\_START\_INT\_CLR** Set this bit to clear UHCI\_RX\_START\_INT interrupt. (WO)

**UHCI\_TX\_START\_INT\_CLR** Set this bit to clear UHCI\_TX\_START\_INT interrupt. (WO)

**UHCI\_RX\_HUNG\_INT\_CLR** Set this bit to clear UHCI\_RX\_HUNG\_INT interrupt. (WO)

**UHCI\_TX\_HUNG\_INT\_CLR** Set this bit to clear UHCI\_TX\_HUNG\_INT interrupt. (WO)

**UHCI\_IN\_DONE\_INT\_CLR** Set this bit to clear UHCI\_IN\_DONE\_INT interrupt. (WO)

**UHCI\_IN\_SUC\_EOF\_INT\_CLR** Set this bit to clear UHCI\_IN\_SUC\_EOF\_INT interrupt. (WO)

**UHCI\_IN\_ERR\_EOF\_INT\_CLR** Set this bit to clear UHCI\_IN\_ERR\_EOF\_INT interrupt. (WO)

**UHCI\_OUT\_DONE\_INT\_CLR** Set this bit to clear UHCI\_OUT\_DONE\_INT interrupt. (WO)

**UHCI\_OUT\_EOF\_INT\_CLR** Set this bit to clear UHCI\_OUT\_EOF\_INT interrupt. (WO)

**UHCI\_IN\_DSCR\_ERR\_INT\_CLR** Set this bit to clear UHCI\_IN\_DSCR\_ERR\_INT interrupt. (WO)

**UHCI\_OUT\_DSCR\_ERR\_INT\_CLR** Set this bit to clear UHCI\_OUT\_DSCR\_ERR\_INT interrupt. (WO)

**UHCI\_IN\_DSCR\_EMPTY\_INT\_CLR** Set this bit to clear UHCI\_IN\_DSCR\_EMPTY\_INT interrupt. (WO)

**UHCI\_OUTLINK\_EOF\_ERR\_INT\_CLR** Set this bit to clear UHCI\_OUTLINK\_EOF\_ERR\_INT interrupt. (WO)

**UHCI\_OUT\_TOTAL\_EOF\_INT\_CLR** Set this bit to clear UHCI\_OUT\_TOTAL\_EOF\_INT interrupt. (WO)

**UHCI\_SEND\_S\_REG\_Q\_INT\_CLR** Set this bit to clear UHCI\_SEND\_S\_UHCI\_Q\_INT interrupt. (WO)

**UHCI\_SEND\_A\_REG\_Q\_INT\_CLR** Set this bit to clear UHCI\_SEND\_A\_UHCI\_Q\_INT interrupt. (WO)

**UHCI\_DMA\_INFIFO\_FULL\_WM\_INT\_CLR** Set this bit to clear UHCI\_DMA\_INFIFO\_FULL\_WM\_INT interrupt. (WO)

**Register 8.60: UHCI\_DMA\_OUT\_STATUS\_REG (0x0014)**

(reserved)																															
31																															

**UHCI\_OUT\_FULL** 1: DMA data-output FIFO is full. (RO)

**UHCI\_OUT\_EMPTY** 1: DMA data-output FIFO is empty. (RO)

**Register 8.61: UHCI\_DMA\_IN\_STATUS\_REG (0x001C)**

(reserved)																																
31																																

**UHCI\_IN\_FULL** Data-input FIFO full signal. (RO)

**UHCI\_IN\_EMPTY** Data-input FIFO empty signal. (RO)

**UHCI\_RX\_ERR\_CAUSE** This register indicates the error type when DMA has received a packet with error. 3'b001: Checksum error in HCI packet; 3'b010: Sequence number error in HCI packet; 3'b011: CRC bit error in HCI packet; 3'b100: 0xc0 is found but received HCI packet is not end; 3'b101: 0xc0 is not found when receiving HCI packet is end; 3'b110: CRC check error. (RO)

## Register 8.62: UHCI\_STATE0\_REG (0x0030)

(reserved)	UHCI_DECODE_STATE	UHCI_INFIFO_CNT_DEBUG	UHCI_IN_STATE	UHCI_IN_DSCR_STATE	UHCI_INLINK_DSCR_ADDR						
31	30	28	27	23	22	20	19	18	17	0	
0	0	0	0	0	0	0	0	0	0	0	Reset

**UHCI\_INLINK\_DSCR\_ADDR** This register stores the current receive descriptor's address. (RO)

**UHCI\_IN\_DSCR\_STATE** reserved (RO)

**UHCI\_IN\_STATE** reserved (RO)

**UHCI\_INFIFO\_CNT\_DEBUG** This register stores the byte number of the data in the receive descriptor's FIFO. (RO)

**UHCI\_DECODE\_STATE** UHCI decoder status. (RO)

## Register 8.63: UHCI\_DMA\_OUT\_EOF\_DES\_ADDR\_REG (0x0038)

31	0	
0x0000000	0	Reset

**UHCI\_OUT\_EOF\_DES\_ADDR** This register stores the address of the transmit descriptor when the EOF bit in this descriptor is 1. (RO)

## Register 8.64: UHCI\_DMA\_IN\_SUC\_EOF\_DES\_ADDR\_REG (0x003C)

31	0	
0x0000000	0	Reset

**UHCI\_IN\_SUC\_EOF\_DES\_ADDR** This register stores the address of the receive descriptor when received successful EOF. (RO)

**Register 8.65: UHCI\_DMA\_IN\_ERR\_EOF\_DES\_ADDR\_REG (0x0040)**

UHCI_IN_ERR_EOF_DES_ADDR	
31	0
0x000000	Reset

**UHCI\_IN\_ERR\_EOF\_DES\_ADDR** This register stores the address of the receive descriptor when there are some errors in this descriptor. (RO)

**Register 8.66: UHCI\_DMA\_OUT\_EOF\_BFR\_DES\_ADDR\_REG (0x0044)**

UHCI_OUT_EOF_BFR_DES_ADDR	
31	0
0x000000	Reset

**UHCI\_OUT\_EOF\_BFR\_DES\_ADDR** This register stores the address of the transmit descriptor before the last transmit descriptor. (RO)

**Register 8.67: UHCI\_DMA\_IN\_DSCR\_REG (0x004C)**

UHCI_INLINK_DSCR	
31	0
0	Reset

**UHCI\_INLINK\_DSCR** This register stores the third word of the next receive descriptor. (RO)

**Register 8.68: UHCI\_DMA\_IN\_DSCR\_BF0\_REG (0x0050)**

UHCI_INLINK_DSCR_BF0	
31	0
0	Reset

**UHCI\_INLINK\_DSCR\_BF0** This register stores the third word of the current receive descriptor. (RO)

**Register 8.69: UHCI\_DMA\_IN\_DSCR\_BF1\_REG (0x0054)**

UHCI_INLINK_DSCR_BF1	
31	0
0	Reset

**UHCI\_INLINK\_DSCR\_BF1** This register stores the second word of the current receive descriptor. (RO)

**Register 8.70: UHCI\_DMA\_OUT\_DSCR\_REG (0x0058)**

UHCI_OUTLINK_DSCR	
31	0
0	Reset

**UHCI\_OUTLINK\_DSCR** This register stores the third word of the next transmit descriptor. (RO)

**Register 8.71: UHCI\_DMA\_OUT\_DSCR\_BF0\_REG (0x005C)**

UHCI_OUTLINK_DSCR_BF0	
31	0
0	Reset

**UHCI\_OUTLINK\_DSCR\_BF0** This register stores the third word of the current transmit descriptor.  
(RO)

**Register 8.72: UHCI\_DMA\_OUT\_DSCR\_BF1\_REG (0x0060)**

UHCI_OUTLINK_DSCR_BF1	
31	0
0	Reset

**UHCI\_OUTLINK\_DSCR\_BF1** This register stores the second word of the current transmit descriptor.  
(RO)

**Register 8.73: UHCI\_RX\_HEAD\_REG (0x0070)**

UHCI_RX_HEAD	
31	0
0x000000	Reset

**UHCI\_RX\_HEAD** This register stores the header of the current received packet. (RO)

**Register 8.74: UHCI\_DMA\_OUT\_PUSH\_REG (0x0018)**

31		17	16	15		9	8	0
0	0	0	0	0	0	0	0	0x0 Reset

**UHCI\_OUTFIFO\_WDATA** This is the data that need to be pushed into data-output FIFO. (R/W)

**UHCI\_OUTFIFO\_PUSH** Set this bit to push data into the data-output FIFO. (R/W)

**Register 8.75: UHCI\_DMA\_IN\_POP\_REG (0x0020)**

31		17	16	15		12	11	0
0	0	0	0	0	0	0	0	0x0 Reset

**UHCI\_INFIFO\_RDATA** This register stores the data popping from data-input FIFO. (RO)

**UHCI\_INFIFO\_POP** Set this bit to pop data from data-input FIFO. (R/W)

**Register 8.76: UHCI\_DMA\_OUT\_LINK\_REG (0x0024)**

31	30	29	28	27		20	19	0
0	0	0	0	0	0	0	0	0x000 Reset

**UHCI\_OUTLINK\_ADDR** This register is used to specify the least significant 20 bits of the first transmit descriptor's address. (R/W)

**UHCI\_OUTLINK\_STOP** Set this bit to stop dealing with the transmit descriptor. (R/W)

**UHCI\_OUTLINK\_START** Set this bit to start a new transmit descriptor. (R/W)

**UHCI\_OUTLINK\_RESTART** Set this bit to restart the transmit descriptor from the last address. (R/W)

**UHCI\_OUTLINK\_PARK** 1: the transmit descriptor's FSM is in idle state. 0: the transmit descriptor's FSM is working. (RO)

## Register 8.77: UHCI\_DMA\_IN\_LINK\_REG (0x0028)

31	30	29	28	27	21	20	19	0
0	0	0	0	0	0	0	0	0x000

Reset

**UHCI\_INLINK\_ADDR** This register is used to specify the least significant 20 bits of the first receive descriptor's address. (R/W)

**UHCI\_INLINK\_AUTO\_RET** This is the enable bit to return to current receive descriptor's address, when there are some errors in current packet. (R/W)

**UHCI\_INLINK\_STOP** Set this bit to stop dealing with the receive descriptors. (R/W)

**UHCI\_INLINK\_START** Set this bit to start dealing with the receive descriptors. (R/W)

**UHCI\_INLINK\_RESTART** Set this bit to restart new receive descriptors. (R/W)

**UHCI\_INLINK\_PARK** 1: the receive descriptor's FSM is in idle state. 0: the receive descriptor's FSM is working. (RO)

## Register 8.78: UHCI\_STATE1\_REG (0x0034)

31	30	28	27	23	22	20	19	18	17	0
0	0	0	0	0	0	0	0	0	0	0x000

Reset

**UHCI\_OUTLINK\_DSCR\_ADDR** This register stores the current transmit descriptor's address. (RO)

**UHCI\_OUT\_DSCR\_STATE** reserved (RO)

**UHCI\_OUT\_STATE** reserved (RO)

**UHCI\_OUTFIFO\_CNT** This register stores the byte number of the data in the transmit descriptor's FIFO. (RO)

**UHCI\_ENCODE\_STATE** UHCI encoder status. (RO)

**Register 8.79: UHCI\_DATE\_REG (0x00FC)**

UHCI_DATE	
31	0
0x18073001	Reset

**UHCI\_DATE** This is the version control register. (R/W)

## 9. LED PWM Controller

### 9.1 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.

### 9.2 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  $\text{PWM}_n$ , and the four timers are collectively referred to as  $\text{Timer}_x$ .

### 9.3 Functional Description

#### 9.3.1 Architecture

Figure 9-1 shows the architecture of the LED PWM controller. Figure 9-2 illustrates a PWM generator with its selected timer and a counter.

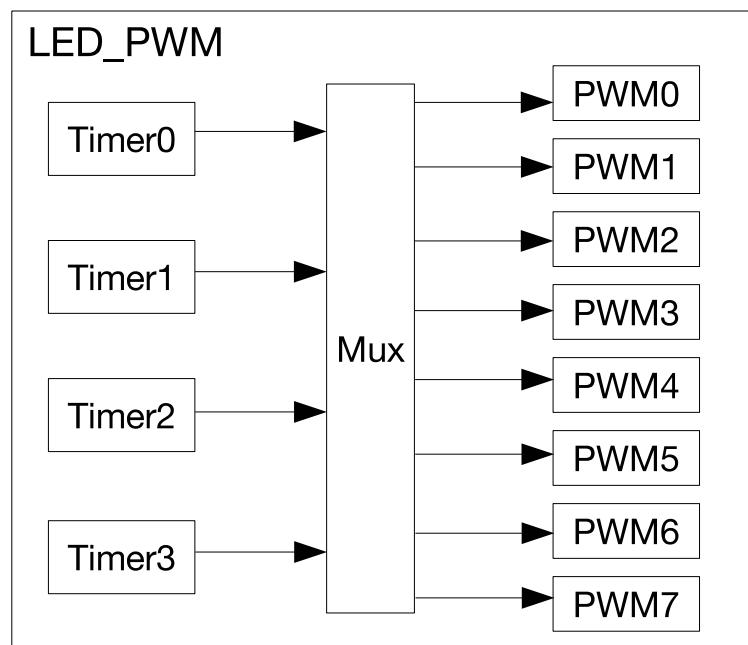


Figure 9-1. LED\_PWM Architecture

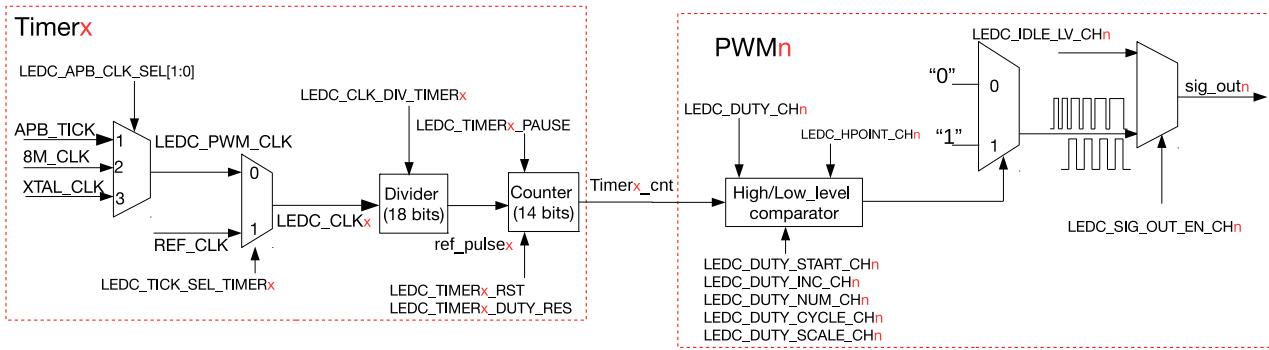


Figure 9-2. LED\_PWM generator Diagram

### 9.3.2 Timers

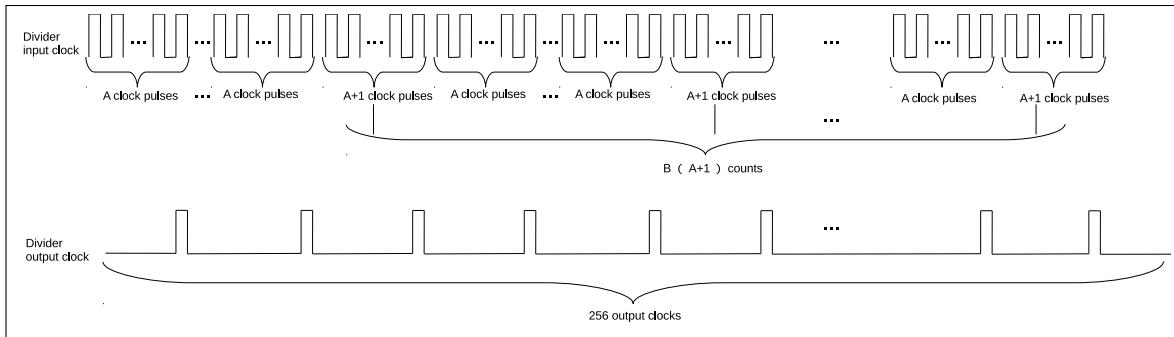


Figure 9-3. LED\_PWM Divider

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_CLKx`, 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_CLKx` is used as the base clock for the counter. The divider's divisor is configured by `LEDC_CLK_DIV_TIMERx`. 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_DIVx` 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 9-3. 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_TIMERx_OVF_INT` interrupt can be generated every time when the counter overflows or when it overflows for  $(LEDC\_OVF\_NUM\_CHn + 1)$  times. The interrupt configuration is as follows:

1. Set `LEDC_OVF_CNT_EN_CHn`

2. Configure `LEDC_OVF_NUM_CHn` with the times of overflow minus 1
3. Set `LEDC_OVF_CNT_CHn_INT_ENA`
4. Set `LEDC_TIMERx_DUTY_RES` to enable the timer and wait for a `LEDC_OVF_CNT_CHn_INT` interrupt

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

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

To change the divisor and times of overflow, you should configure `LEDC_CLK_DIV_TIMERx` and `LEDC_TIMERx_DUTY_RES` respectively, and then set `LEDC_TIMERx_PARA_UP`; otherwise this change is not valid. The newly configured value is updated upon next overflow of the counter.

### 9.3.3 PWM Generators

As shown in figure 9-2, 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 `Timerx_cnt == Hpointn`, `sig_outn` is 1.
- If `Timerx_cnt == Lpointn`, `sig_outn` is 0.

`Hpointn` is updated by `LEDC_HPOINT_CHn` when the counter overflows. The initial value of `Lpointn` is the sum of `LEDC_DUTY_CHn[18..4]` and `LEDC_HPOINT_CHn` when the counter overflows. By configuring these two fields, the relative phase and the duty cycle of the PWM output can be set.

Figure 9-4 illustrates PWM's waveform when the duty cycle is fixed.

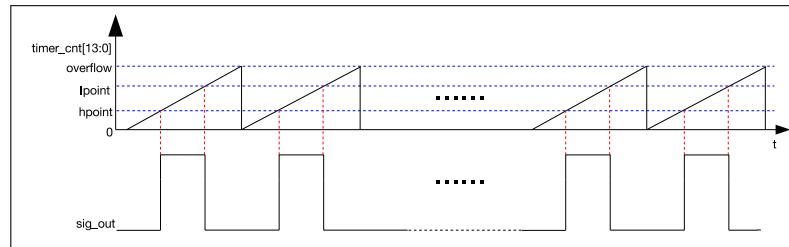


Figure 9-4. LED\_PWM Output Signal Diagram

`LEDC_DUTY_CHn` is a fixed-point register with four fractional bits. `LEDC_DUTY_CHn[18..4]` is the integral part used directly for PWM calculation. `LEDC_DUTY_CHn[3..0]` is the fractional part used to dither the output. If `LEDC_DUTY_CHn[3..0]` is non-zero, then among every 16 cycles of `sig_outn`, `LEDC_DUTY_CHn[3..0]` have PWM pulses with width one timer cycle longer than that of  $(16 \cdot \text{LEDC\_DUTY\_CH}_n[3..0])$ . This feature effectively increases the resolution of the PWM generator to 18 bits.

### 9.3.4 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_CHn`, `LEDC_DUTY_START_CHn`, `LEDC_DUTY_INC_CHn`, `LEDC_DUTY_NUM_CHn` and `LEDC_DUTY_SCALE_CHn`.

`LEDC_DUTY_START_CHn` is used to update the value of `Lpointn`. When this bit is set and the counter overflows, `Lpointn` increments or decrements automatically, depending on whether the bit `LEDC_DUTY_INC_CHn` is set or cleared.

The duty cycle of `sig_outn` changes every `LEDC_DUTY_CYCLE_CHn` PWM pulse cycles by adding or subtracting the value of `LEDC_DUTY_SCALE_CHn`.

Figure 9-5 is a diagram of fading duty cycle. Upon reaching `LEDC_DUTY_NUM_CHn`, the fade stops and a

`LEDC_DUTY_CHNG_END_CHn_INT` interrupt is generated. When configured like this, the duty cycle of `sig_outn` increases by `LEDC_DUTY_SCALE_CHn` every `LEDC_DUTY_CYCLE_CHn` PWM pulse cycles.

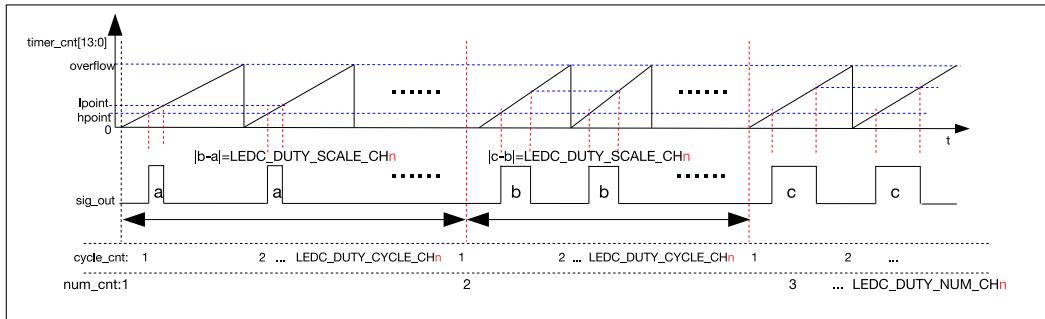


Figure 9-5. Output Signal Diagram of Fading Duty Cycle

`LEDC_SIG_OUT_EN_CHn` used to enable PWM waveform output. When `LEDC_SIG_OUT_EN_CHn` is 0, the level of `sig_outn` is constant as specified in `LEDC_IDLE_LV_CHn`.

If `LEDC_HPOINT_CHn`, `LEDC_DUTY_START_CHn`, `LEDC_SIG_OUT_EN_CHn`, `LEDC_TIMER_SEL_CHn`, `LEDC_DUTY_NUM_CHn`, `LEDC_DUTY_CYCLE_CHn`, `LEDC_DUTY_SCALE_CHn`, `LEDC_DUTY_INC_CHn` and

`LEDC_OVF_CNT_EN_CHn` are reconfigured, `LEDC_PARA_UP_CHn` should be set to apply the new configuration.

### 9.3.5 Interrupts

- `LEDC_OVF_CNT_CHn_INT`: Triggered when the timer counter overflows for (`LEDC_OVF_NUM_CHn` + 1) times and register `LEDC_OVF_CNT_EN_CHn` is set to 1.
- `LEDC_DUTY_CHNG_END_CHn_INT`: Triggered when a fade on a LED PWM generator has finished.
- `LEDC_TIMERx_OVF_INT`: Triggered when a LED PWM timer has reached its maximum counter value.

## 9.4 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 39: LED\_PWM Base Address**

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

## 9.5 Register Summary

The addresses in the following table are relative to the LED PWM controller base addresses provided in Section 9.4.

Name	Description	Address	Access
<b>Configuration Register</b>			
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
<b>Hpoint Register</b>			
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
<b>Duty Cycle Register</b>			
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
<b>Timer Register</b>			
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
<b>Interrupt Register</b>			
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
<b>Version Register</b>			
LEDC_DATE_REG	Version control register	0x00FC	R/W

## 9.6 Registers

Register 9.1: **LEDC\_CH $n$ \_CONF0\_REG** ( $n$ : 0-7) (0x0000+20\* $n$ )

31	18	17	16	15	14		5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0x0

Reset

**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 times of overflow minus 1. The LEDC\_OVF\_CNT\_CH $n$ \_INT interrupt will be triggered when channel  $n$  overflows for (LEDC\_OVF\_NUM\_CH $n$  + 1) times. (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 9.2: LEDC\_CH<sub>n</sub>\_CONF1\_REG (<sub>n</sub>: 0-7) (0x000C+20\*<sub>n</sub>)**

31	30	29	20	19	10	9	0
0	1	0x0		0x0		0x0	Reset

**LEDC\_DUTY\_SCALE\_CH<sub>n</sub>** This register is used to configure the changing step scale of duty on channel <sub>n</sub>. (R/W)

**LEDC\_DUTY\_CYCLE\_CH<sub>n</sub>** The duty will change every LEDC\_DUTY\_CYCLE\_CH<sub>n</sub> on channel <sub>n</sub>. (R/W)

**LEDC\_DUTY\_NUM\_CH<sub>n</sub>** This register is used to control the number of times the duty cycle will be changed. (R/W)

**LEDC\_DUTY\_INC\_CH<sub>n</sub>** This register is used to increase or decrease the duty of output signal on channel <sub>n</sub>. 1: Increase; 0: Decrease. (R/W)

**LEDC\_DUTY\_START\_CH<sub>n</sub>** Other configured fields in LEDC\_CH<sub>n</sub>\_CONF1\_REG will start to take effect when this bit is set to 1. (R/W)

**Register 9.3: LEDC\_CONF\_REG (0x00D0)**

31	30	(reserved)	2	1	0
0	0	0 0	0x0	Reset	

**LEDC\_APB\_CLK\_SEL** This bit is used to select clock source for 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 9.4: LEDC\_CH $n$ \_HPOINT\_REG ( $n$ : 0-7) (0x0004+20\* $n$ )**

31				14	13	0
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0				0x00	Reset

**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 9.5: LEDC\_CH $n$ \_DUTY\_REG ( $n$ : 0-7) (0x0008+20\* $n$ )**

31				19	18	0
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0				0x000	Reset

**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 9.6: LEDC\_CH $n$ \_DUTY\_R\_REG ( $n$ : 0-7) (0x0010+20\* $n$ )**

31				19	18	0
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0				0x000	Reset

**LEDC\_DUTY\_R\_CH $n$**  This register stores the current duty of output signal on channel  $n$ . (RO)

**Register 9.7: LEDC\_TIMER<sub>x</sub>\_CONF\_REG (x: 0-3) (0x00A0+8\*x)**

The diagram shows the bit field layout of Register 9.7. The register is 32 bits wide. Bits 31 to 21 are labeled '(reserved)'. Bits 20 to 16 are labeled 'LEDC\_TIMER<sub>x</sub>\_PARA\_UP'. Bits 15 to 14 are labeled 'LEDC\_TICK\_SEL\_TIMER<sub>x</sub>'. Bits 13 to 12 are labeled 'LEDC\_TIMER<sub>x</sub>\_RST'. Bits 11 to 10 are labeled 'LEDC\_TIMER<sub>x</sub>\_PAUSE'. Bits 9 to 8 are labeled 'LEDC\_CLK\_DIV\_TIMER<sub>x</sub>'. Bits 7 to 4 are labeled 'LEDC\_TIMER<sub>x</sub>\_DUTY\_RES'. Bit 3 is labeled 'Reset'.

31	26	25	24	23	22	21	4	3	0
0	0	0	0	0	0	0	1	0	0x000

0x000      0x0      Reset

**LEDC\_TIMER<sub>x</sub>\_DUTY\_RES** This register is used to control the range of the counter in timer  $x$ . (R/W)

**LEDC\_CLK\_DIV\_TIMER<sub>x</sub>** 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\_TIMER<sub>x</sub>\_PAUSE** This bit is used to suspend the counter in timer  $x$ . (R/W)

**LEDC\_TIMER<sub>x</sub>\_RST** This bit is used to reset timer  $x$ . The counter will show 0 after reset. (R/W)

**LEDC\_TICK\_SEL\_TIMER<sub>x</sub>** 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\_TIMER<sub>x</sub>\_PARA\_UP** Set this bit to update LEDC\_CLK\_DIV\_TIMER<sub>x</sub> and LEDC\_TIMER<sub>x</sub>\_DUTY\_RES. (WO)

**Register 9.8: LEDC\_TIMER<sub>x</sub>\_VALUE\_REG (x: 0-3) (0x00A4+8\*x)**

The diagram shows the bit field layout of Register 9.8. The register is 32 bits wide. Bits 31 to 21 are labeled '(reserved)'. Bits 20 to 16 are labeled 'LEDC\_TIMER<sub>x</sub>\_CNT'. Bits 15 to 14 are labeled 'LEDC\_TIMER<sub>x</sub>\_CNT'. Bits 13 to 0 are labeled 'LEDC\_TIMER<sub>x</sub>\_CNT'. Bit 0 is labeled 'Reset'.

31	14	13	0
0	0	0	0

0x00      0x0      Reset

**LEDC\_TIMER<sub>x</sub>\_CNT** This register stores the current counter value of timer  $x$ . (RO)

**Register 9.9: LEDC\_INT\_RAW\_REG (0x00C0)**

(reserved)																					
31	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**LEDC\_TIMER<sub>x</sub>\_OVF\_INT\_RAW** Triggered when the timer<sub>x</sub> has reached its maximum counter value.  
(RO)

**LEDC\_DUTY\_CHNG\_END\_CH<sub>n</sub>\_INT\_RAW** Interrupt raw bit for channel *n*. Triggered when the gradual change of duty has finished. (RO)

**LEDC\_OVF\_CNT\_CH<sub>n</sub>\_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 9.10: LEDC\_INT\_ST\_REG (0x00C4)**

(reserved)																					
31	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**LEDC\_TIMER<sub>x</sub>\_OVF\_INT\_ST** This is the masked interrupt status bit for the LEDC\_TIMER<sub>x</sub>\_OVF\_INT interrupt when LEDC\_TIMER<sub>x</sub>\_OVF\_INT\_ENA is set to 1. (RO)

**LEDC\_DUTY\_CHNG\_END\_CH<sub>n</sub>\_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\_ENA is set to 1. (RO)

**LEDC\_OVF\_CNT\_CH<sub>n</sub>\_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 9.11: LEDC\_INT\_ENA\_REG (0x00C8)

(reserved)																															
31	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	0	0	
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	0	0	0	0	0	0	0

**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 9.12: LEDC\_INT\_CLR\_REG (0x00CC)

(reserved)																															
31	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	0	0	
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	0	0	0	0	0	0	0

**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 9.13: LEDC\_DATE\_REG (0x00FC)**

LEDC_DATE	
31	0
0x19072601	Reset

**LEDC\_DATE** This is the version control register. (R/W)

# 10. Remote Control Peripheral

## 10.1 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.

## 10.2 Functional Description

### 10.2.1 RMT Architecture

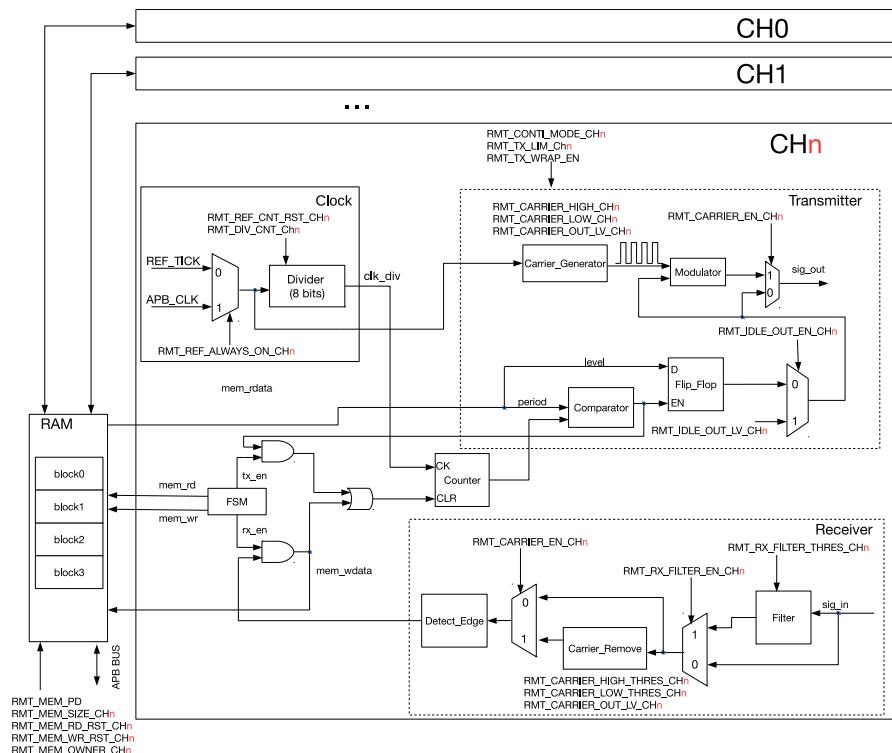


Figure 10-1. RMT Architecture

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.

## 10.2.2 RMT RAM

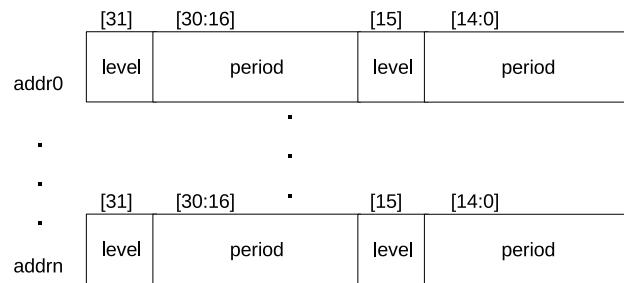


Figure 10-2. Format of Pulse Code in RAM

The format of pulse code in RAM is shown in Figure 10-2. 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 10-1 clk\_div) for which the level lasts. A zero period is interpreted as a transmission end-marker.

The RAM is divided into four  $64 \times 32$ -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  $64 \times 32$ -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  $(n) \sim$  block  $(n + RMT\_MEM\_SIZE\_CHn - 1)$ . If so, the subsequent channels  $n + 1 \sim n + RMT\_MEM\_SIZE\_CHn - 1$  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`.

## 10.2.3 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 10-2).

## 10.2.4 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.

### 10.2.5 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\\_THRESH\\_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\\_THRESH\\_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\\_THRESH\\_CH \$n\$](#)  APB clock cycles.

### 10.2.6 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\$](#) .

## 10.3 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 41: RMT Base Address**

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

## 10.4 Register Summary

The addresses in the following table are relative to RMT base addresses provided in Section 10.3.

Name	Description	Address	Access
<b>Configuration registers</b>			
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
<b>Carrier wave duty cycle registers</b>			

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
<b>Tx event configuration registers</b>			
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
<b>Status registers</b>			
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
<b>Version register</b>			
RMT_DATE_REG	Version control register	0x00FC	R/W
<b>FIFO R/W registers</b>			
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
<b>Interrupt registers</b>			
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

## 10.5 Registers

**Register 10.1: RMT\_CH $n$ CONF0\_REG ( $n$ : 0-3) (0x0010+8\* $n$ )**

31	30	29	28	27	26	24	23	8	7	0	Reset
0	0	1	1	1	0x1		0x1000		0x2		

**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 10.2: RMT\_CH<sub>n</sub>CONF1\_REG (*n*: 0-3) (0x0014+8\**n*)

The diagram shows the bit field layout for the RMT\_CH<sub>n</sub>CONF1\_REG register. The bits are numbered from 31 down to 0. Bit 31 is labeled '(reserved)'. Bits 21 to 15 are grouped under the label 'RMT\_TX\_STOP\_CH<sub>n</sub>'. Bits 20 to 17 are grouped under 'RMT\_IDLE\_OUT\_EN\_CH<sub>n</sub>'. Bits 18 to 16 are grouped under 'RMT\_IDLE\_OUT\_LV\_CH<sub>n</sub>'. Bits 15 to 14 are grouped under 'RMT\_REF\_ALWAYS\_ON\_CH<sub>n</sub>'. Bits 13 to 12 are grouped under 'RMT\_CHK\_RX\_CARRIER\_EN\_CH<sub>n</sub>'. Bits 8 to 0 are grouped under 'RMT\_RX\_FILTER\_THRES\_CH<sub>n</sub>'. Bits 7 to 6 are grouped under 'RMT\_RX\_FILTER\_EN\_CH<sub>n</sub>'. Bits 5 to 4 are grouped under 'RMT\_TX\_CONTI\_MODE\_CH<sub>n</sub>'. Bits 3 to 2 are grouped under 'RMT\_MEM\_OWNER\_CH<sub>n</sub>'. Bits 1 to 0 are grouped under 'RMT\_APB\_MEM\_RD\_RST\_CH<sub>n</sub>' and 'RMT\_MEM\_WR\_RST\_CH<sub>n</sub>'. Bit 0 is labeled 'Reset'.

31	21	20	19	18	17	16	15		8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0xf	0	0	1	0	0	0	0	0	Reset

**RMT\_TX\_START\_CH<sub>n</sub>** Set this bit to start sending data on channel *n*. (R/W)

**RMT\_RX\_EN\_CH<sub>n</sub>** Set this bit to enable receiver to receive data on channel *n*. (R/W)

**RMT\_MEM\_WR\_RST\_CH<sub>n</sub>** Set this bit to reset RAM write address accessed by the receiver for channel *n*. (WO)

**RMT\_MEM\_RD\_RST\_CH<sub>n</sub>** Set this bit to reset RAM read address accessed by the transmitter for channel *n*. (WO)

**RMT\_MEM\_OWNER\_CH<sub>n</sub>** 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<sub>n</sub>** Set this bit to restart transmission in continuous mode from the first data in channel *n*. (R/W)

**RMT\_RX\_FILTER\_EN\_CH<sub>n</sub>** Set this bit to enable the receiver's filter for channel *n*. (R/W)

**RMT\_RX\_FILTER\_THRES\_CH<sub>n</sub>** Set this field to ignore the input pulse when its width is less than RMT\_RX\_FILTER\_THRES\_CH<sub>n</sub> APB clock cycles in receive mode. (R/W)

**RMT\_CHK\_RX\_CARRIER\_EN\_CH<sub>n</sub>** Set this bit to enable memory loop read mode when carrier modulation is enabled for channel *n*. (R/W)

**RMT\_REF\_ALWAYS\_ON\_CH<sub>n</sub>** 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<sub>n</sub>** 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<sub>n</sub>** This is the output enable bit for channel *n* in idle state. (R/W)

**RMT\_TX\_STOP\_CH<sub>n</sub>** Set this bit to stop the transmitter of channel *n* sending data out. (R/W)

### Register 10.3: 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 10.4: RMT\_REF\_CNT\_RST\_REG (0x0088)

**RMT\_REF\_CNT\_RST\_CHn** This bit is used to reset the clock divider of channel *n*. (R/W)

**Register 10.5: RMT\_CH $n$ \_RX\_CARRIER\_RM\_REG ( $n$ : 0-3) (0x008C+4\* $n$ )**

The diagram shows two fields in Register 10.5. The first field, labeled "RMT\_CARRIER\_HIGH\_THRES\_CH $n$ ", is located at the top left. The second field, labeled "RMT\_CARRIER\_LOW\_THRES\_CH $n$ ", is located at the top right. Both fields are represented by diagonal lines pointing towards the center of the register structure.

31	16	15	0
0x00		0x00	Reset

**RMT\_CARRIER\_LOW\_THRES\_CH $n$**  The low level period in carrier modulation mode is  $(RMT\_CARRIER\_LOW\_THRES\_CHn + 1)$  clock cycles for channel  $n$ . (R/W)

**RMT\_CARRIER\_HIGH\_THRES\_CH $n$**  The high level period in carrier modulation mode is  $(RMT\_CARRIER\_HIGH\_THRES\_CHn + 1)$  clock cycles for channel  $n$ . (R/W)

**Register 10.6: RMT\_CH $n$ CARRIER\_DUTY\_REG ( $n$ : 0-3) (0x0060+4\* $n$ )**

The diagram shows two fields in Register 10.6. The first field, labeled "RMT\_CARRIER\_HIGH\_CH $n$ ", is located at the top left. The second field, labeled "RMT\_CARRIER\_LOW\_CH $n$ ", is located at the top right. Both fields are represented by diagonal lines pointing towards the center of the register structure.

31	16	15	0
0x40		0x40	Reset

**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 10.7: RMT\_CH<sub>n</sub>\_TX\_LIM\_REG (*n*: 0-3) (0x0070+4\**n*)**

31	21	20	19	18	9	8	0
0	0	0	0	0	0	0	0x80 Reset

**RMT\_TX\_LIM\_CH<sub>n</sub>** This field is used to configure the maximum entries channel *n* can send out.

When **RMT\_MEM\_SIZE\_CH<sub>n</sub>** = 1, this field can be set to any value among 0 ~ 128 (64 \* 32/16 = 128); when **RMT\_MEM\_SIZE\_CH<sub>n</sub>** > 1, this field can be set to any value among (0 ~ 128) \* **RMT\_MEM\_SIZE\_CH<sub>n</sub>**. (R/W)

**RMT\_TX\_LOOP\_NUM\_CH<sub>n</sub>** This field is used to configure the maximum loop times when continuous transmission mode is enabled. (R/W)

**RMT\_TX\_LOOP\_CNT\_EN\_CH<sub>n</sub>** This bit is used to enable loop counting. (R/W)

**RMT\_LOOP\_COUNT\_RESET\_CH<sub>n</sub>** This bit is used to reset loop counting when continuous transmission mode is valid. (WO)

**Register 10.8: RMT\_TX\_SIM\_REG (0x0084)**

31	5	4	3	2	1	0	
0	0	0	0	0	0	0	0x00000000 Reset

**RMT\_TX\_SIM\_CH<sub>n</sub>** 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 10.9: RMT\_CH $n$ STATUS\_REG ( $n$ : 0-3) (0x0030+4\* $n$ )**

The diagram shows the bit field layout of Register 10.9. The register is 32 bits wide, with bit 31 at the top and bit 0 at the bottom. The fields are labeled as follows:

- Bit 31: (reserved)
- Bits 28-27: (reserved)
- Bit 26: RMT\_APB\_MEM\_RD\_ERR\_CH $n$
- Bit 25: RMT\_APB\_MEM\_WR\_ERR\_CH $n$
- Bit 24: RMT\_MEM\_EMPTY\_CH $n$
- Bit 23: RMT\_MEM\_FULL\_CH $n$
- Bit 22: RMT\_MEM\_OWNER\_ERR\_CH $n$
- Bit 21: RMT\_STATE\_CH $n$
- Bit 20: (reserved)
- Bit 19: (reserved)
- Bit 18: (reserved)
- Bit 17: RMT\_MEM\_RADDR\_EX\_CH $n$
- Bit 16: (reserved)
- Bit 15: (reserved)
- Bit 14: RMT\_MEM\_WADDR\_EX\_CH $n$
- Bit 13: (reserved)
- Bit 12: (reserved)
- Bit 11: (reserved)
- Bit 10: (reserved)
- Bit 9: (reserved)
- Bit 8: (reserved)
- Bit 7: (reserved)
- Bit 6: (reserved)
- Bit 5: (reserved)
- Bit 4: (reserved)
- Bit 3: (reserved)
- Bit 2: (reserved)
- Bit 1: (reserved)
- Bit 0: (reserved)

Reset value: 0x0

**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 10.10: RMT\_CH $n$ ADDR\_REG ( $n$ : 0-3) (0x0040+4\* $n$ )**

The diagram shows the bit field layout of Register 10.10. The register is 32 bits wide, with bit 31 at the top and bit 0 at the bottom. The fields are labeled as follows:

- Bit 31: (reserved)
- Bit 29: (reserved)
- Bit 28: (reserved)
- Bit 27: (reserved)
- Bit 26: (reserved)
- Bit 25: (reserved)
- Bit 24: (reserved)
- Bit 23: (reserved)
- Bit 22: (reserved)
- Bit 21: (reserved)
- Bit 20: (reserved)
- Bit 19: (reserved)
- Bit 18: (reserved)
- Bit 17: RMT\_APB\_MEM\_RADDR\_CH $n$
- Bit 16: (reserved)
- Bit 15: (reserved)
- Bit 14: (reserved)
- Bit 13: (reserved)
- Bit 12: (reserved)
- Bit 11: (reserved)
- Bit 10: (reserved)
- Bit 9: (reserved)
- Bit 8: (reserved)
- Bit 7: (reserved)
- Bit 6: (reserved)
- Bit 5: (reserved)
- Bit 4: (reserved)
- Bit 3: (reserved)
- Bit 2: (reserved)
- Bit 1: (reserved)
- Bit 0: (reserved)

Reset value: 0x0

**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 10.11: RMT\_DATE\_REG (0x00FC)

RMT_DATE
31

**RMT\_DATE** Version control register. (R/W)

### Register 10.12: RMT\_CHnDATA\_REG ( $n$ : 0-3) (0x0000+4\* $n$ )

RMT_CHn_DATA_REG	
31	0
0x000000	<button>Reset</button>

**RMT\_CH $n$ \_DATA\_REG** This register is used to read and write data for channel  $n$  via APB FIFO. (RO)

### Register 10.13: RMT\_INT\_RAW\_REG (0x0050)

**RMT\_CHn\_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 10.14: 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\_CHn\_RX\_END\_INT\_ST** The masked interrupt status bit for RMT\_CHn\_RX\_END\_INT. (RO)

**RMT\_CHn\_ERR\_INT\_ST** The masked interrupt status bit for RMT\_CHn\_ERR\_INT. (RO)

**RMT\_CHn\_TX\_THR\_EVENT\_INT\_ST** The masked interrupt status bit for RMT\_CHn\_TX\_THR\_EVENT\_INT. (RO)

**RMT\_CHn\_TX\_LOOP\_INT\_ST** The masked interrupt status bit for **RMT\_CHn\_TX\_LOOP\_INT**. (RO)

## Register 10.15: 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\_CHn\_RX\_END\_INT\_ENA** Interrupt enable bit for RMT\_CHn\_RX\_END\_INT. (R/W)

**RMT\_CHn\_ERR\_INT\_ENA** Interrupt enable bit for RMT\_CHn\_ERR\_INT. (R/W)

**RMT\_CHn\_TX\_THR\_EVENT\_INT\_ENA** Interrupt enable bit for RMT\_CHn\_TX\_THR\_EVENT\_INT.  
(R/W)

**RMT\_CHn\_TX\_LOOP\_INT\_ENA** Interrupt enable bit for RMT\_CHn\_TX\_LOOP\_INT. (R/W)

## Register 10.16: RMT\_INT\_CLR\_REG (0x005C)

**RMT\_CHn\_TX\_END\_INT\_CLR** Set this bit to clear RMT\_CHn\_TX\_END\_INT interrupt. (WO)

**RMT\_CHn\_RX\_END\_INT\_CLR** Set this bit to clear RMT\_CHn\_RX\_END\_INT interrupt. (WO)

**RMT\_CH $n$ \_ERR\_INT\_CLR** Set this bit to clear RMT\_CH $n$ \_ERR\_INT interrupt. (WO)

**RMT\_CHn\_TX\_THR\_EVENT\_INT\_CLR** Set this bit to clear RMT\_CHn\_TX\_THR\_EVENT\_INT interrupt. (WO)

**RMT\_CHn\_TX\_LOOP\_INT\_CLR** Set this bit to clear RMT\_CHn\_TX\_LOOP\_INT interrupt. (WO)

# 11. 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 11-1, each channel has two input signals:

1. One control signal (e.g. `ctrl_ch0_un`, the control signal for ch0 of unit n)
2. One input pulse signal (e.g. `sig_ch0_un`, the input pulse signal for ch0 unit n)

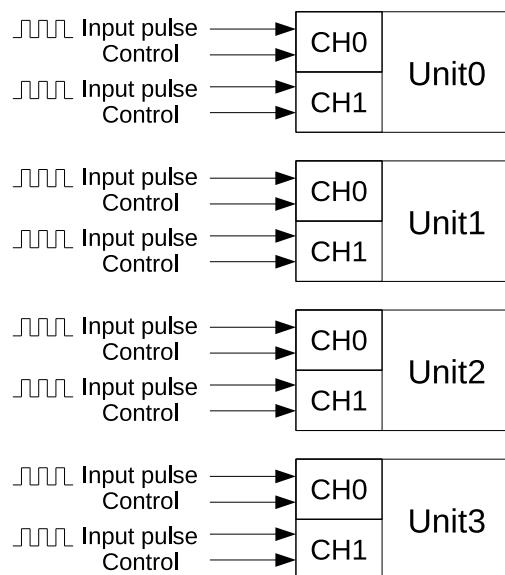


Figure 11-1. PCNT Block Diagram

## 11.1 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_un`) with their corresponding control signals (e.g. `ctrl_ch0_un`)
- Independent filtering of input pulse signals (`sig_ch0_un` and `sig_ch1_un`) and control signals (`ctrl_ch0_un` and `ctrl_ch1_un`) 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

## 11.2 Functional Description

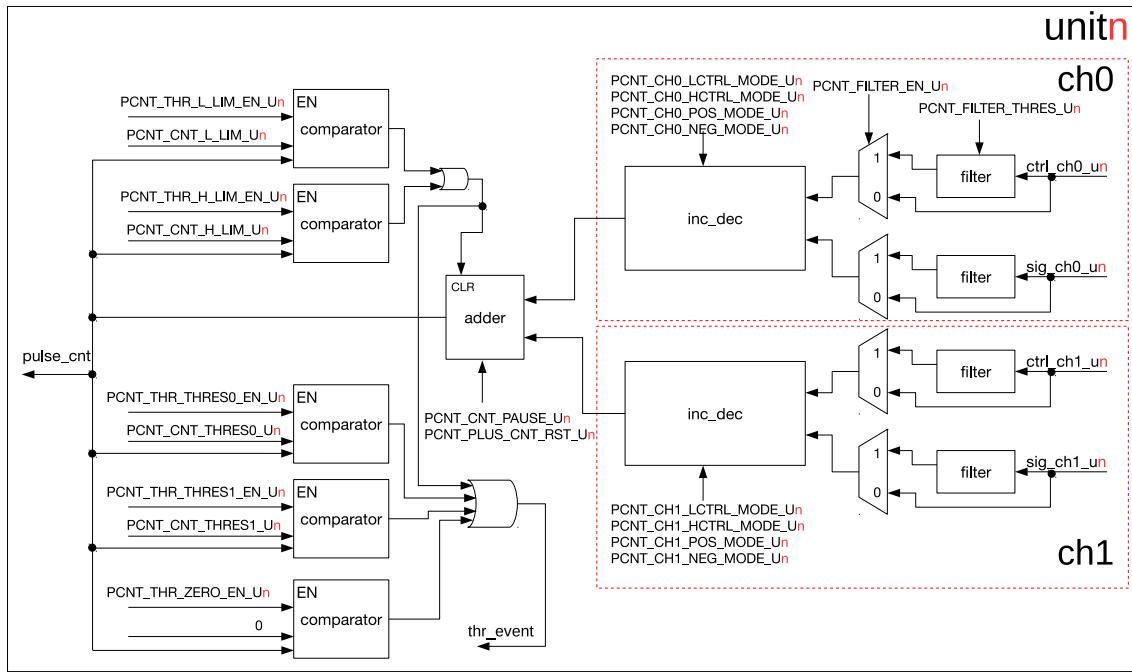


Figure 11-2. PCNT Unit Architecture

Figure 11-2 shows PCNT's architecture. As stated above, `ctrl_ch0_un` 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_un` on negative or positive edges. The available counter modes are as follows:

- Increment mode: When a channel detects an active edge of `sig_ch0_un` (the one configured for counting), the counter value `pulse_cnt` increases by 1. Upon reaching `PCNT_CNT_H_LIM_Un`, `pulse_cnt` is cleared. If the channel's counter mode is changed or if `PCNT_CNT_PAUSE_Un` is set before `pulse_cnt` reaches `PCNT_CNT_H_LIM_Un`, then `pulse_cnt` freezes and its counter mode changes.
- Decrement mode: When a channel detects an active edge of `sig_ch0_un` (the one configured for counting), the counter value `pulse_cnt` decreases by 1. Upon reaching `PCNT_CNT_L_LIM_Un`, `pulse_cnt` is cleared. If the channel's counter mode is changed or if `PCNT_CNT_PAUSE_Un` is set before `pulse_cnt` reaches `PCNT_CNT_H_LIM_Un`, then `pulse_cnt` freezes and its counter mode changes.
- Disable mode: Counting is disabled, and the counter value `pulse_cnt` freezes.

Table 43 to Table 46 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_Un`. The filter monitors the signals and ignores all the noise, i.e. the glitches with pulse widths shorter than `PCNT_FILTER_THRES_Un` 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 43: Counter Mode. Positive Edge of Input Pulse Signal. Control Signal in Low State**

PCNT_CH0_POS_MODE_0n	PCNT_CH0_LCTRL_MODE_0n	Counter Mode
1	0	Increment
	1	Decrement
	Others	Disable
2	0	Decrement
	1	Increment
	Others	Disable
Others	N/A	Disable

**Table 44: Counter Mode. Positive Edge of Input Pulse Signal. Control Signal in High State**

PCNT_CH0_POS_MODE_0n	PCNT_CH0_HCTRL_MODE_0n	Counter Mode
1	0	Increment
	1	Decrement
	Others	Disable
2	0	Decrement
	1	Increment
	Others	Disable
Others	N/A	Disable

**Table 45: Counter Mode. Negative Edge of Input Pulse Signal. Control Signal in Low State**

PCNT_CH0_NEG_MODE_0n	PCNT_CH0_LCTRL_MODE_0n	Counter Mode
1	0	Increment
	1	Decrement
	Others	Disable
2	0	Decrement
	1	Increment
	Others	Disable
Others	N/A	Disable

**Table 46: Counter Mode. Negative Edge of Input Pulse Signal. Control Signal in High State**

PCNT_CH0_NEG_MODE_0n	PCNT_CH0_HCTRL_MODE_0n	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_Un`, and cleared by setting `PCNT_PULSE_CNT_RST_Un`.

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_Un`, an interrupt is triggered and `PCNT_CNT_THR_H_LIM_LAT_Un` is high.
- Minimum count value: When `pulse_cnt` reaches `PCNT_CNT_L_LIM_Un`, an interrupt is triggered and `PCNT_CNT_THR_L_LIM_LAT_Un` is high.
- Two threshold values: When `pulse_cnt` equals either `PCNT_CNT_THRES0_Un` or `PCNT_CNT_THRES1_Un`, an interrupt is triggered and either `PCNT_CNT_THR_THRES0_LAT_Un` or `PCNT_CNT_THR_THRES1_LAT_Un` is high respectively.
- Zero: When `pulse_cnt` is 0, an interrupt is triggered and `PCNT_CNT_THR_ZERO_LAT_Un` is valid.

## 11.3 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/decrementing independently, or one channel incrementing while the other decrementing), reference can be made to these three subsections.

### 11.3.1 Channel 0 Incrementing Independently

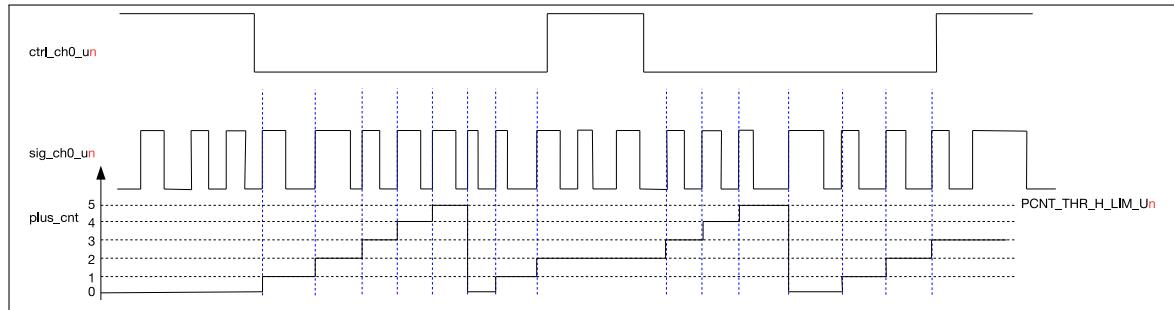


Figure 11-3. Channel 0 Up Counting Diagram

Figure 11-3 illustrates how channel 0 is configured to increment independently on the positive edge of `sig_ch0_un` while channel 1 is disabled (see subsection 11.2 for how to disable channel 1). The configuration of channel 0 is shown below.

- `PCNT_CH0_LCTRL_MODE_Un`=0: When `ctrl_ch0_un` is low, the counter mode specified for the low state turns on, in this case it is Increment mode.
- `PCNT_CH0_HCTRL_MODE_Un`=2: When `ctrl_ch0_un` is high, the counter mode specified for the low state turns on, in this case it is Disable mode.
- `PCNT_CH0_POS_MODE_Un`=1: The counter increments on the positive edge of `sig_ch0_un`.
- `PCNT_CH0_NEG_MODE_Un`=0: The counter idles on the negative edge of `sig_ch0_un`.

- PCNT\_CNT\_H\_LIM\_U $n$ =5: When pulse\_cnt counts up to PCNT\_CNT\_H\_LIM\_U $n$ , it is cleared.

### 11.3.2 Channel 0 Decrementing Independently

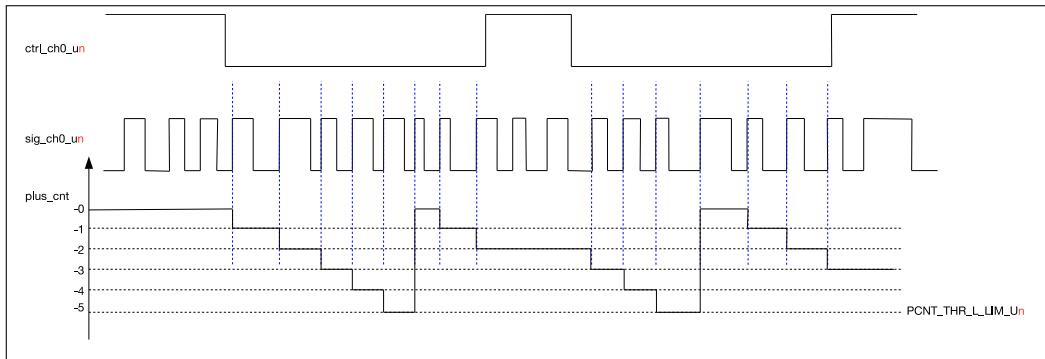


Figure 11-4. Channel 0 Down Counting Diagram

Figure 11-4 illustrates how channel 0 is configured to decrement independently on the positive edge of sig\_ch0\_un while channel 1 is disabled. The configuration of channel 0 in this case differs from that in Figure 11-3 in the following aspects:

- PCNT\_CH0\_POS\_MODE\_U $n$ =2: the counter decrements on the positive edge of sig\_ch0\_un.
- PCNT\_CNT\_L\_LIM\_U $n$ =-5: when pulse\_cnt counts down to PCNT\_CNT\_L\_LIM\_U $n$ , it is cleared.

### 11.3.3 Channel 0 and Channel 1 Incrementing Together

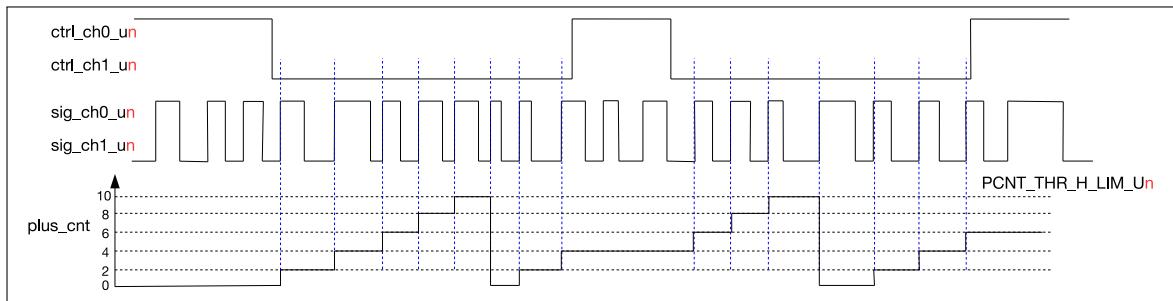


Figure 11-5. Two Channels Up Counting Diagram

Figure 11-5 illustrates how channel 0 and channel 1 are configured to increment on the positive edge of sig\_ch0\_un and sig\_ch1\_un respectively at the same time. It can be seen in Figure 11-5 that control signal ctrl\_ch0\_un and ctrl\_ch1\_un have the same waveform, so as input pulse signal sig\_ch0\_un and sig\_ch1\_un. The configuration procedure is shown below.

- For channel 0:
  - PCNT\_CH0\_LCTRL\_MODE\_U $n$ =0: When ctrl\_ch0\_un 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\_un 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\_un.
  - PCNT\_CH0\_NEG\_MODE\_U $n$ =0: The counter idles on the negative edge of sig\_ch0\_un.

- For channel 1:
  - **PCNT\_CH1\_LCTRL\_MODE\_U<sub>n</sub>**=0: When **ctrl\_ch1\_u<sub>n</sub>** is low, the counter mode specified for the low state turns on, in this case it is Increment mode.
  - **PCNT\_CH1\_HCTRL\_MODE\_U<sub>n</sub>**=2: When **ctrl\_ch1\_u<sub>n</sub>** is high, the counter mode specified for the low state turns on, in this case it is Disable mode.
  - **PCNT\_CH1\_POS\_MODE\_U<sub>n</sub>**=1: The counter increments on the positive edge of **sig\_ch1\_u<sub>n</sub>**.
  - **PCNT\_CH1\_NEG\_MODE\_U<sub>n</sub>**=0: The counter idles on the negative edge of **sig\_ch1\_u<sub>n</sub>**.
- **PCNT\_CNT\_H\_LIM\_U<sub>n</sub>**=10: When **pulse\_cnt** counts up to **PCNT\_CNT\_H\_LIM\_U<sub>n</sub>**, it is cleared.

## 11.4 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 47: PCNT Base Address**

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

## 11.5 Register Summary

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

Name	Description	Address	Access
<b>Configuration Register</b>			
<b>PCNT_U0_CONF0_REG</b>	Configuration register 0 for unit 0	0x0000	R/W
<b>PCNT_U0_CONF1_REG</b>	Configuration register 1 for unit 0	0x0004	R/W
<b>PCNT_U0_CONF2_REG</b>	Configuration register 2 for unit 0	0x0008	R/W
<b>PCNT_U1_CONF0_REG</b>	Configuration register 0 for unit 1	0x000C	R/W
<b>PCNT_U1_CONF1_REG</b>	Configuration register 1 for unit 1	0x0010	R/W
<b>PCNT_U1_CONF2_REG</b>	Configuration register 2 for unit 1	0x0014	R/W
<b>PCNT_U2_CONF0_REG</b>	Configuration register 0 for unit 2	0x0018	R/W
<b>PCNT_U2_CONF1_REG</b>	Configuration register 1 for unit 2	0x001C	R/W
<b>PCNT_U2_CONF2_REG</b>	Configuration register 2 for unit 2	0x0020	R/W
<b>PCNT_U3_CONF0_REG</b>	Configuration register 0 for unit 3	0x0024	R/W
<b>PCNT_U3_CONF1_REG</b>	Configuration register 1 for unit 3	0x0028	R/W
<b>PCNT_U3_CONF2_REG</b>	Configuration register 2 for unit 3	0x002C	R/W
<b>PCNT_CTRL_REG</b>	Control register for all counters	0x0060	R/W
<b>Status Register</b>			
<b>PCNT_U0_CNT_REG</b>	Counter value for unit 0	0x0030	RO
<b>PCNT_U1_CNT_REG</b>	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
<b>Interrupt Register</b>			
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
<b>Version Register</b>			
PCNT_DATE_REG	PCNT version control register	0x00FC	R/W

## 11.6 Registers

**Register 11.1: PCNT\_U<sub>n</sub>\_CONF0\_REG (*n*: 0-3) (0x0000+12\**n*)**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	0
0x0	0x10																						

Reset

**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 11.1: PCNT\_Un\_CONF0\_REG ( $n$ : 0-3) (0x0000+12\*n)

**Continued from the previous page ...**

**PCNT\_CH1\_HCTRL\_MODE\_U<sub>n</sub>** This register configures how the CH<sub>n</sub>\_POS\_MODE/CH<sub>n</sub>\_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<sub>n</sub>** This register configures how the CH<sub>n</sub>\_POS\_MODE/CH<sub>n</sub>\_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 11.2: PCNT\_Un\_CONF1\_REG (n: 0-3) (0x0004+12\*n)**

PCNT_CNT_THRESH1_U0	PCNT_CNT_THRESH0_U0								
<table border="1"><tr><td>31</td><td>16</td><td>15</td><td>0</td></tr><tr><td>0x00</td><td></td><td>0x00</td><td>Reset</td></tr></table>	31	16	15	0	0x00		0x00	Reset	
31	16	15	0						
0x00		0x00	Reset						

**PCNT\_CNT\_THRES0\_U<sub>n</sub>** This register is used to configure the thres0 value for unit *n*. (R/W)

**PCNT\_CNT\_THRES1\_U<sub>n</sub>** This register is used to configure the thres1 value for unit *n*. (R/W)

Register 11.3: PCNT\_Un\_CONF2\_REG ( $n$ : 0-3) (0x0008+12 $\times$  $n$ )

PCNT_CNT_L_LIM_U0	PCNT_CNT_H_LIM_U0
31	16 15 0

**PCNT\_CNT\_H\_LIM\_***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 11.4: PCNT\_CTRL\_REG (0x0060)

PCNT_CTRL_REG (0x0060)																
PCNT_CTRL_REG (0x0060)								PCNT_CTRL_REG (0x0060)								
(reserved)				PCNT_CLK_EN				(reserved)				PCNT_CTRL_REG (0x0060)				
31				17	16	15		8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
Reset																

**PCNT\_PULSE\_CNT\_RST\_Us** Set this bit to clear unit  $n$ 's counter. (R/W)

**PCNT\_CNT\_PAUSE\_Us** 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 11.5: PCNT\_DATE\_REG (0x00FC)

PCNT_DATE_REG (0x00FC)	
PCNT_DATE_REG (0x00FC)	
31	0
	0x19072601
Reset	

**PCNT\_DATE** This is the PCNT version control register. (R/W)

Register 11.6: PCNT\_UN\_CNT\_REG ( $n$ : 0-3) (0x0030+4\*n)

PCNT_UN_CNT_REG (0x0030+4*n)																
PCNT_UN_CNT_REG (0x0030+4*n)								PCNT_UN_CNT_REG (0x0030+4*n)								
(reserved)				PCNT_PULSE_CNT_U0				(reserved)				PCNT_PULSE_CNT_U0				
31				16	15			8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Reset																

**PCNT\_PULSE\_CNT\_Us** This register stores the current pulse count value for unit  $n$ . (RO)

Register 11.7: PCNT\_Un\_STATUS\_REG ( $n$ : 0-3) (0x0050+4\* $n$ )

**PCNT\_CNT\_THR\_ZERO\_MODE\_U<sub>n</sub>** The pulse counter status of PCNT\_U<sub>n</sub> 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\_THRESH1\_LAT\_U<sub>n</sub>** The latched value of thresh1 event of PCNT\_U<sub>n</sub> when threshold event interrupt is valid. 1: the current pulse counter equals to thresh1 and thresh1 event is valid. 0: others (RO)

**PCNT\_CNT\_THR\_THRESH0\_LAT\_U<sub>n</sub>** The latched value of thresh0 event of PCNT\_U<sub>n</sub> when threshold event interrupt is valid. 1: the current pulse counter equals to thresh0 and thresh0 event is valid. 0: others (RO)

**PCNT\_CNT\_THR\_L\_LIM\_LAT\_U<sub>n</sub>** The latched value of low limit event of PCNT\_U<sub>n</sub> 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<sub>n</sub>** The latched value of high limit event of PCNT\_U<sub>n</sub> 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\_Un** The latched value of zero threshold event of PCNT\_Un when threshold event interrupt is valid. 1: the current pulse counter equals to 0 and zero threshold event is valid. 0: others (RO)

### Register 11.8: PCNT\_INT\_RAW\_REG (0x0040)

**PCNT\_CNT\_THR\_EVENT\_Un\_INT\_RAW** The raw interrupt status bit for the PCNT\_CNT\_THR\_EVENT\_Un\_INT interrupt. (RO)

## Register 11.9: PCNT\_INT\_ST\_REG (0x0044)

**PCNT\_CNT\_THR\_EVENT\_Un\_INT\_ST** The masked interrupt status bit for the PCNT\_CNT\_THR\_EVENT\_Un\_INT interrupt. (RO)

## Register 11.10: PCNT\_INT\_ENA\_REG (0x0048)

**PCNT\_CNT\_THR\_EVENT\_Un\_INT\_ENA** The interrupt enable bit for the PCNT CNT THR EVENT Un INT interrupt. (R/W)

**Register 11.11: PCNT\_INT\_CLR\_REG (0x004C)**

(reserved)																														
31																														
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	0	0	0	0	0	0

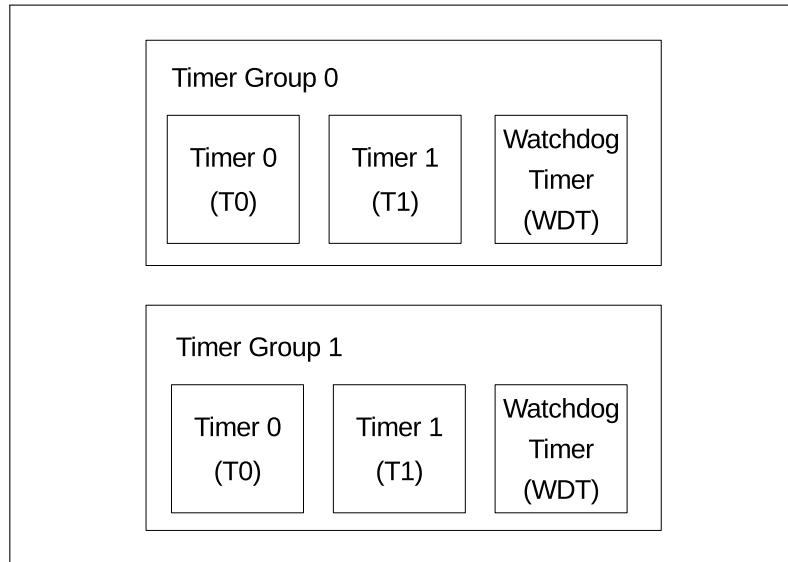
PCNT\_CNT\_THR\_EVENT\_U3\_INT\_CLR  
PCNT\_CNT\_THR\_EVENT\_U2\_INT\_CLR  
PCNT\_CNT\_THR\_EVENT\_U1\_INT\_CLR  
PCNT\_CNT\_THR\_EVENT\_U0\_INT\_CLR

**PCNT\_CNT\_THR\_EVENT\_U<sub>n</sub>\_INT\_CLR** Set this bit to clear the PCNT\_CNT\_THR\_EVENT\_U<sub>n</sub>\_INT interrupt. (WO)

# 12. 64-bit Timers

## 12.1 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 12-1, 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 12-1. 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 13: *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

## 12.2 Functional Description

### 12.2.1 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\\_Tx\\_USE\\_XTAL](#) field of the [TIMG\\_TxCONFIG\\_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\\_Tx\\_DIVIDER](#) field and can take any value from 1 to 65536. Note that programming a value of 0 in [TIMG\\_Tx\\_DIVIDER](#) will result in the divisor being 65536.

The timer must be disabled (i.e. [TIMG\\_Tx\\_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.

### 12.2.2 64-bit Time-based Counter

The 64-bit time-base counters are based on TB\_CLK and can be configured to increment or decrement via the [TIMG\\_Tx\\_INCREASE](#) field. The time-base counter can be enabled/disabled by setting/clearing the [TIMG\\_Tx\\_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\\_Tx\\_INCREASE](#) field can be changed whilst [TIMG\\_Tx\\_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\\_TxUPDATE\\_REG](#), the current value of the 64-bit timer is instantly latched into the [TIMG\\_TxLO\\_REG](#) and [TIMG\\_TxHI\\_REG](#) registers containing the lower and upper 32-bits respectively. [TIMG\\_TxLO\\_REG](#) and [TIMG\\_TxHI\\_REG](#) registers will remain unchanged for the CPU to read in its own time until [TIMG\\_TxUPDATE\\_REG](#) is written to again.

### 12.2.3 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 12.2.4). The 64-bit alarm value is configured in the [TIMG\\_TxALARMLO\\_REG](#) and [TIMG\\_TxALARMHI\\_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\\_Tx\\_ALARM\\_EN](#) field. In order to simplify 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\\_Tx\\_ALARM\\_EN](#) field is automatically cleared and no alarm will occur again until the [TIMG\\_Tx\\_ALARM\\_EN](#) is set.

### 12.2.4 Timer Reload

A timer is reloaded when a timer's current value is overwritten with a reload value stored in the [TIMG\\_Tx\\_LOAD\\_LO](#) and [TIMG\\_Tx\\_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\\_Tx\\_LOAD\\_LO](#) and [TIMG\\_Tx\\_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.

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

## 12.3 Configuration and Usage

### 12.3.1 Timer as a Simple Clock

1. Configure the time-base counter
  - Select clock source by setting **TIMG\_Tx\_USE\_XTAL** field.
  - Configure the 16-bit prescaler by setting **TIMG\_Tx\_DIVIDER**.
  - Configure the timer direction by setting/clearing **TIMG\_Tx\_INCREASE**.
  - Set the timer's starting value by writing the starting value to **TIMG\_Tx\_LOAD\_LO** and **TIMG\_Tx\_LOAD\_HI**, then reloading it into the timer by writing any value to **TIMG\_TxLOAD\_REG**.
2. Start the timer by setting **TIMG\_Tx\_EN**.
3. Getting the timer's current value.
  - Write any value to **TIMG\_TxUPDATE\_REG** to latch the timer's current value.
  - Read the latched timer value from **TIMG\_TxLO\_REG** and **TIMG\_TxHI\_REG**.

### 12.3.2 Timer as One-shot Alarm

1. Configure the time-base counter following step 1 of Section 12.3.1.
2. Configure the alarm.
  - Configure the alarm value by setting **TIMG\_TxALARMLO\_REG** and **TIMG\_TxALARMHI\_REG**.
  - Enable interrupt by setting **TIMG\_Tx\_LEVEL\_INT\_EN** or **TIMG\_Tx\_EDGE\_INT\_EN** for level or edge interrupts respectively.
3. Disable auto reload by clearing **TIMG\_Tx\_AUTORELOAD**.
4. Start the timer by setting **TIMG\_Tx\_EN**.
5. Handle the alarm interrupt.
  - Clear the interrupt by setting the timer's corresponding bit in **TIMG\_Tx\_INT\_CLR**.
  - Disable the timer by clearing **TIMG\_Tx\_EN**.

### 12.3.3 Timer as Periodic Alarm

1. Configure the time-base counter following step 1 of Section 12.3.1.
2. Configure the alarm following step 2 of Section 12.3.2.
3. Enable auto reload by setting `TIMG_Tx_AUTORELOAD` and setting the reload value in `TIMG_Tx_LOAD_LO` and `TIMG_Tx_LOAD_HI`.
4. Start the timer by setting `TIMG_Tx_EN`.
5. Handle the alarm interrupt (repeat on each alarm iteration).
  - Clear the interrupt by setting the timer's corresponding bit in `TIMG_Tx_INT_CLR`.
  - If the next alarm requires a new alarm value and reload value (i.e. different alarm interval per iteration), then `TIMG_Tx_ALARMLO_REG`, `TIMG_Tx_ALARMHI_REG`, `TIMG_Tx_LOAD_LO`, and `TIMG_Tx_LOAD_HI` should be reconfigured as needed. Otherwise, the aforementioned registers should remain unchanged.
  - Re-enable the alarm by setting `TIMG_Tx_ALARM_EN`.
6. Stopping the timer (on final alarm iteration).
  - Clear the interrupt by setting the timer's corresponding bit in `TIMG_Tx_INT_CLR`.
  - Disable the timer by clearing `TIMG_Tx_EN`.

## 12.4 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 49: 64-bit Timers Base Address**

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

## 12.5 Register Summary

The addresses in the following table are relative to the 64-bit Timer base addresses provided in Section 12.4.

Name	Description	Address	Access
<b>Configuration and Control Register for Timer 0</b>			
<code>TIMG_T0CONFIG_REG</code>	Timer 0 configuration register	0x0000	R/W
<code>TIMG_T0LO_REG</code>	Timer 0 current value, low 32 bits	0x0004	RO
<code>TIMG_T0HI_REG</code>	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_HI_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 32 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
<b>Configuration and Control Register for Timer 1</b>			
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_HI_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 32 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_T1LOADLO_REG and TIMG_T1LOADHI_REG	0x0044	WO
<b>Configuration and Control Register for WDT</b>			
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
<b>Configuration and Control Register for RTC CALI</b>			
TIMG_RTCCALICFG2_REG	Timer group calibration register	0x00A8	varies
<b>Interrupt Register</b>			
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
<b>Version Register</b>			
TIMG_TIMERS_DATE_REG	Version control register	0x00F8	R/W
<b>Configuration Register</b>			
TIMG_REGCLK_REG	Timer group clock gate register	0x00FC	R/W

## 12.6 Registers

Register 12.1: TIMG\_T<sub>x</sub>CONFIG\_REG ( $x: 0-1$ ) (0x0000+36\*x)

31	30	29	28		13	12	11	10	9	8		0
0	1	1		0x01	0	0	0	0	0	0	0	0

Reset

**TIMG\_T<sub>x</sub>\_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\_T<sub>x</sub>\_ALARM\_EN** When set, the alarm is enabled. This bit is automatically cleared once an alarm occurs. (R/W)

**TIMG\_T<sub>x</sub>\_LEVEL\_INT\_EN** When set, an alarm will generate a level type interrupt. (R/W)

**TIMG\_T<sub>x</sub>\_EDGE\_INT\_EN** When set, an alarm will generate an edge type interrupt. (R/W)

**TIMG\_T<sub>x</sub>\_DIVIDER** Timer  $x$  clock (Tx\_clk) prescaler value. (R/W)

**TIMG\_T<sub>x</sub>\_AUTORELOAD** When set, timer  $x$  auto-reload at alarm is enabled. (R/W)

**TIMG\_T<sub>x</sub>\_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\_T<sub>x</sub>\_EN** When set, the timer  $x$  time-base counter is enabled. (R/W)

Register 12.2: TIMG\_T<sub>x</sub>LO\_REG ( $x: 0-1$ ) (0x0004+36\*x)

31		0
	0x0000000	Reset

**TIMG\_T<sub>x</sub>\_LO** After writing to TIMG\_T<sub>x</sub>UPDATE\_REG, the low 32 bits of the time-base counter of timer  $x$  can be read here. (RO)

### Register 12.3: TIMG\_TxHI\_REG (x: 0-1) (0x0008+36\*x)

TIMG\_T~~X~~-H

**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 12.4: TIMG\_T<sub>x</sub>UPDATE\_REG (<sub>x</sub>: 0-1) (0x000C+36\*<sub>x</sub>)

**TIMG\_Tx\_UPDATE** After writing 0 or 1 to TIMG\_TxUPDATE\_REG, the counter value is latched. (R/W)

## Register 12.5: TIMG\_TxALARMLO\_REG (x: 0-1) (0x0010+36\*x)

TIMG_T <sub>X</sub> _ALARM_LO
0x000000

**TIMG\_Tx\_ALARM\_LO** Timer  $x$  alarm trigger time-base counter value, low 32 bits. (R/W)

Register 12.6: TIMG\_TxALARMHI\_REG (x: 0-1) (0x0014+36\*x)

31		0
	0x000000	Reset

**TIMG\_Tx\_ALARM\_HI** Timer  $x$  alarm trigger time-base counter value, high 32 bits. (R/W)

**Register 12.7: TIMG\_T<sub>x</sub>\_LOADLO\_REG (<sub>x</sub>: 0-1) (0x0018+36\*<sub>x</sub>)**

TIMG_T <sub>x</sub> _LOAD_LO	
31	0
0x000000	Reset

**TIMG\_T<sub>x</sub>\_LOAD\_LO** Low 32 bits of the value that a reload will load onto timer <sub>x</sub> time-base Counter.  
(R/W)

**Register 12.8: TIMG\_T<sub>x</sub>\_LOADHI\_REG (<sub>x</sub>: 0-1) (0x001C+36\*<sub>x</sub>)**

TIMG_T <sub>x</sub> _LOAD_HI	
31	0
0x000000	Reset

**TIMG\_T<sub>x</sub>\_LOAD\_HI** High 32 bits of the value that a reload will load onto timer <sub>x</sub> time-base counter.  
(R/W)

**Register 12.9: TIMG\_T<sub>x</sub>\_LOAD\_REG (<sub>x</sub>: 0-1) (0x0020+36\*<sub>x</sub>)**

TIMG_T <sub>x</sub> _LOAD	
31	0
0x000000	Reset

**TIMG\_T<sub>x</sub>\_LOAD** Write any value to trigger a timer <sub>x</sub> time-base counter reload. (WO)

## Register 12.10: 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 12.11: 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 12.12: TIMG\_WDTCONFIG2\_REG (0x0050)**

TIMG_WDT_STG0_HOLD	
31	0
0x18cba80	Reset

**TIMG\_WDT\_STG0\_HOLD** Stage 0 timeout value, in MWDT clock cycles. (R/W)

**Register 12.13: TIMG\_WDTCONFIG3\_REG (0x0054)**

TIMG_WDT_STG1_HOLD	
31	0
0x7fffff	Reset

**TIMG\_WDT\_STG1\_HOLD** Stage 1 timeout value, in MWDT clock cycles. (R/W)

**Register 12.14: TIMG\_WDTCONFIG4\_REG (0x0058)**

TIMG_WDT_STG2_HOLD	
31	0
0x0fffff	Reset

**TIMG\_WDT\_STG2\_HOLD** Stage 2 timeout value, in MWDT clock cycles. (R/W)

**Register 12.15: TIMG\_WDTCONFIG5\_REG (0x005C)**

TIMG_WDT_STG3_HOLD	
31	0
0xffff	Reset

**TIMG\_WDT\_STG3\_HOLD** Stage 3 timeout value, in MWDT clock cycles. (R/W)

**Register 12.16: TIMG\_WDTFEED\_REG (0x0060)**

TIMG_WDT_FEED	
31	0
0x000000	Reset

**TIMG\_WDT\_FEED** Write any value to feed the MWDT. (WO) (WO)

**Register 12.17: TIMG\_WDTWPROTECT\_REG (0x0064)**

TIMG_WDT_WKEY	
31	0
0x50d83aa1	Reset

**TIMG\_WDT\_WKEY** If the register contains a different value than its reset value, write protection is enabled. (R/W)

**Register 12.18: TIMG\_RTCCALICFG2\_REG (0x00A8)**

TIMG_RTC_CALI_TIMEOUT_THRES								TIMG_RTC_CALI_TIMEOUT_RST_CNT			
(reserved)								TIMG_RTC_CALI_TIMEOUT			
(reserved)								(reserved)			
TIMG_RTC_CALI_TIMEOUT								TIMG_RTC_CALI_TIMEOUT_RST_CNT			
31	7	6	3	2	1	0		0x3	0	0	0
0x1fffff								Reset			

**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 12.19: TIMG\_INT\_ENA\_TIMERS\_REG (0x0098)**

(reserved)								TIMG_WDT_INT_ENA			
(reserved)								TIMG_T1_INT_ENA			
(reserved)								TIMG_TO_INT_ENA			
(reserved)								(reserved)			
31	7	6	3	2	1	0		0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0								Reset			

**TIMG\_T<sub>X</sub>\_INT\_ENA** The interrupt enable bit for the TIMG\_T<sub>X</sub>\_INT interrupt. (R/W)

**TIMG\_WDT\_INT\_ENA** The interrupt enable bit for the TIMG\_WDT\_INT interrupt. (R/W)

**Register 12.20: TIMG\_INT\_RAW\_TIMERS\_REG (0x009C)**

(reserved)								TIMG_WDT_INT_RAW			
(reserved)								TIMG_T1_INT_RAW			
(reserved)								TIMG_TO_INT_RAW			
(reserved)								(reserved)			
31	7	6	3	2	1	0		0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0								Reset			

**TIMG\_T<sub>X</sub>\_INT\_RAW** The raw interrupt status bit for the TIMG\_T<sub>X</sub>\_INT interrupt. (RO)

**TIMG\_WDT\_INT\_RAW** The raw interrupt status bit for the TIMG\_WDT\_INT interrupt. (RO)

**Register 12.21: TIMG\_INT\_ST\_TIMERS\_REG (0x00A0)**

31				
0	0	0	0	0

3 2 1 0

TIMG\_WDT\_INT\_ST  
TIMG\_T1\_INT\_ST  
TIMG\_TO\_INT\_ST  
Reset

**TIMG\_T<sub>x</sub>\_INT\_ST** The masked interrupt status bit for the TIMG\_T<sub>x</sub>\_INT interrupt. (RO)

**TIMG\_WDT\_INT\_ST** The masked interrupt status bit for the TIMG\_WDT\_INT interrupt. (RO)

**Register 12.22: TIMG\_INT\_CLR\_TIMERS\_REG (0x00A4)**

31				
0	0	0	0	0

3 2 1 0

TIMG\_WDT\_INT\_CLR  
TIMG\_T1\_INT\_CLR  
TIMG\_TO\_INT\_CLR  
Reset

**TIMG\_T<sub>x</sub>\_INT\_CLR** Set this bit to clear the TIMG\_T<sub>x</sub>\_INT interrupt. (WO)

**TIMG\_WDT\_INT\_CLR** Set this bit to clear the TIMG\_WDT\_INT interrupt. (WO)

**Register 12.23: TIMG\_TIMERS\_DATE\_REG (0x00F8)**

31	28	27		
0	0	0	0	0x1907261

0

Reset

**TIMG\_TIMERS\_DATE** Version control register. (R/W)

## Register 12.24: TIMG\_REGCLK\_REG (0x00FC)

TIMG_CLK_EN		(reserved)		0	
31	30	0	0	0	0
0	0	0	0	0	Reset

**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)

# 13. Watchdog Timers

## 13.1 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 12: *64-bit Timers*.

## 13.2 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.

## 13.3 Functional Description

### 13.3.1 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 MWDTs and RTC\_CNTL\_RTC\_WDT\_FEED for RWDT.

### 13.3.2 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:

$$T_{hold0} = \text{RTC\_CTRL\_WDT\_STG0\_HOLD\_REG} \ll \text{EFUSE\_WDT\_DELAY\_SEL} + 1$$

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](#).

### 13.3.3 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.

#### 13.3.4 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.

### 13.4 Registers

MWDT registers are part of the timer submodule and are described in the Chapter [12: 64-bit Timers Timer Registers](#) section.

# 14. eFuse Controller

## 14.1 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.

## 14.2 Features

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

## 14.3 Functional Description

### 14.3.1 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 51 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 [14.3.1.1](#) and [14.3.1.2](#).

**Table 51: Parameters in BLOCK0**

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

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

Parameters	Offset	Bit Width	Hardware Use	Programming-Protection by <a href="#">EFUSE_WR_DIS</a> Bit Number	Description
<a href="#">EFUSE_SPI_BOOT_CRYPT_CNT</a>	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.
<a href="#">EFUSE_SECURE_BOOT_KEY_REVOKE0</a>	85	1	N	5	If set, revokes use of secure boot key digest 0
<a href="#">EFUSE_SECURE_BOOT_KEY_REVOKE1</a>	86	1	N	6	If set, revokes use of secure boot key digest 1
<a href="#">EFUSE_SECURE_BOOT_KEY_REVOKE2</a>	87	1	N	7	If set, revokes use of secure boot key digest 2
<a href="#">EFUSE_KEY_PURPOSE_0</a>	88	4	Y	8	KEY0 purpose, see Table 52
<a href="#">EFUSE_KEY_PURPOSE_1</a>	92	4	Y	9	KEY1 purpose, see Table 52
<a href="#">EFUSE_KEY_PURPOSE_2</a>	96	4	Y	10	KEY2 purpose, see Table 52
<a href="#">EFUSE_KEY_PURPOSE_3</a>	100	4	Y	11	KEY3 purpose, see Table 52
<a href="#">EFUSE_KEY_PURPOSE_4</a>	104	4	Y	12	KEY4 purpose, see Table 52
<a href="#">EFUSE_KEY_PURPOSE_5</a>	108	4	Y	13	KEY5 purpose, see Table 52
<a href="#">EFUSE_SECURE_BOOT_EN</a>	116	1	N	15	Enables secure boot
<a href="#">EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKES</a>	117	1	N	16	Enables aggressive secure boot key revocation mode
<a href="#">EFUSE_FLASH_TPUW</a>	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.
<a href="#">EFUSE_DIS_DOWNLOAD_MODE</a>	128	1	N	18	Disables all Download boot modes
<a href="#">EFUSE_DIS_LEGACY_SPI_BOOT</a>	129	1	N	18	Disables Legacy SPI boot mode
<a href="#">EFUSE_UART_PRINT_CONTROL</a>	130	1	N	18	Selects the default UART for printing boot messages, 0: UART0; 1: UART1
<a href="#">EFUSE_DIS_USB_DOWNLOAD_MODE</a>	132	1	N	18	Disables use of USB in UART download boot mode
<a href="#">EFUSE_ENABLE_SECURITY_DOWNLOAD</a>	133	1	N	18	Enables secure UART download mode (read/write flash only)
<a href="#">EFUSE_UART_PRINT_CONTROL</a>	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 <a href="#">EFUSE_WR_DIS</a> Bit Number	Description
<a href="#">EFUSE_PIN_POWER_SELECTION</a>	136	1	N	18	Sets default power supply for GPIO33 ~ GPIO37, set when SPI flash is initialized. 0: VDD3P3_CPU; 1: VDD_SPI
<a href="#">EFUSE_FLASH_TYPE</a>	137	1	N	18	Selects SPI flash type, 0: maximum four data lines; 1: eight data lines
<a href="#">EFUSE_FORCE_SEND_RESUME</a>	138	1	N	18	Forces ROM code to send an SPI flash resume command during SPI boot
<a href="#">EFUSE_SECURE_VERSION</a>	139	16	N	18	Secure version (used by ESP-IDF anti-rollback feature)

Table 52 lists all key purpose and their values. Setting the eFuse parameter [EFUSE\\_KEY\\_PURPOSE\\_n](#) programs the purpose for eFuse block KEYn (n: 0 ~ 5).

**Table 52: 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 53 provides the details on the parameters in BLOCK1 ~ BLOCK10.

**Table 53: Parameters in BLOCK1-10**

BLOCK	Parameters	Bit Width	Hardware Use	Write Protection by <a href="#">EFUSE_WR_DIS</a> Bit Number	Software Read Protection by <a href="#">EFUSE_RD_DIS</a> Bit Number	Description
BLOCK1	<a href="#">EFUSE_MAC</a>	48	N	20	N/A	MAC address
	<a href="#">EFUSE_SPI_PAD_CONFIGURE</a>	[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 <a href="#">EFUSE_WR_DIS</a> Bit Number	Software Read Protection by <a href="#">EFUSE_RD_DIS</a> 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
		[54:59]	N	20	N/A	D6
		[60:65]	N	20	N/A	D7
	<a href="#">EFUSE_SYS_DATA_PART0</a>	78	N	20	N/A	System data
BLOCK2	<a href="#">EFUSE_SYS_DATA_PART1</a>	256	N	21	N/A	System data
BLOCK3	<a href="#">EFUSE_USR_DATA</a>	256	N	22	N/A	User data
BLOCK4	<a href="#">EFUSE_KEY0_DATA</a>	256	Y	23	0	Key0 or user data
BLOCK5	<a href="#">EFUSE_KEY1_DATA</a>	256	Y	24	1	Key1 or user data
BLOCK6	<a href="#">EFUSE_KEY2_DATA</a>	256	Y	25	2	Key2 or user data
BLOCK7	<a href="#">EFUSE_KEY3_DATA</a>	256	Y	26	3	Key3 or user data
BLOCK8	<a href="#">EFUSE_KEY4_DATA</a>	256	Y	27	4	Key4 or user data
BLOCK9	<a href="#">EFUSE_KEY5_DATA</a>	256	Y	28	5	Key5 or user data
BLOCK10	<a href="#">EFUSE_SYS_DATA_PART2</a>	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 52). 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 14.3.1.3: *Data Storage* and Section 14.3.2: *Software Programming of Parameters*).

### 14.3.1.1 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 14.3.3).

The columns “Programming-Protected by [EFUSE\\_WR\\_DIS](#)” in Table 51 and Table 53 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.

### 14.3.1.2 EFUSE\_RD\_DIS

Only the eFuse blocks BLCK0 ~ BLOCK10 can be individually software read protected. The corresponding bit in [EFUSE\\_RD\\_DIS](#) is shown in Table 53. 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 14.3.3).

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.

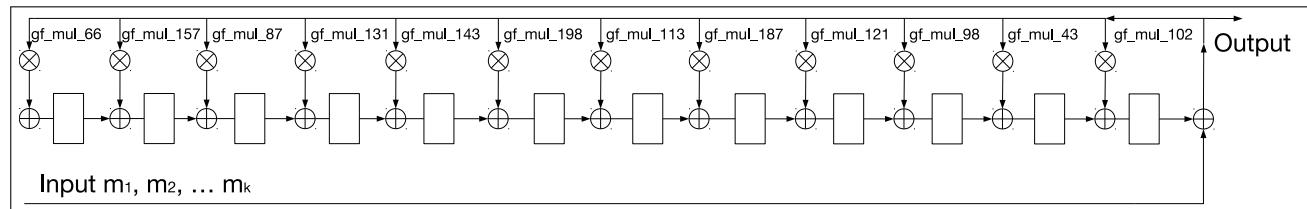
Although BLOCK4 ~ BLOCK10 can be set to read-protected, they can still be used by hardware modules, if the [EFUSE\\_KEY\\_PURPOSE\\_n](#) bit is set accordingly.

### 14.3.1.3 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 14-1, 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  $\alpha^n$ .



**Figure 14-1. 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.

### 14.3.2 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 14.3.4.1: *eFuse-Programming Timing*.
4. Ensure the eFuse programming voltage VDDQ is set correctly as described in Section 14.3.4.2: *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 14.3.3.

8. Clear the parameters written into the register.
9. Trigger an eFuse read operation (see Section 14.3.3: 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.

### 14.3.3 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	<a href="#">EFUSE_RD_WR_DIS_REG</a>	<a href="#">EFUSE_PGM_DATA0_REG</a>
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 <sub>n</sub> _DATA0 ~ 7_REG ( <sub>n</sub> : 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 14.3.4.3: *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.

### 14.3.4 Timing

#### 14.3.4.1 eFuse-Programming Timing

Figure 14-2 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 55: Configuration of eFuse-Programming Timing Parameters**

APB Frequency	<a href="#">EFUSE_TSUP_A</a> (> 6.669 ns)	<a href="#">EFUSE_TPGM</a> (9-11 $\mu$ s, usually 10 $\mu$ s)	<a href="#">EFUSE_THP_A</a> (> 6.166 ns)	<a href="#">EFUSE_TPGM_INACTIVE</a> (> 35.96 ns)
80 MHz	0x2	0x320	0x2	0x4

APB Frequency	<a href="#">EFUSE_TSUP_A</a> (> 6.669 ns)	<a href="#">EFUSE_TPGM</a> (9-11 $\mu$ s, usually 10 $\mu$ s)	<a href="#">EFUSE_THP_A</a> (> 6.166 ns)	<a href="#">EFUSE_TPGM_INACTIVE</a> (> 35.96 ns)
40 MHz	0x1	0x190	0x1	0x2
20 MHz	0x1	0xC8	0x1	0x1

In Figure 14-2, Address A0 is programmed, then the corresponding eFuse bit is 1; Address A1 is not programmed, then the corresponding eFuse bit is 0.

#### 14.3.4.2 eFuse VDDQ Timing Setting

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

**Table 56: Configuration of VDDQ Timing Parameters**

APB Frequency	<a href="#">EFUSE_DAC_CLK_DIV</a> (> 1 $\mu$ s)	<a href="#">EFUSE_PWR_ON_NUM</a> (> EFUSE_DAC_CLK_DIV*255)	<a href="#">EFUSE_PWR_OFF_NUM</a> (> 3 $\mu$ s)
80 MHz	0xA0	0xA200	0x100
40 MHz	0x50	0x5100	0x80
20 MHz	0x28	0x2880	0x40

#### 14.3.4.3 eFuse-Read Timing

Figure 14-3 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 57: Configuration of eFuse-Reading Parameters**

APB Frequency	<a href="#">EFUSE_TSUR_A</a> (> 6.669 ns)	<a href="#">EFUSE_TRD</a> (> 35.96 ns)	<a href="#">EFUSE_THR_A</a> (> 6.166 ns)
80 MHz	0x2	0x4	0x2
40 MHz	0x1	0x2	0x1
20 MHz	0x1	0x1	0x1

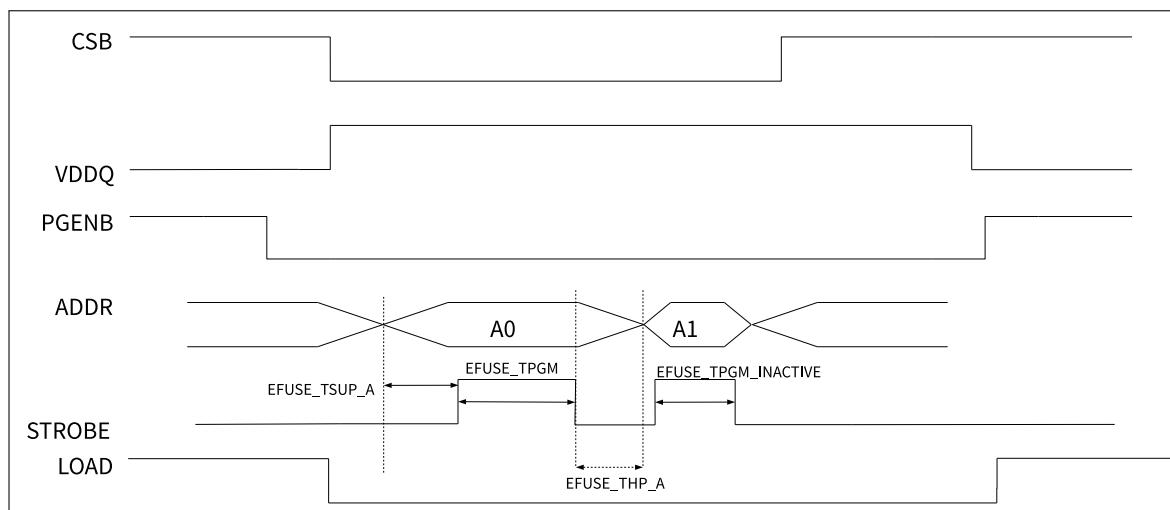


Figure 14-2. eFuse-Programming Timing Diagram

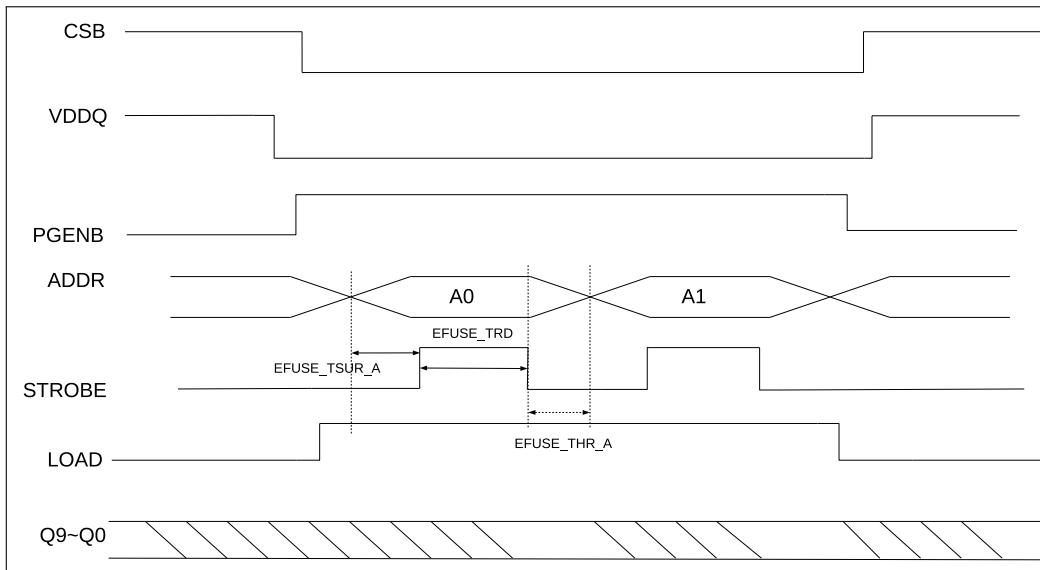


Figure 14-3. Timing Diagram for Reading eFuse

### 14.3.5 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 51 and 53, specifically those marked with “Y” in columns “Hardware Use”. Software cannot change this behavior.

### 14.3.6 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.

## 14.4 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 58: eFuse Controller Base Address**

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

## 14.5 Register Summary

The addresses in the following table are relative to the eFuse base addresses provided in Section 14.4.

Name	Description	Address	Access
<b>PGM Data Registers</b>			
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
<b>Read Data Registers</b>			
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

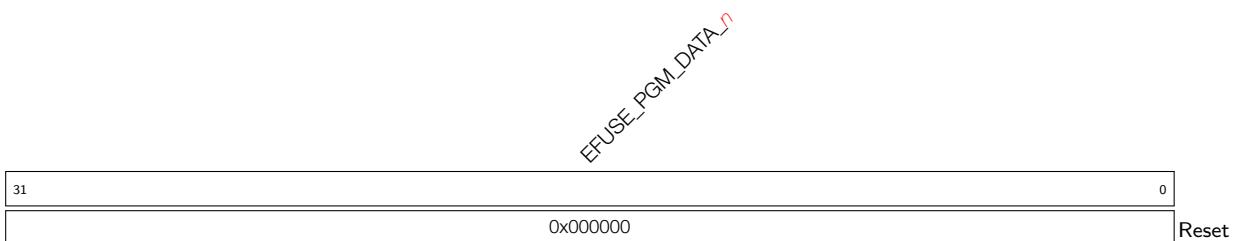
Name	Description	Address	Access
EFUSE_RD_SYS_DATA_PART1_0_REG	Register 0 of BLOCK2 (system).	0x005C	RO
EFUSE_RD_SYS_DATA_PART1_1_REG	Register 1 of BLOCK2 (system).	0x0060	RO
EFUSE_RD_SYS_DATA_PART1_2_REG	Register 2 of BLOCK2 (system).	0x0064	RO
EFUSE_RD_SYS_DATA_PART1_3_REG	Register 3 of BLOCK2 (system).	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
<b>Error Status Registers</b>			
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
<b>Control/Status Registers</b>			
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
<b>Interrupt Registers</b>			
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
<b>Configuration Registers</b>			
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
<b>Version Register</b>			
EFUSE_DATE_REG	Version control register.	0x01FC	R/W

## 14.6 Registers

Register 14.1: EFUSE\_PGM\_DATA<sub>n</sub>\_REG ( $n$ : 0-7) (0x0000+4 $\times$  $n$ )



**EFUSE\_PGM\_DATA<sub>n</sub>** The content of the  $n$ th 32-bit data to be programmed. (R/W)

**Register 14.2: EFUSE\_PGM\_CHECK\_VALUE<sub>n</sub>\_REG (*n*: 0-2) (0x0020+4\**n*)**

EFUSE_PGM_RS_DATA_	
31	0
0x000000	Reset

**EFUSE\_PGM\_RS\_DATA\_***n* The content of the *n*th 32-bit RS code to be programmed. (R/W)

**Register 14.3: EFUSE\_RD\_WR\_DIS\_REG (0x002C)**

EFUSE_WR_DIS	
31	0
0x000000	Reset

**EFUSE\_WR\_DIS** Disables programming of individual eFuses. (RO)

## Register 14.4: EFUSE\_RD\_REPEAT\_DATA0\_REG (0x0030)

	31	29	28	27	26	25	24	23	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	0
	0	0	0	0x0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0	Reset

**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 14.5: EFUSE\_RD\_REPEAT\_DATA1\_REG (0x0034)

	EFUSE_KEY_PURPOSE_1	EFUSE_KEY_PURPOSE_0	EFUSE_SECURE_BOOT_KEY_REVOKE2	EFUSE_SECURE_BOOT_KEY_REVOKE1	EFUSE_SECURE_BOOT_KEY_REVOKE0	EFUSE_SPI_BOOT_CRYPT_CNT	EFUSE_WDT_DELAY_SEL	(reserved)	EFUSE_VDD_SPI_FORCE	EFUSE_VDD_SPI_TIEH	EFUSE_VDD_SPI_XPD	(reserved)	Reset				
31	28	27	24	23	22	21	20	18	17	16	15	7	6	5	4	3	0
0x0	0x0	0	0	0	0	0x0	0x0	0	0	0	0	0	0	0	0	0	0

**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 52 Key Purpose Values. (RO)

**EFUSE\_KEY\_PURPOSE\_1** Purpose of KEY1. Refer to Table 52 Key Purpose Values. (RO)

## Register 14.6: EFUSE\_RD\_REPEAT\_DATA2\_REG (0x0038)

	EFUSE_FLASH_TPUW	EFUSE_RPT4_RESERVED1	EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKEREFUSE_SECURE_BOOT_EN (reserved)	EFUSE_KEY_PURPOSE_5	EFUSE_KEY_PURPOSE_4	EFUSE_KEY_PURPOSE_3	EFUSE_KEY_PURPOSE_2	Reset
31	28   27	22   21   20   19	16   15	12   11	8   7	4   3	0	
0x0	0x0	0   0   0   0   0   0	0x0	0x0	0x0	0x0	0x0	Reset

**EFUSE\_KEY\_PURPOSE\_2** Purpose of KEY2. Refer to Table 52 Key Purpose Values. (RO)

**EFUSE\_KEY\_PURPOSE\_3** Purpose of KEY3. Refer to Table 52 Key Purpose Values. (RO)

**EFUSE\_KEY\_PURPOSE\_4** Purpose of KEY4. Refer to Table 52 Key Purpose Values. (RO)

**EFUSE\_KEY\_PURPOSE\_5** Purpose of KEY5. Refer to Table 52 Key Purpose Values. (RO)

**EFUSE\_SECURE\_BOOT\_EN** Set this bit to enable secure boot. (RO)

**EFUSE\_SECURE\_BOOT\_AGGRESSIVE\_REVOCEREFUSE\_SECURE\_BOOT\_EN** 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 14.7: EFUSE\_RD\_REPEAT\_DATA3\_REG (0x003C)

31	27	26		11	10	9	8	7	6	5	4	3	2	1	0
0x0			0x00	0	0	0	0x0	0	0	0	0	0	0	0	Reset

EFUSE\_RPT4\_RESERVED2

EFUSE\_SECURE\_VERSION

EFUSE\_FORCE\_SEND\_RESUME  
EFUSE\_FLASH\_TYPE  
EFUSE\_PIN\_POWER\_SELECTION  
EFUSE\_ENABLE\_SECURITY\_DOWNLOAD  
EFUSE\_DIS\_USB\_DOWNLOAD\_MODE  
EFUSE\_RPT4\_RESERVED3  
EFUSE\_UART\_PRINT\_CHANNEL  
EFUSE\_DIS\_LEGACY\_SPI\_BOOT  
EFUSE\_DIS\_DOWNLOAD\_MODE

**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 14.8: EFUSE\_RD\_REPEAT\_DATA4\_REG (0x0040)**

								<i>EFUSE_RPT4_RESERVED4</i>
(reserved)								
31	24	23						0
0	0	0	0	0	0	0	0	Reset

**EFUSE\_RPT4\_RESERVED4** Reserved (used for four backups method). (RO)

**Register 14.9: EFUSE\_RD\_MAC\_SPI\_SYS\_0\_REG (0x0044)**

								<i>EFUSE_MAC_0</i>
(reserved)								
31								0
								0x000000
								Reset

**EFUSE\_MAC\_0** Stores the low 32 bits of MAC address. (RO)

**Register 14.10: EFUSE\_RD\_MAC\_SPI\_SYS\_1\_REG (0x0048)**

								<i>EFUSE_MAC_1</i>
(reserved)								
31	16	15						0
								0x00
								Reset

**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 14.11: EFUSE\_RD\_MAC\_SPI\_SYS\_2\_REG (0x004C)**

EFUSE_SPI_PAD_CONF_1	
31	0
0x000000	Reset

**EFUSE\_SPI\_PAD\_CONF\_1** Stores the first part of SPI\_PAD\_CONF. (RO)

**Register 14.12: EFUSE\_RD\_MAC\_SPI\_SYS\_3\_REG (0x0050)**

EFUSE_SYS_DATA_PART0_0	
31	18   17
0x00	0x000

**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 14.13: EFUSE\_RD\_MAC\_SPI\_SYS\_4\_REG (0x0054)**

EFUSE_SYS_DATA_PART0_1	
31	0
0x000000	Reset

**EFUSE\_SYS\_DATA\_PART0\_1** Stores the fist part of the zeroth part of system data. (RO)

**Register 14.14: EFUSE\_RD\_MAC\_SPI\_SYS\_5\_REG (0x0058)**

EFUSE_SYS_DATA_PART0_2	
31	0
0x000000	Reset

**EFUSE\_SYS\_DATA\_PART0\_2** Stores the second part of the zeroth part of system data. (RO)

**Register 14.15: EFUSE\_RD\_SYS\_DATA\_PART1\_n\_REG ( $n$ : 0-7) (0x005C+4\*n)**

EFUSE_SYS_DATA_PART1_n	
31	0
0x000000	Reset

**EFUSE\_SYS\_DATA\_PART1\_n** Stores the  $n$ th 32 bits of the first part of system data. (RO)

**Register 14.16: EFUSE\_RD\_USR\_DATAn\_REG ( $n$ : 0-7) (0x007C+4\*n)**

EFUSE_USR_DATA $n$	
31	0
0x000000	Reset

**EFUSE\_USR\_DATA $n$**  Stores the  $n$ th 32 bits of BLOCK3 (user). (RO)

**Register 14.17: EFUSE\_RD\_KEY0\_DATA $n$ \_REG ( $n$ : 0-7) (0x009C+4\* $n$ )**

EFUSE_KEY0_DATA $n$	
31	0
0x000000	Reset

**EFUSE\_KEY0\_DATA $n$**  Stores the  $n$ th 32 bits of KEY0. (RO)

**Register 14.18: EFUSE\_RD\_KEY1\_DATA $n$ \_REG ( $n$ : 0-7) (0x00BC+4\* $n$ )**

EFUSE_KEY1_DATA $n$	
31	0
0x000000	Reset

**EFUSE\_KEY1\_DATA $n$**  Stores the  $n$ th 32 bits of KEY1. (RO)

**Register 14.19: EFUSE\_RD\_KEY2\_DATA $n$ \_REG ( $n$ : 0-7) (0x00DC+4\* $n$ )**

EFUSE_KEY2_DATA $n$	
31	0
0x000000	Reset

**EFUSE\_KEY2\_DATA $n$**  Stores the  $n$ th 32 bits of KEY2. (RO)

**Register 14.20: EFUSE\_RD\_KEY3\_DATA $n$ \_REG ( $n$ : 0-7) (0x00FC+4\* $n$ )**

EFUSE_KEY3_DATA $n$	
31	0
0x000000	Reset

**EFUSE\_KEY3\_DATA $n$**  Stores the  $n$ th 32 bits of KEY3. (RO)

**Register 14.21: EFUSE\_RD\_KEY4\_DATA $n$ \_REG ( $n$ : 0-7) (0x011C+4\* $n$ )**

EFUSE_KEY4_DATA $n$	
31	0
0x000000	Reset

**EFUSE\_KEY4\_DATA $n$**  Stores the  $n$ th 32 bits of KEY4. (RO)

**Register 14.22: EFUSE\_RD\_KEY5\_DATA $n$ \_REG ( $n$ : 0-7) (0x013C+4\* $n$ )**

EFUSE_KEY5_DATA $n$	
31	0
0x000000	Reset

**EFUSE\_KEY5\_DATA $n$**  Stores the  $n$ th 32 bits of KEY5. (RO)

**Register 14.23: EFUSE\_RD\_SYS\_DATA\_PART2\_ $n$ \_REG ( $n$ : 0-7) (0x015C+4\* $n$ )**

EFUSE_SYS_DATA_PART2 $n$	
31	0
0x000000	Reset

**EFUSE\_SYS\_DATA\_PART2 $n$**  Stores the  $n$ th 32 bits of the 2nd part of system data. (RO)

## Register 14.24: EFUSE\_RD\_REPEAT\_ERR0\_REG (0x017C)

	EFUSE_RD_DIS_ERR	EFUSE_DIS_RTC_RAM_BOOT_ERR	EFUSE_DIS_DCACHE_ERR	EFUSE_DIS_DOWNLOAD_DCACHE_ERR	EFUSE_DIS_FORCE_DOWNLOAD_DCACHE_ERR	EFUSE_DIS_USB_ERR	EFUSE_DIS_CAN_ERR	EFUSE_DIS_BOOT_REMAP_ERR	EFUSE_SOFT_DIS_JTAG_ERR	EFUSE_HARD_DIS_JTAG_ERR	EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT_ERR	(reserved)	EFUSE_USB_EXCHG_PINS_ERR	EFUSE_EXT_PHY_ENABLE_ERR	EFUSE_USB_FORCE_NOPERSIST_ERR	EFUSE_RPT4_RESERVED0_ERR	(reserved)	EFUSE_RD_DIS_ERR
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0x0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	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	0	0	0	0	0	0	0	0	0	0	0	0

**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\_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 14.24: 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 14.25: EFUSE\_RD\_REPEAT\_ERR1\_REG (0x0180)

31	28	27	24	23	22	21	20	18	17	16	15		7	6	5	4	3	0
0x0		0x0	0	0	0		0x0	0x0	0	0	0	(reserved)	0	0	0	0	0	Reset

**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\_REVOKEO\_ERR** Any bit equal to 1 denotes a programming error in [EFUSE\\_SECURE\\_BOOT\\_KEY\\_REVOKEO](#). (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 14.26: EFUSE\_RD\_REPEAT\_ERR2\_REG (0x0184)

31	28	27	22	21	20	19	16	15	12	11	8	7	4	3	0
0x0		0x0	0	0	0	0	0	0	0x0	0x0	0x0	0x0	0x0	Reset	

**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\_REVOKER** Any bit equal to 1 denotes a programming error in [EFUSE\\_SECURE\\_BOOT\\_AGGRESSIVE\\_REVOKER](#). (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 14.27: EFUSE\_RD\_REPEAT\_ERR3\_REG (0x0188)

		EFUSE_RD_REPEAT_ERR3_REG												
		Bit Description												
31	27	26	11	10	9	8	7	6	5	4	3	2	1	0
0x0		0x00	0	0	0	0x0	0	0	0	0	0	0	0	Reset

**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 14.28: EFUSE\_RD\_REPEAT\_ERR4\_REG (0x0190)**

31	24	23	0
0	0	0	0

EFUSE\_RPT4\_RESERVED4\_ERR

Reset

**EFUSE\_RPT4\_RESERVED4\_ERR** If any bit in RPT4\_RESERVED4 is 1, there is a programming error in [EFUSE\\_RPT4\\_RESERVED4](#). (RO)

**Register 14.29: EFUSE\_RD\_RS\_ERR0\_REG (0x01C0)**

31	30	28	27	26	24	23	22	20	19	18	16	15	14	12	11	10	8	7	6	4	3	2	0
0	0x0	0	0x0	0	0x0	0	0x0																

Reset

**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<sub>n</sub>\_ERR\_NUM** The value of this signal means the number of error bytes in KEY<sub>n</sub>. (RO)

**EFUSE\_KEY<sub>n</sub>\_FAIL** 0: Means no failure and that the data of KEY<sub>n</sub> is reliable; 1: Means that programming KEY<sub>n</sub> failed and the number of error bytes is over 5. (RO)

## Register 14.30: EFUSE\_RD\_RS\_ERR1\_REG (0x01C4)

								EFUSE_KEY5_FAIL	EFUSE_KEY5_ERR_NUM
								EFUSE_SYS_PART2_FAIL	EFUSE_SYS_PART2_ERR_NUM
								(reserved)	
31								8	7 6 4 3 2 0
0	0	0	0	0	0	0	0	0	0x0 0 0x0 Reset

**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 14.31: EFUSE\_CLK\_REG (0x01C8)

								EFUSE_EFUSE_MEM_FORCE_PU	EFUSE_EFUSE_MEM_CLK_FORCE_ON
								EFUSE_EFUSE_MEM_FORCE_ON	EFUSE_EFUSE_MEM_CLK_FORCE_ON
								(reserved)	(reserved)
31				17	16	15		3	2 1 0
0	0	0	0	0	0	0	0	0	0 1 0 Reset

**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 14.32: EFUSE\_CONF\_REG (0x01CC)**

(reserved)																EFUSE_OP_CODE			
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																0x00			
Reset																0			

**EFUSE\_OP\_CODE** 0x5A5A: Operate programming command; 0x5AA5: Operate read command.  
(R/W)

**Register 14.33: EFUSE\_CMD\_REG (0x01D4)**

(reserved)																EFUSE_BLK_NUM EFUSE_PGM_CMD EFUSE_READ_CMD			
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																0x00			
Reset																0 0 0 0			

**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 14.34: EFUSE\_DAC\_CONF\_REG (0x01E8)**

(reserved)																EFUSE_OE_CLR EFUSE_DAC_NUM EFUSE_DAC_CLK_PAD_SEL EFUSE_DAC_CLK_DIV			
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																9 8 7 0			
Reset																0 28			

**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 14.35: EFUSE\_STATUS\_REG (0x01D0)**

										EFUSE_REPEAT_ERR_CNT		EFUSE_STATE	
(reserved)													
31	18	17			10	9			4	3	0		
0	0	0	0	0	0	0	0	0	0	0	0	0x0	Reset

**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 14.36: EFUSE\_INT\_RAW\_REG (0x01D8)**

																EFUSE_PGM_DONE_INT_RAW			EFUSE_READ_DONE_INT_RAW		
(reserved)																					
31																2	1	0			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**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 14.37: EFUSE\_INT\_ST\_REG (0x01DC)**

																EFUSE_PGM_DONE_INT_ST			EFUSE_READ_DONE_INT_ST		
(reserved)																					
31																2	1	0			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**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 14.38: EFUSE\_INT\_ENA\_REG (0x01E0)

(reserved)																																	
31																																	

**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 14.39: EFUSE\_INT\_CLR\_REG (0x01E4)

(reserved)																																	
31																																	

**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 14.40: EFUSE\_RD\_TIM\_CONF\_REG (0x01EC)

EFUSE_READ_INIT_NUM	EFUSE_TSUR_A	EFUSE_TRD	EFUSE_THR_A																														
31	24	23	16	15	8	7	0																										
0x12		0x1		0x1		0x1		0x1																									

**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 14.41: EFUSE\_WR\_TIM\_CONF0\_REG (0x01F0)

	EFUSE_TPGM	EFUSE_TPGM_INACTIVE	EFUSE_THP_A
31	16   15	8   7	0
0xc8	0x1	0x1	Reset

**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 14.42: EFUSE\_WR\_TIM\_CONF1\_REG (0x01F4)

EFUSE_PWR_ON_NUM							
(reserved)				EFUSE_TSUP_A			
31	24	23	8	7	0		
0	0	0	0	0	0	0x1	Reset

**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 14.43: EFUSE\_WR\_TIM\_CONF2\_REG (0x01F8)

**EFUSE PWR OFF NUM** Configures the power outage time for VDDQ. (R/W)

**Register 14.44: EFUSE\_DATE\_REG (0x01FC)**

EFUSE_DATE	
31	0
0x19081100	<a href="#">Reset</a>

**EFUSE\_DATE** Version control register. (R/W)

# 15. I<sup>2</sup>C Controller

## 15.1 Overview

The I<sup>2</sup>C (Inter-Integrated Circuit) controller allows ESP32-S2 to communicate with multiple peripheral devices. These peripheral devices can share one bus.

## 15.2 Features

The I<sup>2</sup>C 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

## 15.3 I<sup>2</sup>C Functional Description

### 15.3.1 I<sup>2</sup>C Introduction

The I<sup>2</sup>C 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 I<sup>2</sup>C 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 low 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.

### 15.3.2 I<sup>2</sup>C Architecture

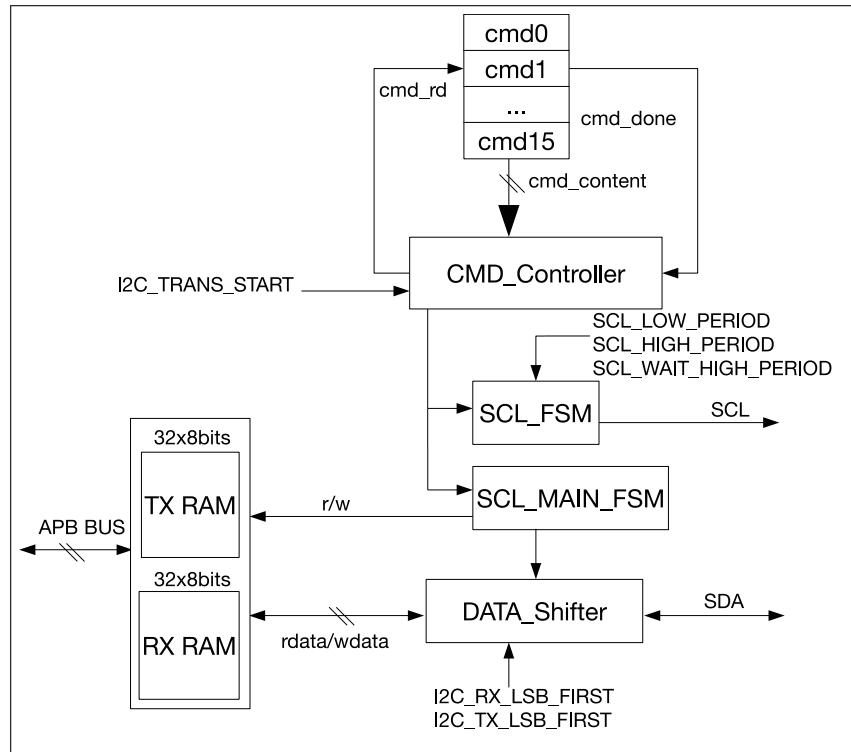


Figure 15-1. I<sup>2</sup>C Master Architecture

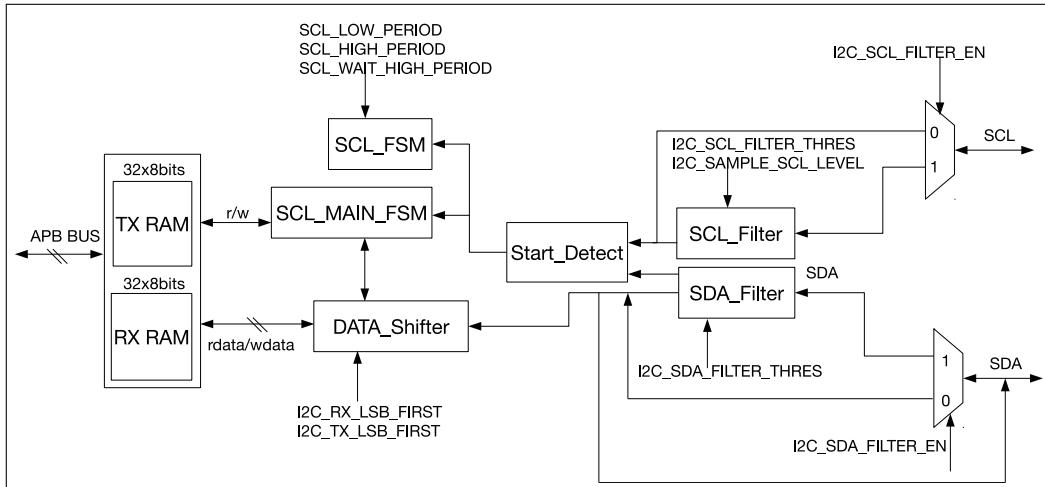


Figure 15-2. I<sup>2</sup>C Slave Architecture

The I<sup>2</sup>C controller runs either in master mode or slave mode, which is determined by [I2C\\_MS\\_MODE](#) bit. Figure 15-1 shows the architecture of a master, while Figure 15-2 shows that of a slave. The I<sup>2</sup>C 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).

### 15.3.2.1 TX/RX RAM

Both TX RAM and RX RAM are  $32 \times 8$  bits. TX RAM stores data that the I<sup>2</sup>C controller needs to send. During communication, when the I<sup>2</sup>C controller needs to send data (except acknowledgement bits), it reads data from TX RAM and sends it sequentially via SDA. When the I<sup>2</sup>C 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 I<sup>2</sup>C controller works in slave mode, TX RAM only stores data to be sent.

RX RAM stores data the I<sup>2</sup>C controller receives during communication. When the I<sup>2</sup>C 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 I<sup>2</sup>C communication completes.

Both TX RAM and RX RAM can be accessed in FIFO or non-FIFO mode. The [I<sup>2</sup>C\\_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 [I<sup>2</sup>C\\_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 ([I<sup>2</sup>C Base Address](#) + 0x100) ~([I<sup>2</sup>C Base Address](#) + 0x17C). Each byte in TX RAM occupies an entire word in the address space. Therefore, the address of the first byte is [I<sup>2</sup>C Base Address](#) + 0x100, the second byte [I<sup>2</sup>C Base Address](#) + 0x104, the third byte [I<sup>2</sup>C 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 [I<sup>2</sup>C 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 [I<sup>2</sup>C\\_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 ([I<sup>2</sup>C Base Address](#) + 0x100) ~([I<sup>2</sup>C Base Address](#) + 0x17C). Each byte in RX RAM occupies an entire word in the address space. Therefore, the address of the first byte is [I<sup>2</sup>C Base Address](#) + 0x100, the second byte [I<sup>2</sup>C Base Address](#) + 0x104, the third byte [I<sup>2</sup>C 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  $32 \times 8$  RAM. In following sections TX RAM and RX RAM are referred to as RAM.

### 15.3.2.2 CMD\_Controller

When the I<sup>2</sup>C 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.

	31	30:14	13:11	10	9	8	7:0
cmd0	CMD_DONE	N/A	op_code	ack_value	ack_exp	ack_check_en	byte_num
⋮							
cmd15	CMD_DONE	N/A	op_code	ack_value	ack_exp	ack_check_en	byte_num

Figure 15-3. Structure of I<sup>2</sup>C Command Register

Command registers, whose structure is illustrated in Figure 15-3, are active only when the I<sup>2</sup>C 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 I<sup>2</sup>C controller supports five commands:
  - RSTART: op\_code = 0: The I<sup>2</sup>C controller sends a START bit and a RESTART bit defined by the I<sup>2</sup>C protocol.
  - WRITE: op\_code = 1: The I<sup>2</sup>C controller sends a slave address, a register address (only in double addressing mode) and data to the slave.
  - READ: op\_code = 2: The I<sup>2</sup>C controller reads data from the slave.
  - STOP: op\_code = 3: The I<sup>2</sup>C controller sends a STOP bit defined by the I<sup>2</sup>C 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 I<sup>2</sup>C controller pulls the SCL pin down and suspends I<sup>2</sup>C 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 I<sup>2</sup>C 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 I<sup>2</sup>C controller during a write operation. This bit is ignored during RSTART, STOP, END and READ conditions.
5. ack\_check\_en: Used to enable the I<sup>2</sup>C 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 I<sup>2</sup>C 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 I<sup>2</sup>C communication.

### 15.3.2.3 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.

### 15.3.2.4 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.

### 15.3.2.5 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.

### 15.3.2.6 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.

### 15.3.3 I<sup>2</sup>C Bus Timing

The I<sup>2</sup>C 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.

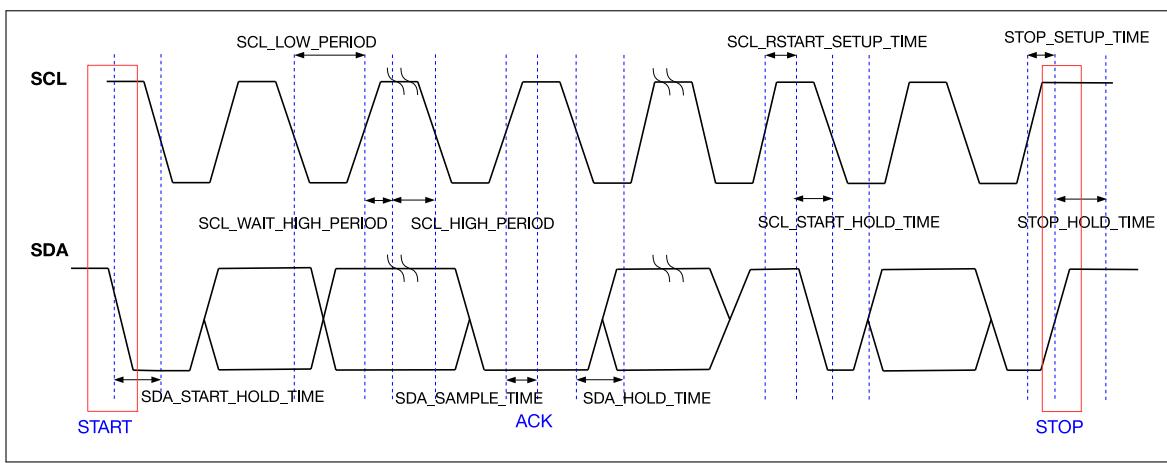
Figure 15-4. I<sup>2</sup>C Timing Diagram

Figure 15-4 shows the timing diagram of an I<sup>2</sup>C master. The unit of parameters in this figure is I<sup>2</sup>C\_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 15-4 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 15-4 are described as the following:

- 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  $(I2C\_SCL\_START\_HOLD\_TIME + 1) \times T_{I2C\_CLK}$ . This register is active only when the I<sup>2</sup>C controller works in master mode.
- I2C\_SCL\_LOW\_PERIOD:** Specifies the low period of SCL. This period lasts  $(I2C\_SCL\_START\_HOLD\_TIME + 1) \times T_{I2C\_CLK}$ . However, it could be extended when SCL is pulled low by peripheral devices or by an END command executed by the I<sup>2</sup>C controller, or when the clock is stretched. This register is active only when the I<sup>2</sup>C controller works in master mode.
- 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 I<sup>2</sup>C controller works in master mode.
- I2C\_SCL\_HIGH\_PERIOD:** Specifies the high period of SCL in  $T_{I2C\_CLK}$ . This register is active only when the I<sup>2</sup>C controller works in master mode. When SCL goes high within  $(I2C\_SCL\_WAIT\_HIGH\_PERIOD + 1) \times T_{I2C\_CLK}$ , its frequency is:

$$f_{scl} = \frac{f_{I2C\_CLK}}{I2C\_SCL\_LOW\_PERIOD + 1 + I2C\_SCL\_HIGH\_PERIOD + I2C\_SCL\_WAIT\_HIGH\_PERIOD}$$

- 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.
- 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:

- Set `I2C_SCL_FORCE_OUT` and `I2C_SDA_FORCE_OUT`, and configure `GPIO_PINn_PAD_DRIVER` register for corresponding SCL and SDA pads as open-drain.

2. Clear `I2C_SCL_FORCE_OUT` and `I2C_SDA_FORCE_OUT`.

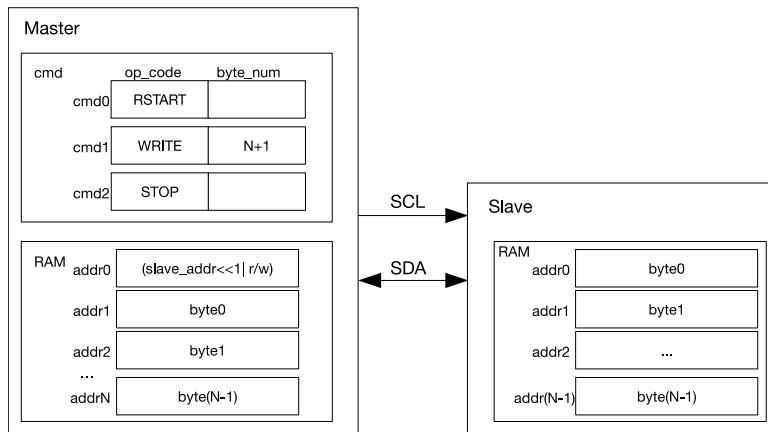
Because these lines are configured as open-drain, the low to high transition time of each line is determined together by the pull-up resistance and the capacitance on the line. The output frequency of I<sup>2</sup>C is limited by the SDA and SCL line's pull-up speed, mainly SCL's speed.

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.

## 15.4 Typical Applications

For the convenience of description, I<sup>2</sup>C masters and slaves in all subsequent figures refer to ESP32-S2 I<sup>2</sup>C peripheral controllers. However, these controllers can communicate with any other I<sup>2</sup>C devices.

### 15.4.1 An I<sup>2</sup>C Master Writes to an I<sup>2</sup>C Slave with a 7-bit Address in One Command Sequence



**Figure 15-5. An I<sup>2</sup>C Master Writing to an I<sup>2</sup>C Slave with a 7-bit Address**

Figure 15-5 shows how an I<sup>2</sup>C master writes N bytes of data using 7-bit addressing. As shown in figure 15-5, the first byte in the master's RAM is a 7-bit slave address followed by a  $R/\bar{W}$  bit. When the  $R/\bar{W}$  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 I<sup>2</sup>C 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\_RX\_UPDATE bit to latch the start address and the end address of data sent in the RAM, and then read I2C\_RXFIFO\_START\_ADDR and I2C\_RXFIFO\_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 I<sup>2</sup>C 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 I<sup>2</sup>C 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 I<sup>2</sup>C master, the I<sup>2</sup>C 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.

### 15.4.2 An I<sup>2</sup>C Master Writes to an I<sup>2</sup>C Slave with a 10-bit Address in One Command Sequence

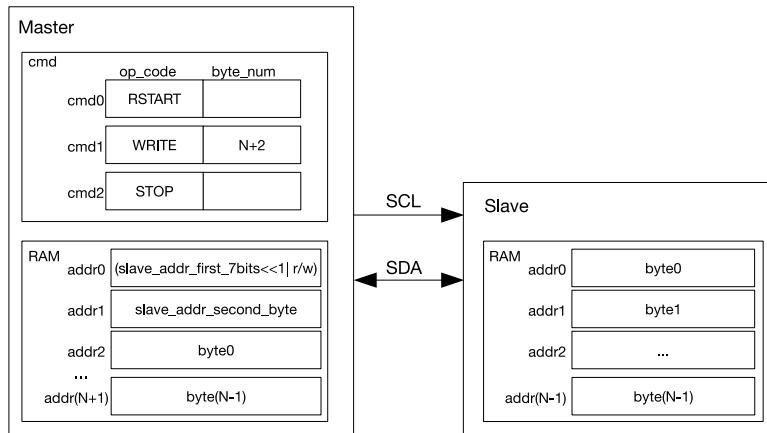


Figure 15-6. A Master Writing to a Slave with a 10-bit Address

Besides 7-bit addressing (SLV\_ADDR[6:0]), the ESP32-S2 I<sup>2</sup>C controller also supports 10-bit addressing (SLV\_ADDR[9:0]). In the following text, the slave address is referred to as SLV\_ADDR.

Figure 15-6 shows how an I<sup>2</sup>C 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/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 I<sup>2</sup>C 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.

### 15.4.3 An I<sup>2</sup>C Master Writes to an I<sup>2</sup>C Slave with Two 7-bit Addresses in One Command Sequence

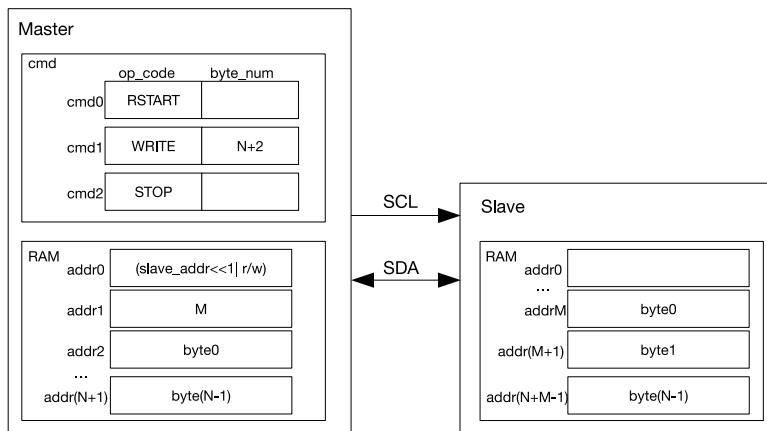


Figure 15-7. An I<sup>2</sup>C Master Writing Address M in the RAM to an I<sup>2</sup>C Slave with a 7-bit Address

When working in slave mode, the controller supports double addressing, where the first address is the address of an I<sup>2</sup>C 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 15-7 illustrates, the I<sup>2</sup>C slave put received byte0 ~ byte(N-1) into its RAM in an order staring from addrM. The RAM is overwritten every 32 bytes.

### 15.4.4 An I<sup>2</sup>C Master Writes to an I<sup>2</sup>C Slave with a 7-bit Address in Multiple Command Sequences

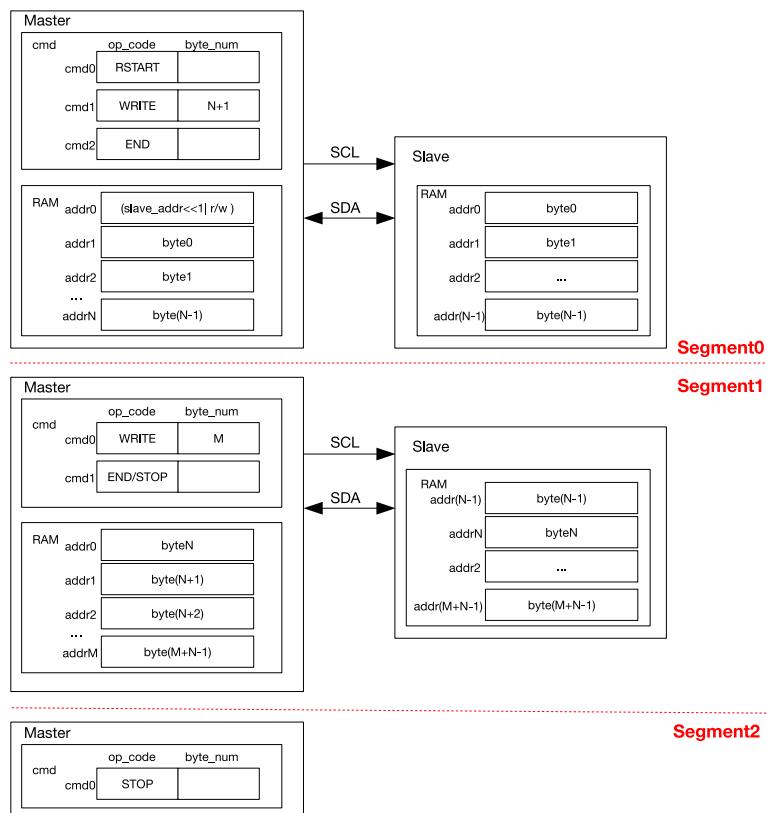


Figure 15-8. An I<sup>2</sup>C Master Writing to an I<sup>2</sup>C Slave with a 7-bit Address in Multiple Sequences

Given that the I<sup>2</sup>C 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 15-8 shows how an I<sup>2</sup>C master writes to an I<sup>2</sup>C 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.

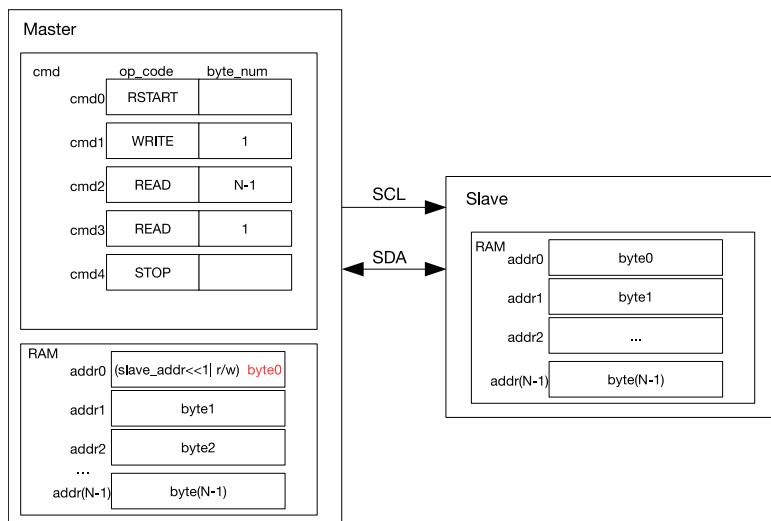
For the third segment, after the second data transfer finishes and an I<sup>2</sup>C\_END\_DETECT\_INT is detected, the CMD\_Controller registers of the master are configured as shown in Segment2. Once I<sup>2</sup>C\_TRANS\_START is set, the master generates a STOP bit and terminates the transfer.

Please note that other I<sup>2</sup>C 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 I<sup>2</sup>C controller can be reset by setting I<sup>2</sup>C\_FSM\_RST at any time. This register will later be cleared automatically by hardware.

When the master is in IDLE state and I<sup>2</sup>C\_SCL\_RST\_SLV\_EN is set, hardware sends I<sup>2</sup>C\_SCL\_RST\_SLV\_NUM SCL pulses. I<sup>2</sup>C\_SCL\_RST\_SLV\_EN will later be cleared automatically by hardware.

Note that the operation of other I<sup>2</sup>C masters and I<sup>2</sup>C slaves may differ from that of the ESP32-S2 I<sup>2</sup>C devices. Please refer to datasheets of specific I<sup>2</sup>C devices.

### 15.4.5 An I<sup>2</sup>C Master Reads an I<sup>2</sup>C Slave with a 7-bit Address in One Command Sequence



**Figure 15-9. An I<sup>2</sup>C Master Reading an I<sup>2</sup>C Slave with a 7-bit Address**

Figure 15-9 shows how an I<sup>2</sup>C master reads N bytes of data from an I<sup>2</sup>C 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*/*W* bit. When the *R*/*W* bit is 1, it indicates a READ operation. If the address of an I<sup>2</sup>C 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 15-9, 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*/*W* bit) is overwritten by byte0 marked red in Figure 15-9.

### 15.4.6 An I<sup>2</sup>C Master Reads an I<sup>2</sup>C Slave with a 10-bit Address in One Command Sequence

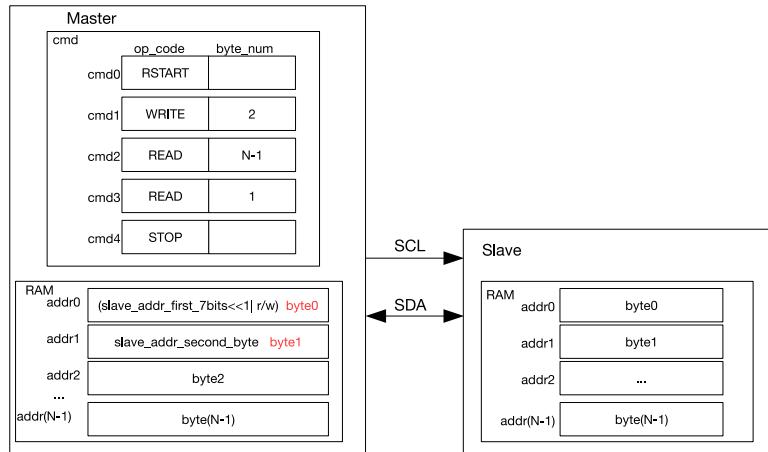


Figure 15-10. An I<sup>2</sup>C Master Reading an I<sup>2</sup>C Slave with a 10-bit Address

Figure 15-10 shows how an I<sup>2</sup>C master reads data from an I<sup>2</sup>C slave using 10-bit addressing. Unlike 7-bit addressing, in 10-bit addressing the WRITE command of the I<sup>2</sup>C 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 [15.4.2](#) for how to set this register.

### 15.4.7 An I<sup>2</sup>C Master Reads an I<sup>2</sup>C Slave with Two 7-bit Addresses in One Command Sequence

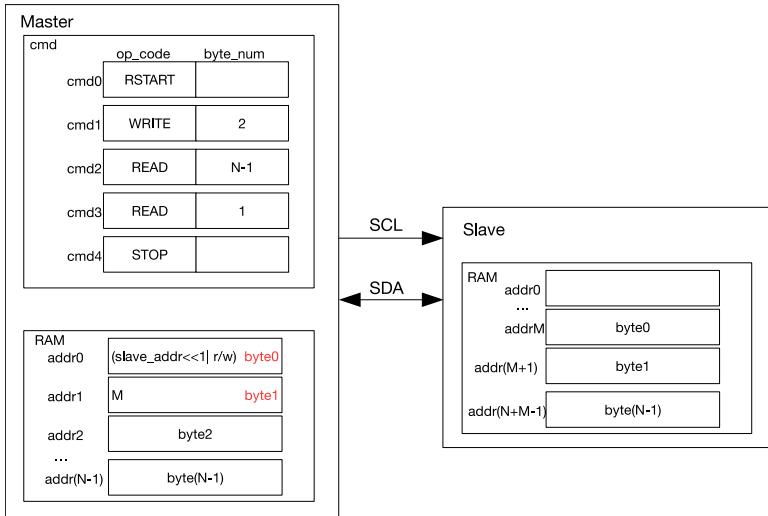


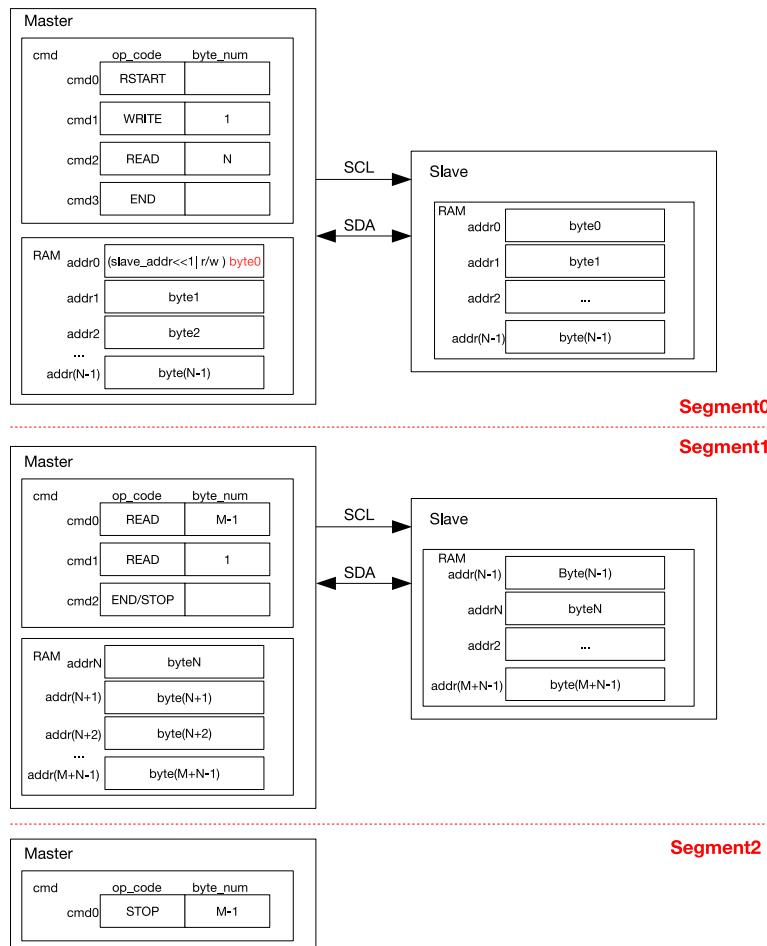
Figure 15-11. An I<sup>2</sup>C Master Reading N Bytes of Data from addrM of an I<sup>2</sup>C Slave with a 7-bit Address

Figure 15-11 shows how an I<sup>2</sup>C master reads data from specified addresses in an I<sup>2</sup>C 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.

### 15.4.8 An I<sup>2</sup>C Master Reads an I<sup>2</sup>C Slave with a 7-bit Address in Multiple Command Sequences



**Figure 15-12. An I<sup>2</sup>C Master Reading an I<sup>2</sup>C Slave with a 7-bit Address in Segments**

Figure 15-12 shows how an I<sup>2</sup>C master reads (N+M) bytes of data from an I<sup>2</sup>C 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.

## 15.5 Clock Stretching

In slave mode, the I<sup>2</sup>C 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 I<sup>2</sup>C controller matches the address sent via the SDA line.
2. RAM being full: In slave mode, RX RAM of the I<sup>2</sup>C controller is full.
3. Data all sent: In slave mode, TX RAM of the I<sup>2</sup>C 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.

## 15.6 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 I<sup>2</sup>C controller receives [I2C\\_NONFIFO\\_RX\\_THRES](#) bytes of data in non-FIFO mode.
- [I2C\\_TXFIFO\\_OVF\\_INT](#): Triggered when the I<sup>2</sup>C 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 I<sup>2</sup>C 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 I<sup>2</sup>C 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 I<sup>2</sup>C 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 I<sup>2</sup>C controller overflows.
- [I2C\\_TXFIFO\\_WM\\_INT](#): I<sup>2</sup>C 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\[4:0\]](#).

- I<sup>2</sup>C\_RXFIFO\_WM\_INT: I<sup>2</sup>C Rx FIFO watermark interrupt. Triggered when I<sup>2</sup>C\_FIFO\_PRT\_EN is 1 and the pointers of Rx FIFO are greater than I<sup>2</sup>C\_RXFIFO\_WM\_THRHD[4:0].

## 15.7 Base Address

Users can access the I<sup>2</sup>C Controller with base addresses in Table 60. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

**Table 60: I<sup>2</sup>C Controller Base Address**

	Bus to Access Peripheral	Base Address
I <sup>2</sup> C0	PeriBUS1	0x3F413000
	PeriBUS2	0x60013000
I <sup>2</sup> C1	PeriBUS1	0x3F427000
	PeriBUS2	0x60027000

## 15.8 Register Summary

Addresses in the following table are relative to I<sup>2</sup>C base addresses provided in Section 15.7.

Name	Description	Address	Access
<b>Timing Register</b>			
I <sup>2</sup> C_SCL_LOW_PERIOD_REG	Configures the low level width of the SCL clock	0x0000	R/W
I <sup>2</sup> C_SDA_HOLD_REG	Configures the hold time after a negative SCL edge	0x0030	R/W
I <sup>2</sup> C_SDA_SAMPLE_REG	Configures the sample time after a positive SCL edge	0x0034	R/W
I <sup>2</sup> C_SCL_HIGH_PERIOD_REG	Configures the high level width of the SCL clock	0x0038	R/W
I <sup>2</sup> C_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
I <sup>2</sup> C_SCL_RSTART_SETUP_REG	Configures the interval between the positive edge of SCL and the negative edge of SDA	0x0044	R/W
I <sup>2</sup> C_SCL_STOP_HOLD_REG	Configures the delay after the SCL clock edge for a stop condition	0x0048	R/W
I <sup>2</sup> C_SCL_STOP_SETUP_REG	Configures the delay between the SDA and SCL positive edge for a stop condition	0x004C	R/W
I <sup>2</sup> C_SCL_ST_TIME_OUT_REG	SCL status time out register	0x0098	R/W
I <sup>2</sup> C_SCL_MAIN_ST_TIME_OUT_REG	SCL main status time out register	0x009C	R/W
<b>Configuration Register</b>			
I <sup>2</sup> C_CTR_REG	Transmission setting	0x0004	R/W
I <sup>2</sup> C_TO_REG	Setting time out control for receiving data	0x000C	R/W
I <sup>2</sup> C_SLAVE_ADDR_REG	Local slave address setting	0x0010	R/W
I <sup>2</sup> C_FIFO_CONF_REG	FIFO configuration register	0x0018	R/W
I <sup>2</sup> C_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
<b>Status Register</b>			
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
<b>Interrupt Register</b>			
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
<b>Filter Register</b>			
I2C_SCL_FILTER_CFG_REG	SCL filter configuration register	0x0050	R/W
I2C_SDA_FILTER_CFG_REG	SDA filter configuration register	0x0054	R/W
<b>Command Register</b>			
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
<b>Version Register</b>			
I2C_DATE_REG	Version control register	0x00F8	R/W

## 15.9 Registers

**Register 15.1: I2C\_SCL\_LOW\_PERIOD\_REG (0x0000)**

31														14	13	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x00	Reset	

**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 15.2: I2C\_SDA\_HOLD\_REG (0x0030)**

31											10	9	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0	Reset

**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 15.3: I2C\_SDA\_SAMPLE\_REG (0x0034)**

31											10	9	0		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0x0	Reset

**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 15.4: I2C\_SCL\_HIGH\_PERIOD\_REG (0x0038)**

31	28	27		14	13	0	
0	0	0	0	0x00		0x00	Reset

**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 15.5: I2C\_SCL\_START\_HOLD\_REG (0x0040)**

31		10	9	0	
0	0	0	0	0	Reset

**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 15.6: I2C\_SCL\_RSTART\_SETUP\_REG (0x0044)**

31		10	9	0	
0	0	0	0	0	Reset

**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 15.7: I2C\_SCL\_STOP\_HOLD\_REG (0x0048)**

31	14	13	0
0	0	0	0x00      Reset

**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 15.8: I2C\_SCL\_STOP\_SETUP\_REG (0x004C)**

31	10	9	0
0	0	0	0x00      Reset

**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 15.9: I2C\_SCL\_ST\_TIME\_OUT\_REG (0x0098)**

31	24	23	0
0	0	0	0x0100      Reset

**I2C\_SCL\_ST\_TO** The threshold value of SCL\_FSM state unchanged period. (R/W)

## Register 15.10: 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 15.11: I2C\_CTR\_REG (0x0004)

	(reserved)																									
31																										
0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	1	Reset

**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 15.12: I2C\_TO\_REG (0x000C)**

The diagram shows the bit field layout of Register 15.12. It consists of a 32-bit register with the following bit descriptions:

- I2C\_TIME\_OUT\_EN**: A bit at index 25.
- (reserved)**: Bits 31 to 26.
- I2C\_TIME\_OUT\_VALUE**: Bits 24 to 23.
- 0**: Bit 22.
- 0x0000**: Bits 21 to 0.
- Reset**: A label indicating the reset value for bits 21 to 0.

31	25	24	23	0	0x0000	Reset
0	0	0	0	0	0	

**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 15.13: I2C\_SLAVE\_ADDR\_REG (0x0010)**

The diagram shows the bit field layout of Register 15.13. It consists of a 32-bit register with the following bit descriptions:

- I2C\_ADDR\_10BIT\_EN**: A bit at index 30.
- (reserved)**: Bits 31 to 29.
- I2C\_SLAVE\_ADDR**: Bits 15 to 14.
- 0**: Bit 13.
- 0x00**: Bits 12 to 0.
- Reset**: A label indicating the reset value for bits 12 to 0.

31	30	15	14	0	0x00	Reset
0	0	0	0	0	0	

**I2C\_SLAVE\_ADDR** When configured as an I<sup>2</sup>C 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 15.14: I2C\_FIFO\_CONF\_REG (0x0018)

31	27	26	25	20	19	14	13	12	11	10	9	5	4	0
0	0	0	0	0	1	0x15	0x15	0	0	0	0	0x4	0xb	Reset

**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 15.15: I<sup>2</sup>C\_SCL\_SP\_CONF\_REG (0x00A0)

31	8	7	6	5	1	0
0 0	0 0	0 0	0 0	0 0	0x0	0 Reset

**I<sup>2</sup>C\_SCL\_RST\_SLV\_EN** When I<sup>2</sup>C master is IDLE, set this bit to send out SCL pulses. The number of pulses equals to I<sup>2</sup>C\_SCL\_RST\_SLV\_NUM[4:0]. (R/W)

**I<sup>2</sup>C\_SCL\_RST\_SLV\_NUM** Configure the pulses of SCL generated in I<sup>2</sup>C master mode. Valid when I<sup>2</sup>C\_SCL\_RST\_SLV\_EN is 1. (R/W)

**I<sup>2</sup>C\_SCL\_PD\_EN** The power down enable bit for the I<sup>2</sup>C output SCL line. 1: Power down. 0: Not power down. Set I<sup>2</sup>C\_SCL\_FORCE\_OUT and I<sup>2</sup>C\_SCL\_PD\_EN to 1 to stretch SCL low. (R/W)

**I<sup>2</sup>C\_SDA\_PD\_EN** The power down enable bit for the I<sup>2</sup>C output SDA line. 1: Power down. 0: Not power down. Set I<sup>2</sup>C\_SDA\_FORCE\_OUT and I<sup>2</sup>C\_SDA\_PD\_EN to 1 to stretch SDA low. (R/W)

Register 15.16: I<sup>2</sup>C\_SCL\_STRETCH\_CONF\_REG (0x00A4)

31	12	11	10	9	0
0 0	0 0	0 0	0 0	0 0	0x0 Reset

**I<sup>2</sup>C\_STRETCH\_PROTECT\_NUM** Configure the period of I<sup>2</sup>C slave stretching SCL line. (R/W)

**I<sup>2</sup>C\_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 I<sup>2</sup>C\_SLAVE\_SCL\_STRETCH\_EN is 1 and stretch event happens. The stretch cause can be seen in I<sup>2</sup>C\_STRETCH\_CAUSE. (R/W)

**I<sup>2</sup>C\_SLAVE\_SCL\_STRETCH\_CLR** Set this bit to clear the I<sup>2</sup>C slave SCL stretch function. (WO)

**Register 15.17: I<sup>2</sup>C\_SR\_REG (0x0008)**

I <sup>2</sup> C_SR_REG Register Layout																						
31	30	28	27	26	24	23	18	17	16	15	14	13	8	7	6	5	4	3	2	1	0	Reset
0	0x0	0	0x0		0x0		0	0	0x0		0x0		0	0	0	0	0	0	0	0	0	Reset

**I<sup>2</sup>C\_RESP\_REC** The received ACK value in master mode or slave mode. 0: ACK; 1: NACK. (RO)

**I<sup>2</sup>C\_SLAVE\_RW** When in slave mode, 1: master reads from slave; 0: master writes to slave. (RO)

**I<sup>2</sup>C\_TIME\_OUT** When the I<sup>2</sup>C controller takes more than I<sup>2</sup>C\_TIME\_OUT clocks to receive a data bit, this field changes to 1. (RO)

**I<sup>2</sup>C\_ARB\_LOST** When the I<sup>2</sup>C controller loses control of SCL line, this register changes to 1. (RO)

**I<sup>2</sup>C\_BUS\_BUSY** 1: the I<sup>2</sup>C bus is busy transferring data; 0: the I<sup>2</sup>C bus is in idle state. (RO)

**I<sup>2</sup>C\_SLAVE\_ADDRESSED** When configured as an I<sup>2</sup>C 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)

**I<sup>2</sup>C\_BYTE\_TRANS** This field changes to 1 when one byte is transferred. (RO)

**I<sup>2</sup>C\_RXFIFO\_CNT** This field represents the amount of data needed to be sent. (RO)

**I<sup>2</sup>C\_STRETCH\_CAUSE** The cause of stretching SCL low in slave mode. 0: stretching SCL low at the beginning of I<sup>2</sup>C read data state. 1: stretching SCL low when I<sup>2</sup>C TX FIFO is empty in slave mode. 2: stretching SCL low when I<sup>2</sup>C RX FIFO is full in slave mode. (RO)

**I<sup>2</sup>C\_TXFIFO\_CNT** This field stores the amount of received data in RAM. (RO)

**I<sup>2</sup>C\_SCL\_MAIN\_STATE\_LAST** This field indicates the states of the I<sup>2</sup>C module state machine. 0: Idle; 1: Address shift; 2: ACK address; 3: RX data; 4: TX data; 5: Send ACK; 6: Wait ACK (RO)

**I<sup>2</sup>C\_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 15.18: I<sup>2</sup>C\_FIFO\_ST\_REG (0x0014)

31	30	29	22	21	20	19	15	14	10	9	5	4	0	
0	0	0x0	0	0	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 15.19: I<sup>2</sup>C\_DATA\_REG (0x001C)

31	8	7	0	
0	0	0	0	Reset

**I2C\_FIFO\_RDATA** The value of RX FIFO read data. (RO)

## Register 15.20: 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 15.21: 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 SCI MAIN ST TO INT C1 B** Set this bit to clear I2C SCI MAIN ST TO INT in

**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.

Register 15.22: I<sup>2</sup>C\_INT\_ENA\_REG (0x0028)

Bit Map:

31		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**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 15.23: I<sup>2</sup>C\_INT\_STATUS\_REG (0x002C)

31		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**I2C\_RXFIFO\_WM\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_RXFIFO\_WM\_INT interrupt. (RO)

**I2C\_TXFIFO\_WM\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_TXFIFO\_WM\_INT interrupt. (RO)

**I2C\_RXFIFO\_OVF\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_RXFIFO\_OVF\_INT interrupt. (RO)

**I2C\_END\_DETECT\_INT\_ST** The masked interrupt status bit for the I<sup>2</sup>C\_END\_DETECT\_INT interrupt. (RO)

**I2C\_BYTETRANS\_DONE\_INT\_ST** The masked interrupt status bit for the I<sup>2</sup>C\_END\_DETECT\_INT interrupt. (RO)

**I2C\_ARBITRATION\_LOST\_INT\_ST** The masked interrupt status bit for the I<sup>2</sup>C\_ARBITRATION\_LOST\_INT interrupt. (RO)

**I2C\_MST\_TXFIFO\_UDF\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_TRANS\_COMPLETE\_INT interrupt. (RO)

**I2C\_TRANS\_COMPLETE\_INT\_ST** The masked interrupt status bit for the I<sup>2</sup>C\_TRANS\_COMPLETE\_INT interrupt. (RO)

**I2C\_TIME\_OUT\_INT\_ST** The masked interrupt status bit for the I<sup>2</sup>C\_TIME\_OUT\_INT interrupt. (RO)

**I2C\_TRANS\_START\_INT\_ST** The masked interrupt status bit for the I<sup>2</sup>C\_TRANS\_START\_INT interrupt. (RO)

**I2C\_NACK\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_SLAVE\_STRETCH\_INT interrupt. (RO)

**I2C\_TXFIFO\_OVF\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_TXFIFO\_OVF\_INT interrupt. (RO)

**I2C\_RXFIFO\_UDF\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_RXFIFO\_UDF\_INT interrupt. (RO)

**I2C\_SCL\_ST\_TO\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_SCL\_ST\_TO\_INT interrupt. (RO)

**I2C\_SCL\_MAIN\_ST\_TO\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_SCL\_MAIN\_ST\_TO\_INT interrupt. (RO)

**I2C\_DET\_START\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_DET\_START\_INT interrupt. (RO)

**I2C\_SLAVE\_STRETCH\_INT\_ST** The masked interrupt status bit for I<sup>2</sup>C\_SLAVE\_STRETCH\_INT interrupt. (RO)

**Register 15.24: I2C\_SCL\_FILTER\_CFG\_REG (0x0050)**

31	(reserved)				5	4	3	0
0	0	0	0	0	0	0	0	0x0 Reset

**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 15.25: I2C\_SDA\_FILTER\_CFG\_REG (0x0054)**

31	(reserved)				5	4	3	0
0	0	0	0	0	0	0	0	0x0 Reset

**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 15.26: I2C\_COMMAND0\_REG (0x0058)**

31	30	(reserved)				14	13	0
0	0	0	0	0	0	0	0	0x00 Reset

**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 15.27: I2C\_COMD1\_REG (0x005C)**

31	30	(reserved)	14	13	0
0	0	0 0	0x00	Reset	

**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 15.28: I2C\_COMD2\_REG (0x0060)**

31	30	(reserved)	14	13	0
0	0	0 0	0x00	Reset	

**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 15.29: I<sup>2</sup>C\_COMD3\_REG (0x0064)

31	30	(reserved)	14	13	0
0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		0x00	Reset

**I<sup>2</sup>C\_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 I<sup>2</sup>C cmd structure for more Information. (R/W)

**I<sup>2</sup>C\_COMMAND3\_DONE** When command 3 is done in I<sup>2</sup>C Master mode, this bit changes to high level. (R/W)

Register 15.30: I<sup>2</sup>C\_COMD4\_REG (0x0068)

31	30	(reserved)	14	13	0
0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		0x00	Reset

**I<sup>2</sup>C\_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 I<sup>2</sup>C cmd structure for more Information. (R/W)

**I<sup>2</sup>C\_COMMAND4\_DONE** When command 4 is done in I<sup>2</sup>C Master mode, this bit changes to high level. (R/W)

**Register 15.31: I2C\_COMD5\_REG (0x006C)**

31	30	(reserved)	14	13	0
0	0	0	0	0	0x00 Reset

**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 15.32: I2C\_COMD6\_REG (0x0070)**

31	30	(reserved)	14	13	0
0	0	0	0	0	0x00 Reset

**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 15.33: I2C\_COMD7\_REG (0x0074)

31	30	(reserved)	14	13	0
0	0	0 0	0x00		Reset

**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 15.34: I2C\_COMD8\_REG (0x0078)

31	30	(reserved)	14	13	0
0	0	0 0	0x00		Reset

**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 15.35: I2C\_COMD9\_REG (0x007C)**

31	30	(reserved)	14	13	0
0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		0x00	Reset

**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 15.36: I2C\_COMD10\_REG (0x0080)**

31	30	(reserved)	14	13	0
0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0		0x00	Reset

**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 15.37: I<sup>2</sup>C\_COMMAND11\_REG (0x0084)

31	30														14	13	0
0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0													0x00	Reset	

**I<sup>2</sup>C\_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 I<sup>2</sup>C cmd structure for more Information. (R/W)

**I<sup>2</sup>C\_COMMAND11\_DONE** When command 11 is done in I<sup>2</sup>C Master mode, this bit changes to high level. (R/W)

Register 15.38: I<sup>2</sup>C\_COMMAND12\_REG (0x0088)

31	30														14	13	0
0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0													0x00	Reset	

**I<sup>2</sup>C\_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 I<sup>2</sup>C cmd structure for more Information. (R/W)

**I<sup>2</sup>C\_COMMAND12\_DONE** When command 12 is done in I<sup>2</sup>C Master mode, this bit changes to high level. (R/W)

**Register 15.39: I2C\_COMD13\_REG (0x008C)**

31	30	(reserved)										14	13	0
0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0										0x00		Reset

**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 15.40: I2C\_COMD14\_REG (0x0090)**

31	30	(reserved)										14	13	0
0	0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0										0x00		Reset

**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 15.41: I2C\_COMMD15\_REG (0x0094)**

31	30	(reserved)	14	13	0
0	0	0	0	0	0x00 Reset

**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 15.42: I2C\_DATE\_REG (0x00F8)**

31	0
0x19052000	Reset

**I2C\_DATE** This is the the version control register. (R/W)

# 16. AES Accelerator

## 16.1 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](#).

## 16.2 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

## 16.3 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 62 below.

**Table 62: 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 16.4 and Section 16.5 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.

## 16.4 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 63 below.

**Table 63: 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 64 below.

**Table 64: 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.

### 16.4.1 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.

### 16.4.2 Endianness

#### Text Endianness

In Typical AES working mode, the AES Accelerator uses cryptographic keys to encrypt and decrypt data in blocks of 128 bits. 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 be seen in Table 65.

**Table 65: Text Endianness Types for Typical AES**

Word Endian Controlling Bit	Byte Endian Controlling Bit	Plaintext/Ciphertext <sup>2</sup>				
0	0	State <sup>1</sup>		c		
		0	<a href="#">AES_TEXT_X_3_REG[31:24]</a>	<a href="#">AES_TEXT_X_2_REG[31:24]</a>	<a href="#">AES_TEXT_X_1_REG[31:24]</a>	
		r	1	<a href="#">AES_TEXT_X_3_REG[23:16]</a>	<a href="#">AES_TEXT_X_2_REG[23:16]</a>	<a href="#">AES_TEXT_X_1_REG[23:16]</a>
			2	<a href="#">AES_TEXT_X_3_REG[15:8]</a>	<a href="#">AES_TEXT_X_2_REG[15:8]</a>	<a href="#">AES_TEXT_X_1_REG[15:8]</a>
			3	<a href="#">AES_TEXT_X_3_REG[7:0]</a>	<a href="#">AES_TEXT_X_2_REG[7:0]</a>	<a href="#">AES_TEXT_X_1_REG[7:0]</a>
		State		c		
		0	<a href="#">AES_TEXT_X_3_REG[7:0]</a>	<a href="#">AES_TEXT_X_2_REG[7:0]</a>	<a href="#">AES_TEXT_X_1_REG[7:0]</a>	
		1	<a href="#">AES_TEXT_X_3_REG[15:8]</a>	<a href="#">AES_TEXT_X_2_REG[15:8]</a>	<a href="#">AES_TEXT_X_1_REG[15:8]</a>	
		2	<a href="#">AES_TEXT_X_3_REG[23:16]</a>	<a href="#">AES_TEXT_X_2_REG[23:16]</a>	<a href="#">AES_TEXT_X_1_REG[23:16]</a>	
		3	<a href="#">AES_TEXT_X_3_REG[31:24]</a>	<a href="#">AES_TEXT_X_2_REG[31:24]</a>	<a href="#">AES_TEXT_X_1_REG[31:24]</a>	
1	1	State		c		
		0	<a href="#">AES_TEXT_X_0_REG[31:24]</a>	<a href="#">AES_TEXT_X_1_REG[31:24]</a>	<a href="#">AES_TEXT_X_2_REG[31:24]</a>	
		1	<a href="#">AES_TEXT_X_0_REG[23:16]</a>	<a href="#">AES_TEXT_X_1_REG[23:16]</a>	<a href="#">AES_TEXT_X_2_REG[23:16]</a>	
		2	<a href="#">AES_TEXT_X_0_REG[15:8]</a>	<a href="#">AES_TEXT_X_1_REG[15:8]</a>	<a href="#">AES_TEXT_X_2_REG[15:8]</a>	
		3	<a href="#">AES_TEXT_X_0_REG[7:0]</a>	<a href="#">AES_TEXT_X_1_REG[7:0]</a>	<a href="#">AES_TEXT_X_2_REG[7:0]</a>	
		0	<a href="#">AES_TEXT_X_0_REG[7:0]</a>	<a href="#">AES_TEXT_X_1_REG[7:0]</a>	<a href="#">AES_TEXT_X_2_REG[7:0]</a>	
		1	<a href="#">AES_TEXT_X_0_REG[15:8]</a>	<a href="#">AES_TEXT_X_1_REG[15:8]</a>	<a href="#">AES_TEXT_X_2_REG[15:8]</a>	
		2	<a href="#">AES_TEXT_X_0_REG[23:16]</a>	<a href="#">AES_TEXT_X_1_REG[23:16]</a>	<a href="#">AES_TEXT_X_2_REG[23:16]</a>	
		3	<a href="#">AES_TEXT_X_0_REG[31:24]</a>	<a href="#">AES_TEXT_X_1_REG[31:24]</a>	<a href="#">AES_TEXT_X_2_REG[31:24]</a>	

**Note:**

1. The definition of "State" is described in Section 3.4 The State in [NIST FIPS 197](#).
2. Where,
  - When  $x = \text{IN}$ , 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  $x = \text{OUT}$ , 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 be seen in [Table 66](#), [Table 67](#), and [Table 68](#).

Table 66: Key Endianness Types for AES-128 Encryption and Decryption

AES_ENDIAN_REG[1]	AES_ENDIAN_REG[0]	Bit <sup>2</sup>	w[0]	w[1]	w[2]	w[3] <sup>1</sup>
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 67: Key Endianness Types for AES-192 Encryption and Decryption

AES_ENDIAN_REG[1]	AES_ENDIAN_REG[0]	Bit <sup>2</sup>	w[0]	w[1]	w[2]	w[3]	w[4]	w[5] <sup>1</sup>
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 68: Key Endianness Types for AES-256 Encryption and Decryption

AES_ENDIAN_REG[1]	AES_ENDIAN_REG[0]	Bit <sup>2</sup>	w[0]	w[1]	w[2]	w[3]	w[4]	w[5]	w[6]	w[7] <sup>1</sup>
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].

### 16.4.3 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.

## 16.5 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 69 below.

**Table 69: 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 70 below.

**Table 70: 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.

### 16.5.1 Key, Plaintext, and Ciphertext

#### 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 71 below.

**Table 71: TEXT-PADDING**

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

### 16.5.2 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 72 below.

**Table 72: 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.

### 16.5.3 Standard Incrementing Function

AES accelerator provides two Standard Incrementing Functions for the CTR block operation, which are  $\text{INC}_{32}$  and  $\text{INC}_{128}$  Standard Incrementing Functions. By setting the [AES\\_INC\\_SEL\\_REG](#) register to 0 or 1, users can choose the  $\text{INC}_{32}$  or  $\text{INC}_{128}$  functions respectively. For details on the Standard Incrementing Function, please see Chapter B.1 The Standard Incrementing Function in [NIST SP 800-38A](#).

### 16.5.4 Block Number

Register [AES\\_BLOCK\\_NUM\\_REG](#) stores the Block Number of plaintext  $P$  or ciphertext  $C$ . The length of this register equals to  $\text{length}(\text{TEXT-PADDING}(P))/128$  or  $\text{length}(\text{TEXT-PADDING}(C))/128$ . The AES Accelerator only uses this register when working in the DMA-AES mode.

### 16.5.5 Initialization Vector

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

Both IV and ICB are 128-bit strings, which can be divided into Byte0, Byte1, Byte2 … Byte15 (from left to right). [AES\\_IV\\_MEM](#) stores data following the Endianness pattern presented in Table 72, 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](#).

### 16.5.6 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 69.
  - Initialize the [AES\\_BLOCK\\_NUM\\_REG](#) register. For details, see Section 16.5.4.
  - 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 occurred 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.

### 16.5.7 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 16.5.4.
  - Initialize the [AES\\_AAD\\_BLOCK\\_NUM\\_REG](#) register. Details about this register are described in Section 16.6.4.
  - Initialize the [AES\\_REMAINDER\\_BIT\\_NUM\\_REG](#) register. Details about this register is described in Section 16.6.5.
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 70. 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 70.
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 writing 1 to the [AES\\_INT\\_CLR\\_REG](#) register, if any AES interrupt occurred 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.

## 16.6 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 16-1 below demonstrates how GCM encryption is implemented in the AES Accelerator of ESP32-S2.

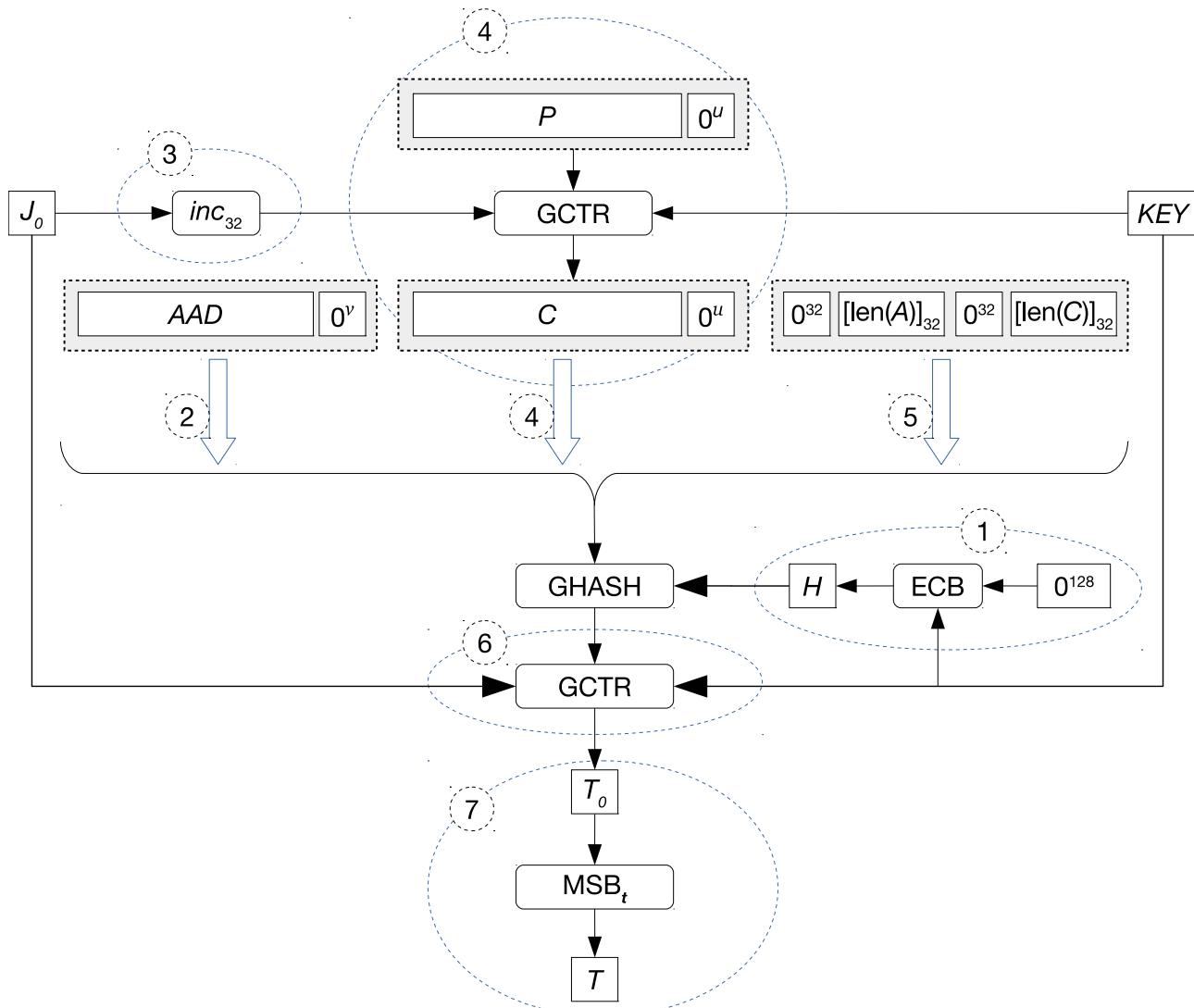


Figure 16-1. GCM Encryption Process

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<sub>32</sub> 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 ciphertext  $C$ .

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  $\text{MSB}_t$  Algorithm to obtain the final result Authenticated Tag  $T$ .

The only difference between GCM decryption and GCM encryption lies in Step 4 in Figure 16-1. 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 ciphertext in GCM decryption. For details, please see [NIST SP 800-38D](#).

### 16.6.1 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 16-1. 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 72.

### 16.6.2 $J_0$

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 16-1. 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 72.

### 16.6.3 Authenticated Tag

Authenticated Tag (Tag for short) is one of the key results of GCM computation, which is demonstrated in Step 7 of Figure 16-1. The value of Tag is determined by the length of Authenticated Tag  $t$  ( $1 \leq t \leq 128$ ):

- 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 72.
- When  $1 \leq t < 128$ , 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,  $\text{MSB}_4(1101011010) = 1110$  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.

### 16.6.4 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  $\text{length}(\text{TEXT-PADDING}(AAD))/128$ . AES Accelerator only uses this register when

working in the DMA-AES mode.

### 16.6.5 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/ciphertext. The value stored in this register equals to  $\text{length}(P) \% 128$  or  $\text{length}(C) \% 128$ . 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 ciphertext, 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.

## 16.7 Base Address

Users can access AES with two base addresses, which can be seen in Table 73. For more information about accessing peripherals from different buses please see Chapter [1 System and Memory](#).

**Table 73: AES Accelerator Base Address**

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F43A000
PeriBUS2	0x6003A000

## 16.8 Memory Summary

Both the starting address and ending address in the following table are relative to AES base addresses provided in Section [16.7](#).

Name	Description	Length	Starting Address	Ending Address	Access
<a href="#">AES_IV_MEM</a>	Memory IV	16 bytes	0x0050	0x005F	R/W
<a href="#">AES_H_MEM</a>	Memory H	16 bytes	0x0060	0x006F	RO
<a href="#">AES_J0_MEM</a>	Memory J0	16 bytes	0x0070	0x007F	R/W
<a href="#">AES_T0_MEM</a>	Memory T0	16 bytes	0x0080	0x008F	RO

## 16.9 Register Summary

The addresses in the following table are relative to AES base addresses provided in Section [16.7](#).

Name	Description	Address	Access
<b>Key Registers</b>			
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
<b>TEXT_IN Registers</b>			
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
<b>TEXT_OUT Registers</b>			
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
<b>Configuration Registers</b>			
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
<b>Controlling / Status Registers</b>			
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
<b>Interruption Registers</b>			
AES_INT_CLR_REG	DMA-AES interrupt clear register	0x00AC	WO
AES_INT_ENA_REG	DMA-AES interrupt enable register	0x00B0	R/W

## 16.10 Registers

**Register 16.1: AES\_KEY<sub>n</sub>\_REG ( $n$ : 0-7) (0x0000+4 $*n$ )**

31	0
0x0000000000	Reset

**AES\_KEY<sub>n</sub>\_REG ( $n$ : 0-7)** Stores AES keys. (R/W)

**Register 16.2: AES\_TEXT\_IN<sub>m</sub>\_REG ( $m$ : 0-3) (0x0020+4 $*m$ )**

31	0
0x0000000000	Reset

**AES\_TEXT\_IN<sub>m</sub>\_REG ( $m$ : 0-3)** Stores the source data when the AES Accelerator operates in the Typical AES working mode. (R/W)

**Register 16.3: AES\_TEXT\_OUT<sub>m</sub>\_REG ( $m$ : 0-3) (0x0030+4 $*m$ )**

31	0
0x0000000000	Reset

**AES\_TEXT\_OUT<sub>m</sub>\_REG ( $m$ : 0-3)** Stores the result data when the AES Accelerator operates in the Typical AES working mode. (RO)

**Register 16.4: AES\_MODE\_REG (0x0040)**

31	(reserved)	3	2	0
0x0000000000		0		Reset

**AES\_MODE** Defines the operation type of the AES Accelerator operating under the Typical AES working mode. For details, see Table 63. (R/W)

**Register 16.5: AES\_ENDIAN\_REG (0x0044)**

						AES_ENDIAN
(reserved)						
31	6	5	0	0	0	Reset
0x00000000						

**AES\_ENDIAN** Defines the endianness of input and output texts. For details, please see Table 65.  
(R/W)

**Register 16.6: AES\_DMA\_ENABLE\_REG (0x0090)**

						AES_DMA_ENABLE
(reserved)						
31	1	0	0	0	0	Reset
0x00000000						

**AES\_DMA\_ENABLE** Defines the working mode of the AES Accelerator. For details, see Table 62.  
(R/W)

**Register 16.7: AES\_BLOCK\_MODE\_REG (0x0094)**

						AES_BLOCK_MODE
(reserved)						
31	3	2	0	0	0	Reset
0x00000000						

**AES\_BLOCK\_MODE** Defines the operation type of the AES Accelerator operating under the DMA-AES working mode. For details, see Table 69. (R/W)

**Register 16.8: AES\_BLOCK\_NUM\_REG (0x0098)**

31	0
0x00000000	

**AES\_BLOCK\_NUM** Stores the Block Number of plaintext or ciphertext when the AES Accelerator operates under the DMA-AES working mode. For details, see Section 16.5.4. (R/W)

**Register 16.9: AES\_INC\_SEL\_REG (0x009C)**

(reserved)		AES_INC_SEL	
31		1   0	Reset
0x00000000		0	

**AES\_INC\_SEL** Defines the Standard Incrementing Function for CTR block operation. Set this bit to 0 or 1 to choose INC<sub>32</sub> or INC<sub>128</sub>. (R/W)

**Register 16.10: AES\_AAD\_BLOCK\_NUM\_REG (0x00A0)**

(reserved)		AES_AAD_BLOCK_NUM	
31		0	Reset
0x00000000		0	

**AES\_AAD\_BLOCK\_NUM** Stores the ADD Block Number for the GCM operation. (R/W) For details, see Section [16.6.4](#).

**Register 16.11: AES\_REMAINDER\_BIT\_NUM\_REG (0x00A4)**

(reserved)		AES_REMAINDER_BIT_NUM	
31		7   6   0	Reset
0x00000000		0	

**AES\_REMAINDER\_BIT\_NUM** Stores the Remainder Bit Number for the GCM operation. For details, see Section [16.6.5](#). (R/W)

**Register 16.12: AES\_TRIGGER\_REG (0x0048)**

(reserved)		AES_TRIGGER	
31		1   0	Reset
0x00000000		x	

**AES\_TRIGGER** Set this bit to 1 to start AES operation. (WO)

**Register 16.13: AES\_STATE\_REG (0x004C)**

(reserved)		
31	2   1   0	
0x00000000	0x0	Reset

**AES\_STATE** Stores the working status of the AES Accelerator. For details, see Table 64 for Typical AES working mode and Table 70 for DMA AES working mode. (RO)

**Register 16.14: AES\_CONTINUE\_REG (0x00A8)**

(reserved)		
31	1   0	
0x00000000	x	Reset

**AES\_CONTINUE** Set this bit to 1 to continue AES operation. (WO)

**Register 16.15: AES\_DMA\_EXIT\_REG (0x00B8)**

(reserved)		
31	1   0	
0x00000000	x	Reset

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

**Register 16.16: AES\_INT\_CLR\_REG (0x00AC)**

(reserved)		
31	1   0	
0x00000000	x	Reset

**AES\_INT\_CLR** Set this bit to 1 to clear AES interrupt. (WO)

**Register 16.17: AES\_INT\_ENA\_REG (0x00B0)**

		AES_INT_ENA
31	0	
0x00000000	0	Reset

**AES\_INT\_ENA** Set this bit to 1 to enable AES interrupt and 0 to disable interrupt. (R/W)

# 17. SHA Accelerator

## 17.1 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](#).

## 17.2 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

## 17.3 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 76.

**Table 76: SHA Accelerator Working Mode**

Working Mode	Configuration Method
Typical SHA	Set <a href="#">SHA_START_REG</a> to 1
DMA-SHA	Set <a href="#">SHA_DMA_START_REG</a> to 1

Users can choose hash algorithms by configuring the [SHA\\_MODE\\_REG](#) register. For details, please see Table 77.

**Table 77: SHA Hash Algorithm**

Hash Algorithm	<a href="#">SHA_MODE_REG</a> 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.

## 17.4 Function Description

SHA accelerator can generate the message digest via two steps: the [Preprocessing](#) and [Hash](#) operation.

### 17.4.1 Preprocessing

Preprocessing consists of three steps: [padding the message](#), [parsing the message into message blocks](#) and [setting the initial hash value](#).

#### 17.4.1.1 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  
 $m + 1 + k \equiv 448 \bmod 512$ ;
  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  
 $m + 1 + k \equiv 896 \bmod 1024$ ;
    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](#).

### 17.4.1.2 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.

### 17.4.1.3 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  $1 \leq t \leq 9$ , then  $t\_length = 7'h48$  and  $t\_string$  is padded in the following format:

8'h3t <sub>0</sub>	1'b1	23'b0
--------------------	------	-------

where  $t_0 = t$ .

For example, if  $t = 8$ , then  $t_0 = 8$  and  $t\_string = 32'h38800000$ .

- If  $10 \leq t \leq 99$ , then  $t\_length = 7'h50$  and  $t\_string$  is padded in the following format:

8'h3t <sub>1</sub>	8'h3t <sub>0</sub>	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  $100 \leq t < 512$ , then  $t\_length = 7'h58$  and  $t\_string$  is padded in the following format:

8'h3t <sub>2</sub>	8'h3t <sub>1</sub>	8'h3t <sub>0</sub>	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”.

## 17.4.2 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.

### 17.4.2.1 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 [77](#).
2. Process the current message block.
  - (a) Write the current message block in registers [SHA\\_M\\_n\\_REG](#);
  - (b) Start the SHA accelerator <sup>1</sup>:
    - 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_Hn_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_Hn_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_Hn_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\_stirring 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 [17.4.1.3](#).
  - (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_Mn_REG`;
  - (b) Start the SHA accelerator <sup>1</sup>:
    - 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_Hn_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_Hn_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_Hn_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_Hn_REG`.

**Note:**

1. In Step 2b, the software can also write the next message block (to be processed) in registers `SHA_Mn_REG`, if any, while the hardware starts SHA computation, to save time.

### 17.4.2.2 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 [17.4.1.2](#).

**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 [77](#).
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<sub>n</sub>\\_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<sub>n</sub>\\_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 [17.4.1.3](#).
  - (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](#).

### 17.4.3 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 81 below:

**Table 81: The Storage and Length of Message Digests from Different Algorithms**

Hash Algorithm	Length of Message Digest (in bits)	Storage <sup>1</sup>
SHA-1	160	<a href="#">SHA_H_0_REG</a> ~ <a href="#">SHA_H_4_REG</a>
SHA-224	224	<a href="#">SHA_H_0_REG</a> ~ <a href="#">SHA_H_6_REG</a>
SHA-256	256	<a href="#">SHA_H_0_REG</a> ~ <a href="#">SHA_H_7_REG</a>
SHA-384	384	<a href="#">SHA_H_0_REG</a> ~ <a href="#">SHA_H_11_REG</a>
SHA-512	512	<a href="#">SHA_H_0_REG</a> ~ <a href="#">SHA_H_15_REG</a>
SHA-512/224	224	<a href="#">SHA_H_0_REG</a> ~ <a href="#">SHA_H_6_REG</a>
SHA-512/256	256	<a href="#">SHA_H_0_REG</a> ~ <a href="#">SHA_H_7_REG</a>
SHA-512/ $t$ <sup>2</sup>	$t$	<a href="#">SHA_H_0_REG</a> ~ <a href="#">SHA_H_x_REG</a>

**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 [17.4.1.2](#).
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  $x = \text{roundup}(t/32)-1$ . 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](#).

### 17.4.4 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.

## 17.5 Base Address

Users can access SHA with two base addresses, which can be seen in Table 82. For more information about accessing peripherals from different buses, please see Chapter 1 *System and Memory*.

**Table 82: Base Address**

Bus	Base Address
PeriBUS1	0x3F43B000
PeriBUS2	0x6003B000

## 17.6 Register Summary

Name	Description	Address	Access
<b>Control/Status registers</b>			
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
<b>Version Register</b>			
SHA_DATE_REG	Version control register	0x002C	R/W
<b>Configuration Registers</b>			
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
<b>Memories</b>			
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

## 17.7 Registers

**Register 17.1: SHA\_START\_REG (0x0010)**

31	1	0
0 0	0	Reset

**SHA\_START** Write 1 to start Typical SHA calculation. (WO)

**Register 17.2: SHA\_CONTINUE\_REG (0x0014)**

31	1	0
0 0	0	Reset

**SHA\_CONTINUE** Write 1 to continue Typical SHA calculation. (WO)

**Register 17.3: SHA\_BUSY\_REG (0x0018)**

31	1	0
0 0	0	Reset

**SHA\_BUSY\_STATE** Indicates the states of SHA accelerator. (RO) 1'h0: idle 1'h1: busy

**Register 17.4: SHA\_DMA\_START\_REG (0x001C)**

31	1	0
0 0	0	Reset

**SHA\_DMA\_START** Write 1 to start DMA-SHA calculation. (WO)

## Register 17.5: SHA\_DMA\_CONTINUE\_REG (0x0020)

**SHA\_DMA\_CONTINUE** Write 1 to continue DMA-SHA calculation. (WO)

## Register 17.6: SHA\_INT\_CLEAR\_REG (0x0024)

**SHA\_CLEAR\_INTERRUPT** Clears DMA-SHA interrupt. (WO)

## Register 17.7: SHA\_INT\_ENA\_REG (0x0028)

**SHA\_INTERRUPT\_ENA** Enables DMA-SHA interrupt. (R/W)

## Register 17.8: SHA\_DATE\_REG (0x002C)

(reserved)			SHA_DATE	
31	30	29		0
0	0		0x20190402	Reset

**SHA\_DATE** Version control register. (R/W)

**Register 17.9: SHA\_MODE\_REG (0x0000)**

(reserved)			SHA_MODE		
31			3	2	0
0 0			0x0 0 Reset		

**SHA\_MODE** Defines the SHA algorithm. For details, please see Table 77. (R/W)

**Register 17.10: SHA\_T\_STRING\_REG (0x0004)**

SHA_T_STRING					
31				0	
0x0000000			0x0 0 Reset		

**SHA\_T\_STRING** Defines t\_string for calculating the initial Hash value for SHA-512/t. (R/W)

**Register 17.11: SHA\_T\_LENGTH\_REG (0x0008)**

(reserved)			SHA_T_LENGTH		
31				6	5
0 0			0x0 0 Reset		

**SHA\_T\_LENGTH** Defines t\_length for calculating the initial Hash value for SHA-512/t. (R/W)

**Register 17.12: SHA\_DMA\_BLOCK\_NUM\_REG (0x000C)**

(reserved)			SHA_DMA_BLOCK_NUM		
31				6	5
0 0			0x0 0 Reset		

**SHA\_DMA\_BLOCK\_NUM** Defines the DMA-SHA block number. (R/W)

**Register 17.13: SHA\_H<sub>n</sub>\_REG (*n*: 0-15) (0x0040+4\**n*)**

SHA\_H<sub>n</sub>

31	0
0x000000	Reset

**SHA\_H<sub>n</sub>** Stores the *n*th 32-bit piece of the Hash value. (R/W)

**Register 17.14: SHA\_M<sub>n</sub>\_REG (*n*: 0-31) (0x0080+4\**n*)**

SHA\_M<sub>n</sub>

31	0
0x000000	Reset

**SHA\_M<sub>n</sub>** Stores the *n*th 32-bit piece of the message. (R/W)

# 18. RSA Accelerator

## 18.1 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.

## 18.2 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

## 18.3 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.

### 18.3.1 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 —  $\bar{r}$  and  $M'$ , which need to be calculated in advance by software.

RSA Accelerator supports operands of length  $N = 32 \times x$ , where  $x \in \{1, 2, 3, \dots, 128\}$ . The bit lengths of arguments  $Z$ ,  $X$ ,  $Y$ ,  $M$ , and  $\bar{r}$  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:

$$n = \frac{N}{32}$$

$$Z = (Z_{n-1} Z_{n-2} \cdots Z_0)_b$$

$$X = (X_{n-1} X_{n-2} \cdots X_0)_b$$

$$Y = (Y_{n-1} Y_{n-2} \cdots Y_0)_b$$

$$M = (M_{n-1} M_{n-2} \cdots M_0)_b$$

$$\bar{r} = (\bar{r}_{n-1} \bar{r}_{n-2} \cdots \bar{r}_0)_b$$

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

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

If we define  $R = b^n$ , the additional arguments can be calculated as  $\bar{r} = R^2 \bmod M$ , 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 18.3.4.
3. Write  $X_i$ ,  $Y_i$ ,  $M_i$  and  $\bar{r}_i$  for  $i \in \{0, 1, \dots, n\}$  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  $i \in \{0, 1, \dots, n\}$  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  $\bar{r}_i$  in [RSA\\_Z\\_MEM](#) computation are overwritten, and only these overwritten memory blocks need to be re-initialized before starting another computation.

### 18.3.2 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  $\bar{r}$  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  $\bar{r}_i$  for  $i \in \{0, 1, \dots, n\}$  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  $i \in \{0, 1, \dots, n\}$  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  $\bar{r}_i$  in memory [RSA\\_Z\\_MEM](#) has already been overwritten, and only this overwritten memory block needs to be re-initialized before starting another computation.

### 18.3.3 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  $x \in \{0, 1, \dots, 64\}$ . 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  $i \in \{0, 1, \dots, n\}$  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  $i \in \{0, 1, \dots, n\}$  to the address of the  $i$  words of the [RSA\\_X\\_MEM](#) register. Note that  $Y_i$  for  $i \in \{0, 1, \dots, n\}$  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  $i \in \{0, 1, \dots, n\}$  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  $\bar{r}_i$  in memory [RSA\\_Z\\_MEM](#) has already been overwritten, and only this overwritten memory block needs to be re-initialized before starting another computation.

### 18.3.4 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

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

where,

- $N$  is the length of  $Y$ ,
- $\tilde{Y}_t$  is 1,
- $\tilde{Y}_{N-1}, \tilde{Y}_{N-2}, \dots, \tilde{Y}_{t+1}$  are all equal to 0,
- and  $\tilde{Y}_{t-1}, \tilde{Y}_{t-2}, \dots, \tilde{Y}_0$  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  $\tilde{Y}_{t-1}\tilde{Y}_{t-2}, \dots, \tilde{Y}_0$  is  $t - m$ .

When the acceleration options are enabled, the RSA accelerator:

- SEARCH Option
  - The accelerator ignores the bit positions of  $\tilde{Y}_i$ , where  $i > \alpha$ . Search position  $\alpha$  is set by configuring the [RSA\\_SEARCH\\_POS\\_REG](#) register. The maximum value of  $\alpha$  is  $N-1$ , which leads to the same result when this acceleration option is disabled. The best acceleration performance can be achieved by setting  $\alpha$  to  $t$ , in which case, all the  $\tilde{Y}_{N-1}, \tilde{Y}_{N-2}, \dots, \tilde{Y}_{t+1}$  of 0s are ignored during the calculation. Note that if you set  $\alpha$  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 84 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,  $\alpha$  is set to 16 when the SEARCH option is enabled.

**Table 84: Acceleration Performance**

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%

## 18.4 Base Address

Users can access RSA with two base addresses, which can be seen in Table 85. For more information about accessing peripherals from different buses please see Chapter 1 *System and Memory*.

**Table 85: RSA Base Address**

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F43C000
PeriBUS2	0x6003C000

## 18.5 Memory Summary

Both the starting address and ending address in the following table are relative to RSA base addresses provided in Section 18.4.

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

## 18.6 Register Summary

The addresses in the following table are relative to RSA base addresses provided in Section 18.4.

Name	Description	Address	Access
<b>Configuration Registers</b>			
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
<b>Status/Control Registers</b>			
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
<b>Interrupt Registers</b>			
RSA_CLEAR_INTERRUPT_REG	RSA clear interrupt register	0x081C	WO
RSA_INTERRUPT_ENA_REG	RSA interrupt enable register	0x082C	R/W
<b>Version Register</b>			
RSA_DATE_REG	Version control register	0x0830	R/W

## 18.7 Registers

**Register 18.1: RSA\_M\_PRIME\_REG (0x800)**

31	0
0x0000000000	Reset

**RSA\_M\_PRIME\_REG** Stores M'.(R/W)

**Register 18.2: RSA\_MODE\_REG (0x804)**

31	7	6	0
0 0	0	0 0	Reset

**RSA\_MODE** Stores the mode of modular exponentiation. (R/W)

**Register 18.3: RSA\_CLEAN\_REG (0x808)**

31	1	0
0 0	0	Reset

**RSA\_CLEAN** The content of this bit is 1 when memories complete initialization. (RO)

**Register 18.4: RSA\_MODEXP\_START\_REG (0x80C)**

31	1	0
0 0	0	Reset

**RSA\_MODEXP\_START** Set this bit to 1 to start the modular exponentiation. (WO)

**Register 18.5: RSA\_MODMULT\_START\_REG (0x0810)**

31	0
0 0	0 Reset

**RSA\_MODMULT\_START** Set this bit to 1 to start the modular multiplication. (WO)

**Register 18.6: RSA\_MULT\_START\_REG (0x0814)**

31	0
0 0	0 Reset

**RSA\_MULT\_START** Set this bit to 1 to start the multiplication. (WO)

**Register 18.7: RSA\_IDLE\_REG (0x0818)**

31	0
0 0	0 Reset

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

**Register 18.8: RSA\_CLEAR\_INTERRUPT\_REG (0x081C)**

31	0
0 0	0 Reset

**RSA\_CLEAR\_INTERRUPT** Set this bit to 1 to clear the RSA interrupts. (WO)

**Register 18.9: RSA\_CONSTANT\_TIME\_REG (0x0820)**

31	1	0
0 1	Reset	

**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 18.10: RSA\_SEARCH\_ENABLE\_REG (0x0824)**

31	1	0
0 0	Reset	

**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 18.11: RSA\_SEARCH\_POS\_REG (0x0828)**

31	12	11	0
0 0	0x000		Reset

**RSA\_SEARCH\_POS** Is used to configure the starting address when the acceleration option of search is used. (R/W)

**Register 18.12: RSA\_INTERRUPT\_ENA\_REG (0x082C)**

RSA_INTERRUPT_ENA																															
(reserved)																															
31	0																													1	0

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

**Register 18.13: RSA\_DATE\_REG (0x0830)**

RSA_DATE																															
(reserved)																														0	
31	30	29	0x20190425																												Reset

**RSA\_DATE** Version control register. (R/W)

# 19. Random Number Generator

## 19.1 Introduction

The ESP32-S2 contains a true random number generator, which generates random numbers that can be used for cryptographical operations, among other things.

## 19.2 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.

## 19.3 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.

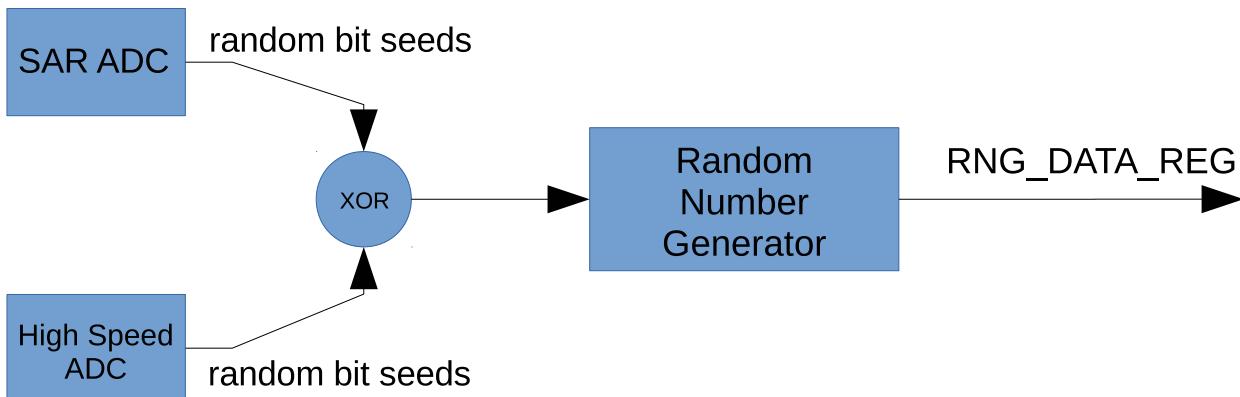


Figure 19-1. Noise Source

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.

## 19.4 Base Address

Users can access the random number generator with two base addresses, which can be seen in Table 88. For more information about accessing peripherals from different buses, please see Chapter 1 *System and Memory*.

**Table 88: Random Number Generator Base Address**

Bus	Base Address
PeriBUS1	0x3FF75000
PeriBUS2	0x60035000

## 19.5 Register Summary

The addresses in the following table are relative to the random number generator base addresses provided in Section 19.4.

Name	Description	Address	Access
RNG_DATA_REG	Random number data	0x0110	RO

## 19.6 Register

The address in this section is relative to the random number generator base addresses provided in Section 19.4.

**Register 19.1: RNG\_DATA\_REG (0x0110)**

31	0
0x00000000	Reset

**RNG\_DATA** Random number source. (RO)

# 20. External Memory Encryption and Decryption

## 20.1 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.

## 20.2 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

## 20.3 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 20-1.

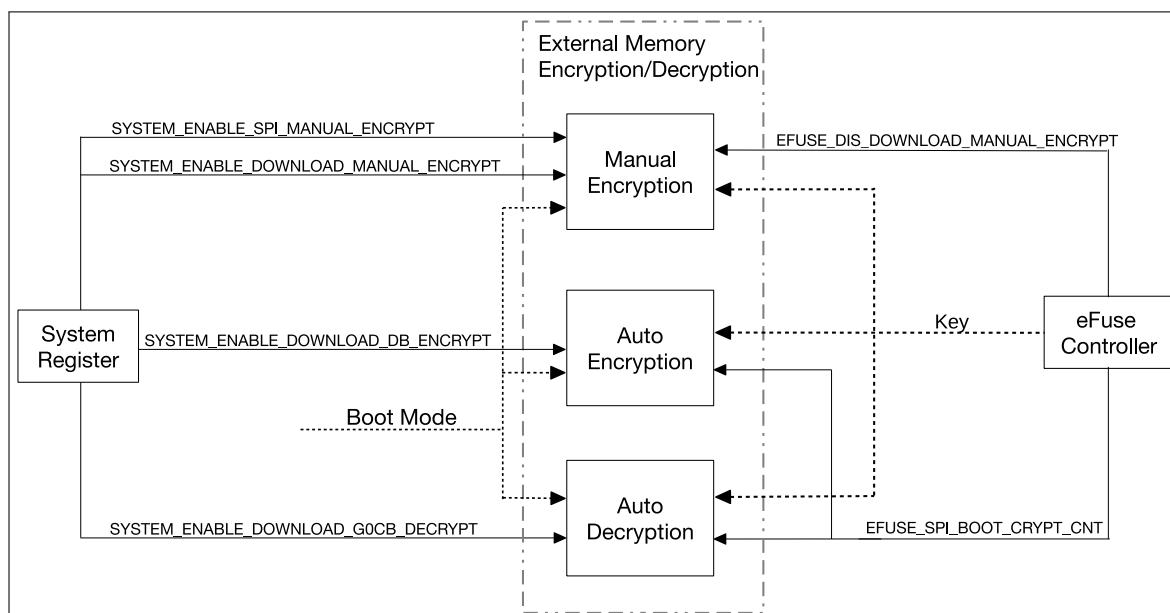


Figure 20-1. Architecture of the External Memory Encryption and Decryption Module

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](#).

### 20.3.1 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](#).

### 20.3.2 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<sub>A</sub>, which refers to the block that has the key purpose set to EFUSE\_KEY\_PURPOSE\_XTS\_AES\_256\_KEY\_1. Block<sub>A</sub> contains 256-bit *Key<sub>A</sub>*.
- Block<sub>B</sub>, which refers to the block that has the key purpose set to EFUSE\_KEY\_PURPOSE\_XTS\_AES\_256\_KEY\_2. Block<sub>B</sub> contains 256-bit *Key<sub>B</sub>*.
- Block<sub>C</sub>, which refers to the block that has the key purpose set to EFUSE\_KEY\_PURPOSE\_XTS\_AES\_128\_KEY. Block<sub>C</sub> contains 256-bit *Key<sub>C</sub>*.

Table 90 shows how the *Key* is generated, depending on whether Block<sub>A</sub>, Block<sub>B</sub>, and Block<sub>C</sub> exists or not.

**Table 90: Key**

Block <sub>A</sub>	Block <sub>B</sub>	Block <sub>C</sub>	Key	Key Length (bit)
Yes	Yes	Don't care	Key <sub>A</sub>   Key <sub>B</sub>	512
Yes	No	Don't care	Key <sub>A</sub>   0 <sup>256</sup>	512
No	Yes	Don't care	0 <sup>256</sup>   Key <sub>B</sub>	512
No	No	Yes	Key <sub>C</sub>	256
No	No	No	0 <sup>256</sup>	256

“Yes” indicates that the block exists; “No” indicates that the block does not exist; “0<sup>256</sup>” 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 [14 eFuse Controller](#).

### 20.3.3 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.

### 20.3.4 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 91.

**Table 91: 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

### 20.3.5 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  $\frac{\text{size}}{32}$ .

For definitions of *type*, *base\_addr*, *size*, please refer to Section 20.3.3.

2. Fill registers XTS\_AES\_PLAIN\_*n*\_REG (*n*: 0-15) in with plaintext (refer to Section 20.3.4). 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.

### 20.3.6 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.

### 20.3.7 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.

## 20.4 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 92: Manual Encryption Block Base Address**

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F43A000
PeriBUS2	0x6003A000

## 20.5 Register Summary

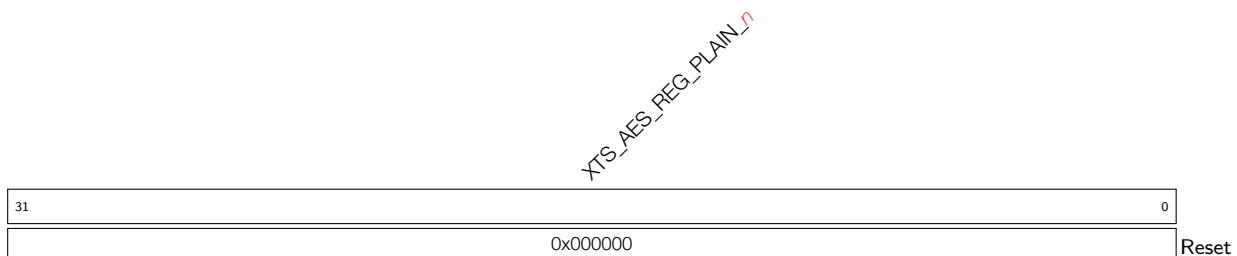
The addresses in the following table are relative to the Manual Encryption block's base addresses provided in Section 20.4.

Name	Description	Address	Access
<b>Plaintext Register Heap</b>			
<a href="#">XTS_AES_PLAIN_0_REG</a>	Plaintext register 0	0x0100	R/W
<a href="#">XTS_AES_PLAIN_1_REG</a>	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
<b>Configuration Registers</b>			
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
<b>Control/Status Registers</b>			
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
<b>Version Register</b>			
XTS_AES_DATE_REG	Version control register	0x015C	RO

## 20.6 Registers

Register 20.1: XTS\_AES\_PLAIN<sub>n</sub>\_REG ( $n$ : 0-15) (0x0100+4\* $n$ )



XTS\_AES\_REG\_PLAIN<sub>n</sub> This register stores  $n$ th 32-bit piece of plaintext. (R/W)

**Register 20.2: XTS\_AES\_LINESIZE\_REG (0x0140)**

XTS\_AES\_REG\_LINESIZE

(reserved)																																	
31																																	
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	0	0	0	0	0	0	0x0	Reset	

**XTS\_AES\_REG\_LINESIZE** Configures the data size of a single encryption. 0: 128 bits; 1: 256 bits; 2: 512 bits. (R/W)

**Register 20.3: XTS\_AES\_DESTINATION\_REG (0x0144)**

XTS\_AES\_REG\_DESTINATION

(reserved)																																	
31																																	
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	0	0	0	0	0	0	Reset		

**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 20.4: XTS\_AES\_PHYSICAL\_ADDRESS\_REG (0x0148)**

XTS\_AES\_REG\_PHYSICAL\_ADDRESS

(reserved)																																		
31	30	29																																
0	0																																	Reset

**XTS\_AES\_REG\_PHYSICAL\_ADDRESS** Physical address. (R/W)

**Register 20.5: XTS\_AES\_TRIGGER\_REG (0x014C)**

31	1	0
0 0	0	Reset

**XTS\_AES\_REG\_TRIGGER** Set to enable manual encryption. (WO)

**Register 20.6: XTS\_AES\_RELEASE\_REG (0x0150)**

31	1	0
0 0	0	Reset

**XTS\_AES\_REG\_RELEASE** Set to grant SPI1 access to encrypted result. (WO)

**Register 20.7: XTS\_AES\_DESTROY\_REG (0x0154)**

31	1	0
0 0	0	Reset

**XTS\_AES\_REG\_DESTROY** Set to destroy encrypted result. (WO)

**Register 20.8: XTS\_AES\_STATE\_REG (0x0158)**

31	(reserved)																													2	1	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	0	0	0	0	0	0	0x0	Reset	

**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 20.9: XTS\_AES\_DATE\_REG (0x015C)**

31	30	29	(reserved)		0
0	0		0x20190514		Reset

**XTS\_AES\_REG\_DATE** Version control register. (RO)

# 21. Digital Signature

## 21.1 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.

## 21.2 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

## 21.3 Functional Description

### 21.3.1 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.

### 21.3.2 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 —  $\bar{r}$  and  $M'$ . These two operands are derived from  $Y$  and  $M$ , but they are calculated in advance by software.

Operands  $Y$ ,  $M$ ,  $\bar{r}$ , 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 21.3.4.

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 18.3.1 *Large Number Modular Exponentiation* in Chapter 18 *RSA Accelerator*.

### 21.3.3 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.

### 21.3.4 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  $\bar{r}$  and  $M'$ , as described in Section 21.3.2.
- 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  $DS\_KEY = \text{HMAC-SHA256}([HMAC\_KEY]_{256}, 1^{256})$ . 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 21-1 below describes the preparations at the software level (the left part) and the DS peripheral operation at the hardware level (the right part).

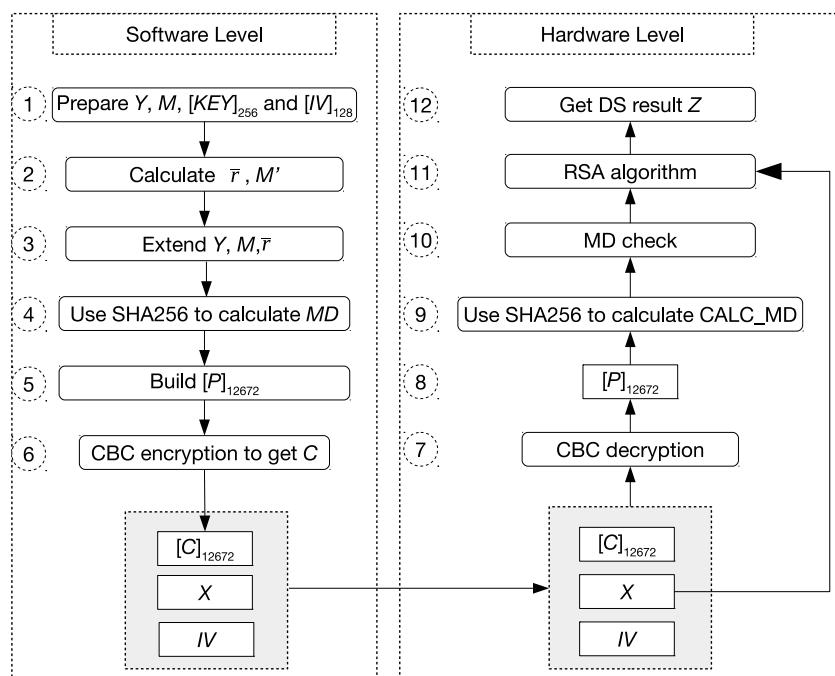


Figure 21-1. Preparations and DS Operation

Users need to follow the steps shown in the left part of Figure 21-1 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 16 *AES Accelerator*.
- **Step 2:** Calculate  $\bar{r}$  and  $M'$  based on  $M$ .
- **Step 3:** Extend  $Y$ ,  $M$ , and  $\bar{r}$ , in order to get  $[Y]_{4096}$ ,  $[M]_{4096}$ , and  $[\bar{r}]_{4096}$ , respectively. Since the largest operand length for  $Y$ ,  $M$ , and  $\bar{r}$  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:  
$$[MD]_{256} = \text{SHA256}([Y]_{4096} || [M]_{4096} || [\bar{r}]_{4096} || [M']_{32} || [L]_{32} || [IV]_{128})$$
- **Step 5:** Build  $[P]_{12672} = ([Y]_{4096} || [M]_{4096} || [\bar{r}]_{4096} || [MD]_{256} || [M']_{32} || [L]_{32} || [\beta]_{64})$ , 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  $C = [C]_{12672} = \text{AES-CBC-ENC}([P]_{12672}, [DS\_KEY]_{256}, [IV]_{128})$ , where  $C$  is the ciphertext that includes RSA operands  $Y$ ,  $M$ ,  $\bar{r}$ ,  $M'$ , 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 21.3.4.

### 21.3.5 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 21.3.4. 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  $P = \text{AES-CBC-DEC}(C, DS\_KEY, IV)$ , 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  $[Y]_{4096}$ ,  $[M]_{4096}$ ,  $[\bar{r}]_{4096}$ ,  $[M']_{32}$ ,  $[L]_{32}$ , 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 21-1, 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  $\bar{r}$  as big numbers. With  $M'$ , all operands to perform  $X^Y \bmod M$  are in place. The operand length is defined by  $L$ . The DS peripheral will get the signed result  $Z$  by calling RSA to perform  $Z = X^Y \bmod M$ .

#### 21.3.6 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 21.3.5.

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 16 [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 21.3.2) 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  $Z_i$  ( $i \in \{0, 1, 2, \dots, n\}$ ) 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.

## 21.4 Base Address

Users can access the DS peripheral with two base addresses, which can be seen in Table 94. For more information about accessing peripherals from different buses please see Chapter 1 [System and Memory](#).

**Table 94: Base Address**

Bus to Access Peripheral	Base Address
PeriBUS1	0x3F43D000
PeriBUS2	0x6003D000

## 21.5 Memory Blocks

Both the starting address and ending address in the following table are relative to the DS peripheral base addresses provided in Section 21.4.

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

## 21.6 Register Summary

The addresses in the following table are relative to the DS peripheral base addresses provided in Section 21.4.

Name	Description	Address	Access
<b>Configuration Registers</b>			
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
<b>Status/Control Registers</b>			
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 <i>DS_KEY</i> is not ready	0x0E10	RO
DS_QUERY_CHECK_REG	Queries DS check result	0x0814	RO
<b>Version Register</b>			
DS_DATE_REG	Version control register	0x0820	W/R

## 21.7 Registers

Register 21.1: DS\_IV\_*m*\_REG (*m*: 0-3) (0x0630+4\**m*)

31	0
0x0000000000	Reset

**DS\_IV\_***m***\_REG** (*m*: 0-3) IV block data. (WO)

Register 21.2: DS\_SET\_START\_REG (0x0E00)

31	0
0 0	DS_SET_START

**DS\_SET\_START** Write 1 to this register to activate the DS peripheral. (WO)

Register 21.3: DS\_SET\_ME\_REG (0x0E04)

31	0
0 0	DS_SET_ME

**DS\_SET\_ME** Write 1 to this register to start DS operation. (WO)

**Register 21.4: DS\_SET\_FINISH\_REG (0x0E08)**

31	0
0 0	0   0 Reset

**DS\_SET\_FINISH** Write 1 to this register to end DS operation. (WO)

**Register 21.5: DS\_QUERY\_BUSY\_REG (0x0E0C)**

31	0
0 0	0   0 Reset

**DS\_QUERY\_BUSY** 1: The DS peripheral is busy; 0: The DS peripheral is idle. (RO)

**Register 21.6: DS\_QUERY\_KEY\_WRONG\_REG (0x0E10)**

31	0
0 0	4   3 0x0 Reset

**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 21.7: DS\_QUERY\_CHECK\_REG (0x0E14)**

																													DS_PADDING_BAD			DS_MD_ERROR		
(reserved)																													2	1	0			
31	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	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	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**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 21.8: DS\_DATE\_REG (0x0E20)**

																														DS_DATE				
(reserved)																												29	0					
31	30	29	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	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	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Reset

**DS\_DATE** Version control register. (R/W)

## Revision History

Date	Version	Release notes
2020.01	V0.2	<p>Added the following chapters:</p> <ul style="list-style-type: none"><li>• Chapter 5 <i>IO MUX and GPIO Matrix</i></li><li>• Chapter 7 <i>DMA Controller</i></li><li>• Chapter 8 <i>UART Controller</i></li></ul> <p>Updated the configurations of eFuse-programming and eFuse-read timing parameters in Chapter 14 <i>eFuse Controller</i>.</p>
2019.11	V0.1	Preliminary release.