# ESP32-C3

## Technical Reference Manual Version 1.3

ESPRESSIF

www.espressif.com

## About This Document

The **ESP32-C3 Technical Reference Manual** is targeted at developers working on low level software projects that use the ESP32-C3 SoC. It describes the hardware modules listed below for the ESP32-C3 SoC and other products in ESP32-C3 series. The modules detailed in this document provide an overview, list of features, hardware architecture details, any necessary programming procedures, as well as register descriptions.

## Navigation in This Document

Here are some tips on navigation through this extensive document:

- Release Status at a Glance on the very next page is a minimal list of all chapters from where you can directly jump to a specific chapter.
- Use the **Bookmarks** on the side bar to jump to any specific chapters or sections from anywhere in the document. Note this PDF document is configured to automatically display **Bookmarks** when open, which is necessary for an extensive document like this one. However, some PDF viewers or browsers ignore this setting, so if you don't see the **Bookmarks** by default, try one or more of the following methods:
  - Install a PDF Reader Extension for your browser;
  - Download this document, and view it with your local PDF viewer;
  - Set your PDF viewer to always automatically display the **Bookmarks** on the left side bar when open.
- Use the native **Navigation** function of your PDF viewer to navigate through the documents. Most PDF viewers support to go **Up**, **Down**, **Previous**, **Next**, **Back**, **Forward** and **Page** with buttons, menu, or hot keys.
- You can also use the built-in **GoBack** button on the upper right corner on each and every page to go back to the previous place before you click a link within the document. Note this feature may only work with some Acrobat-specific PDF viewers (for example, Acrobat Reader and Adobe DC) and browsers with built-in Acrobat-specific PDF viewers or extensions (for example, Firefox).

# Release Status at a Glance

| No. | Chapter | Progress |
|---|---|---|
| | **Part I. Microprocessor and Master** | |
| 1 | ESP-RISC-V CPU | Published |
| 2 | GDMA Controller (GDMA) | Published |
| | **Part II. Memory Organization** | |
| 3 | System and Memory | Published |
| 4 | eFuse Controller (EFUSE) | Published |
| | **Part III. System Component** | |
| 5 | IO MUX and GPIO Matrix (GPIO, IO MUX) | Published |
| 6 | Reset and Clock | Published |
| 7 | Chip Boot Control | Published |
| 8 | Interrupt Matrix (INTERRUPT) | Published |
| 9 | Low-power Management | Published |
| 10 | System Timer (SYSTIMER) | Published |
| 11 | Timer Group (TIMG) | Published |
| 12 | Watchdog Timers (WDT) | Published |
| 13 | XTAL32K Watchdog Timers (XTWDT) | Published |
| 14 | Permission Control (PMS) | Published |
| 15 | World Controller (WCL) | Published |
| 16 | System Registers (SYSREG) | Published |
| 17 | Debug Assistant (ASSIST_DEBUG) | Published |
| | **Part V. Cryptography/Security Component** | |
| 18 | SHA Accelerator (SHA) | Published |
| 19 | AES Accelerator (AES) | Published |
| 20 | RSA Accelerator (RSA) | Published |
| 21 | HMAC Accelerator (HMAC) | Published |
| 22 | Digital Signature (DS) | Published |
| 23 | External Memory Encryption and Decryption (XTS_AES) | Published |
| 24 | Clock Glitch Detection | Published |
| 25 | Random Number Generator (RNG) | Published |
| | **Part VII. Connectivity Interface** | |
| 26 | UART Controller (UART) | Published |
| 27 | SPI Controller (SPI) | Published |
| 28 | I2C Controller (I2C) | Published |
| 29 | I2S Controller (I2S) | Published |
| 30 | USB Serial/JTAG Controller (USB_SERIAL_JTAG) | Published |
| 31 | Two-wire Automotive Interface (TWAI) | Published |
| 32 | LED PWM Controller (LEDC) | Published |
| 33 | Remote Control Peripheral (RMT) | Published |
| | **Part VIII. Analog Signal Processing** | |
| 34 | On-Chip Sensor and Analog Signal Processing | Published |

**Note:**

Check the link or the QR code to make sure that you use the latest version of this document:
https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf

# Contents


I Microprocessor and Master 30

1 ESP-RISC-V CPU 31
1.1 Overview 31
1.2 Features 31
1.3 Address Map 32
1.4 Configuration and Status Registers (CSRs) 32
1.4.1 Register Summary 32
1.4.2 Register Description 34
1.5 Interrupt Controller 42
1.5.1 Features 42
1.5.2 Functional Description 42
1.5.3 Suggested Operation 44
1.5.3.1 Latency Aspects 44
1.5.3.2 Configuration Procedure 45
1.5.4 Register Summary 46
1.5.5 Register Description 46
1.6 Debug 47
1.6.1 Overview 47
1.6.2 Features 48
1.6.3 Functional Description 48
1.6.4 Register Summary 48
1.6.5 Register Description 48
1.7 Hardware Trigger 51
1.7.1 Features 51
1.7.2 Functional Description 51
1.7.3 Trigger Execution Flow 52
1.7.4 Register Summary 52
1.7.5 Register Description 53
1.8 Memory Protection 57
1.8.1 Overview 57
1.8.2 Features 57
1.8.3 Functional Description 57
1.8.4 Register Summary 59
1.8.5 Register Description 59

2 GDMA Controller (GDMA) 60
2.1 Overview 60
2.2 Features 60
2.3 Architecture 61
2.4 Functional Description 61
2.4.1 Linked List 62

2.4.2 Peripheral-to-Memory and Memory-to-Peripheral Data Transfer 63
2.4.3 Memory-to-Memory Data Transfer 63
2.4.4 Enabling GDMA 64
2.4.5 Linked List Reading Process 64
2.4.6 EOF 65
2.4.7 Accessing Internal RAM 65
2.4.8 Arbitration 66
2.5 GDMA Interrupts 66
2.6 Programming Procedures 67
2.6.1 Programming Procedure for GDMA Clock and Reset 67
2.6.2 Programming Procedures for GDMA's Transmit Channel 67
2.6.3 Programming Procedures for GDMA's Receive Channel 67
2.6.4 Programming Procedures for Memory-to-Memory Transfer 68
2.7 Register Summary 69
2.8 Registers 73

II Memory Organization 90

3 System and Memory 91
3.1 Overview 91
3.2 Features 91
3.3 Functional Description 92
3.3.1 Address Mapping 92
3.3.2 Internal Memory 93
3.3.3 External Memory 94
3.3.3.1 External Memory Address Mapping 95
3.3.3.2 Cache 95
3.3.3.3 Cache Operations 96
3.3.4 GDMA Address Space 96
3.3.5 Modules/Peripherals 97
3.3.5.1 Module/Peripheral Address Mapping 98

4 eFuse Controller (EFUSE) 100
4.1 Overview 100
4.2 Features 100
4.3 Functional Description 100
4.3.1 Structure 100
4.3.1.1 EFUSE_WR_DIS 106
4.3.1.2 EFUSE_RD_DIS 106
4.3.1.3 Data Storage 106
4.3.2 Programming of Parameters 107
4.3.3 User Read of Parameters 109
4.3.4 eFuse VDDQ Timing 111
4.3.5 The Use of Parameters by Hardware Modules 111
4.3.6 Interrupts 111
4.4 Register Summary 112

4.5 Registers 116

III System Component 158

5 IO MUX and GPIO Matrix (GPIO, IO MUX) 159
5.1 Overview 159
5.2 Features 159
5.3 Architectural Overview 159
5.4 Peripheral Input via GPIO Matrix 161
5.4.1 Overview 161
5.4.2 Signal Synchronization 162
5.4.3 Functional Description 162
5.4.4 Simple GPIO Input 163
5.5 Peripheral Output via GPIO Matrix 163
5.5.1 Overview 163
5.5.2 Functional Description 164
5.5.3 Simple GPIO Output 165
5.5.4 Sigma Delta Modulated Output (SDM) 165
5.5.4.1 Functional Description 165
5.5.4.2 SDM Configuration 166
5.6 Direct Input and Output via IO MUX 166
5.6.1 Overview 166
5.6.2 Functional Description 166
5.7 Analog Functions of GPIO Pins 166
5.8 Pin Functions in Light-sleep 167
5.9 Pin Hold Feature 167
5.10 Power Supplies and Management of GPIO Pins 167
5.10.1 Power Supplies of GPIO Pins 167
5.10.2 Power Supply Management 168
5.11 Peripheral Signal List 168
5.12 IO MUX Functions List 174
5.13 Analog Functions List 175
5.14 Register Summary 176
5.14.1 GPIO Matrix Register Summary 176
5.14.2 IO MUX Register Summary 178
5.14.3 SDM Register Summary 178
5.15 Registers 179
5.15.1 GPIO Matrix Registers 179
5.15.2 IO MUX Registers 186
5.15.3 SDM Output Registers 188

6 Reset and Clock 191
6.1 Reset 191
6.1.1 Overview 191
6.1.2 Architectural Overview 191
6.1.3 Features 191

6.1.4 Functional Description 192
6.2 Clock 193
6.2.1 Overview 193
6.2.2 Architectural Overview 194
6.2.3 Features 194
6.2.4 Functional Description 194
6.2.4.1 CPU Clock 195
6.2.4.2 Peripheral Clock 195
6.2.4.3 Wi-Fi and Bluetooth LE Clock 197
6.2.4.4 RTC Clock 197

7 Chip Boot Control 198
7.1 Overview 198
7.2 Boot Mode Control 199
7.3 ROM Messages Printing Control 200

8 Interrupt Matrix (INTERRUPT) 202
8.1 Overview 202
8.2 Features 202
8.3 Functional Description 203
8.3.1 Peripheral Interrupt Sources 203
8.3.2 CPU Interrupts 207
8.3.3 Allocate Peripheral Interrupt Source to CPU Interrupt 207
8.3.3.1 Allocate one peripheral interrupt source (Source_*X*) to CPU 207
8.3.3.2 Allocate multiple peripheral interrupt sources (Source_X*n*) to CPU 207
8.3.3.3 Disable CPU peripheral interrupt source (Source_*X*) 207
8.3.4 Query Current Interrupt Status of Peripheral Interrupt Source 207
8.4 Register Summary 209
8.5 Registers 213

9 Low-power Management 219
9.1 Introduction 219
9.2 Features 219
9.3 Functional Description 219
9.3.1 Power Management Unit (PMU) 221
9.3.2 Low-Power Clocks 222
9.3.3 Timers 223
9.3.4 Voltage Regulators 224
9.3.4.1 Digital System Voltage Regulator 224
9.3.4.2 Low-power Voltage Regulator 225
9.3.4.3 Brownout Detector 225
9.4 Power Modes Management 226
9.4.1 Power Domain 226
9.4.2 Pre-defined Power Modes 227
9.4.3 Wakeup Source 227
9.4.4 Reject Sleep 228

9.5 Retention DMA 228
9.6 RTC Boot 229
9.7 Register Summary 231
9.8 Registers 233

10 System Timer (SYSTIMER) 270
10.1 Overview 270
10.2 Features 270
10.3 Clock Source Selection 271
10.4 Functional Description 271
10.4.1 Counter 271
10.4.2 Comparator and Alarm 272
10.4.3 Synchronization Operation 273
10.4.4 Interrupt 274
10.5 Programming Procedure 274
10.5.1 Read Current Count Value 274
10.5.2 Configure One-Time Alarm in Target Mode 274
10.5.3 Configure Periodic Alarms in Period Mode 274
10.5.4 Update After Light-sleep 275
10.6 Register Summary 275
10.7 Registers 277

11 Timer Group (TIMG) 288
11.1 Overview 288
11.2 Functional Description 289
11.2.1 16-bit Prescaler and Clock Selection 289
11.2.2 54-bit Time-base Counter 289
11.2.3 Alarm Generation 290
11.2.4 Timer Reload 291
11.2.5 RTC_SLOW_CLK Frequency Calculation 291
11.2.6 Interrupts 291
11.3 Configuration and Usage 292
11.3.1 Timer as a Simple Clock 292
11.3.2 Timer as One-shot Alarm 292
11.3.3 Timer as Periodic Alarm 293
11.3.4 RTC_SLOW_CLK Frequency Calculation 293
11.4 Register Summary 294
11.5 Registers 295

12 Watchdog Timers (WDT) 305
12.1 Overview 305
12.2 Digital Watchdog Timers 306
12.2.1 Features 306
12.2.2 Functional Description 307
12.2.2.1 Clock Source and 32-Bit Counter 307
12.2.2.2 Stages and Timeout Actions 308

12.2.2.3 Write Protection 308
12.2.2.4 Flash Boot Protection 309
12.3 Super Watchdog 309
12.3.1 Features 309
12.3.2 Super Watchdog Controller 309
12.3.2.1 Structure 310
12.3.2.2 Workflow 310
12.4 Interrupts 310
12.5 Registers 311

13 XTAL32K Watchdog Timers (XTWDT) 312
13.1 Overview 312
13.2 Features 312
13.2.1 Interrupt and Wake-Up 312
13.2.2 BACKUP32K_CLK 313
13.3 Functional Description 313
13.3.1 Workflow 313
13.3.2 BACKUP32K_CLK Working Principle 313
13.3.3 Configuring the Divisor Component of BACKUP32K_CLK 313

14 Permission Control (PMS) 315
14.1 Overview 315
14.2 Features 316
14.3 Privileged Environment and Unprivileged Environment 316
14.4 Internal Memory 317
14.4.1 ROM 317
14.4.2 SRAM 318
14.4.2.1 Internal SRAM0 Access Configuration 318
14.4.2.2 Internal SRAM1 Access Configuration 319
14.4.3 RTC FAST Memory 322
14.5 Peripherals 323
14.5.1 Access Configuration 323
14.5.2 Split Peripheral Regions into Split Regions 324
14.6 External Memory 325
14.6.1 SPI and Cache's Access to External Flash 325
14.6.1.1 Address 326
14.6.1.2 Access Configuration 326
14.6.2 CPU's Access to Cache 327
14.6.2.1 Split Regions 327
14.6.3 Access Configuration 327
14.7 Unauthorized Access and Interrupts 328
14.7.1 Interrupt upon Unauthorized IBUS Access 329
14.7.2 Interrupt upon Unauthorized DBUS Access 329
14.7.3 Interrupt upon Unauthorized Access to External Memory 330
14.7.4 Interrupt upon Unauthorized Access to Internal Memory via GDMA 330
14.7.5 Interrupt upon Unauthorized peripheral bus (PIF) Access 330

14.7.6 Interrupt upon Unauthorized PIF Access Alignment 331
14.8 Register Locks 332
14.9 Register Summary 335
14.10 Registers 338

15 World Controller (WCL) 413
15.1 Introduction 413
15.2 Features 413
15.3 Functional Description 413
15.4 CPU's World Switch 415
15.4.1 From Secure World to Non-secure World 415
15.4.2 From Non-secure World to Secure World 416
15.5 World Switch Log 417
15.5.1 Structure of World Switch Log Register 417
15.5.2 How World Switch Log Registers are Updated 418
15.5.3 How to Read World Switch Log Registers 420
15.5.4 Nested Interrupts 420
15.5.4.1 Programming Procedure 420
15.6 Register Summary 422
15.7 Registers 423

16 System Registers (SYSREG) 427
16.1 Overview 427
16.2 Features 427
16.3 Function Description 427
16.3.1 System and Memory Registers 427
16.3.1.1 Internal Memory 427
16.3.1.2 External Memory 428
16.3.1.3 RSA Memory 428
16.3.2 Clock Registers 429
16.3.3 Interrupt Signal Registers 429
16.3.4 Low-power Management Registers 429
16.3.5 Peripheral Clock Gating and Reset Registers 429
16.4 Register Summary 432
16.5 Registers 434

17 Debug Assistant (ASSIST_DEBUG) 446
17.1 Overview 446
17.2 Features 446
17.3 Functional Description 446
17.3.1 Region Read/Write Monitoring 446
17.3.2 SP Monitoring 446
17.3.3 PC Logging 447
17.3.4 CPU/DMA Bus Access Logging 447
17.4 Recommended Operation 447
17.4.1 Region Monitoring and SP Monitoring Configuration Process 447

17.4.2 PC Logging Configuration Process 448
17.4.3 CPU/DMA Bus Access Logging Configuration Process 449
17.5 Register Summary 453
17.6 Registers 455

IV Cryptography/Security Component 472

18 AES Accelerator (AES) 473
18.1 Introduction 473
18.2 Features 473
18.3 AES Working Modes 473
18.4 Typical AES Working Mode 475
18.4.1 Key, Plaintext, and Ciphertext 475
18.4.2 Endianness 475
18.4.3 Operation Process 477
18.5 DMA-AES Working Mode 477
18.5.1 Key, Plaintext, and Ciphertext 478
18.5.2 Endianness 479
18.5.3 Standard Incrementing Function 479
18.5.4 Block Number 479
18.5.5 Initialization Vector 479
18.5.6 Block Operation Process 480
18.6 Memory Summary 480
18.7 Register Summary 481
18.8 Registers 482

19 HMAC Accelerator (HMAC) 486
19.1 Main Features 486
19.2 Functional Description 486
19.2.1 Upstream Mode 486
19.2.2 Downstream JTAG Enable Mode 487
19.2.3 Downstream Digital Signature Mode 487
19.2.4 HMAC eFuse Configuration 488
19.2.5 HMAC Process (Detailed) 489
19.3 HMAC Algorithm Details 490
19.3.1 Padding Bits 491
19.3.2 HMAC Algorithm Structure 491
19.4 Register Summary 494
19.5 Registers 496

20 RSA Accelerator (RSA) 502
20.1 Introduction 502
20.2 Features 502
20.3 Functional Description 502
20.3.1 Large Number Modular Exponentiation 503
20.3.2 Large Number Modular Multiplication 504

20.3.3 Large Number Multiplication 505
20.3.4 Options for Acceleration 505
20.4 Memory Summary 507
20.5 Register Summary 508
20.6 Registers 509

21 SHA Accelerator (SHA) 513
21.1 Introduction 513
21.2 Features 513
21.3 Working Modes 513
21.4 Function Description 514
21.4.1 Preprocessing 514
21.4.1.1 Padding the Message 514
21.4.1.2 Parsing the Message 514
21.4.1.3 Setting the Initial Hash Value 515
21.4.2 Hash Operation 515
21.4.2.1 Typical SHA Mode Process 515
21.4.2.2 DMA-SHA Mode Process 517
21.4.3 Message Digest 517
21.4.4 Interrupt 518
21.5 Register Summary 518
21.6 Registers 519

22 Digital Signature (DS) 523
22.1 Overview 523
22.2 Features 523
22.3 Functional Description 523
22.3.1 Overview 523
22.3.2 Private Key Operands 524
22.3.3 Software Prerequisites 524
22.3.4 DS Operation at the Hardware Level 526
22.3.5 DS Operation at the Software Level 526
22.4 Memory Summary 529
22.5 Register Summary 530
22.6 Registers 531

23 External Memory Encryption and Decryption (XTS_AES) 534
23.1 Overview 534
23.2 Features 534
23.3 Module Structure 534
23.4 Functional Description 535
23.4.1 XTS Algorithm 535
23.4.2 Key 536
23.4.3 Target Memory Space 536
23.4.4 Data Writing 537
23.4.5 Manual Encryption Block 537

23.4.6 Auto Decryption Block 538
23.5 Software Process 538
23.6 Register Summary 540
23.7 Registers 541

24 Random Number Generator (RNG) 544
24.1 Introduction 544
24.2 Features 544
24.3 Functional Description 544
24.4 Programming Procedure 545
24.5 Register Summary 545
24.6 Register 546

25 Clock Glitch Detection 547
25.1 Overview 547
25.2 Functional Description 547
25.2.1 Clock Glitch Detection 547
25.2.2 Reset 547

V Connectivity Interface 548

26 UART Controller (UART) 549
26.1 Overview 549
26.2 Features 549
26.3 UART Structure 550
26.4 Functional Description 551
26.4.1 Clock and Reset 551
26.4.2 UART RAM 552
26.4.3 Baud Rate Generation and Detection 553
26.4.3.1 Baud Rate Generation 553
26.4.3.2 Baud Rate Detection 554
26.4.4 UART Data Frame 555
26.4.5 AT_CMD Character Structure 556
26.4.6 RS485 556
26.4.6.1 Driver Control 556
26.4.6.2 Turnaround Delay 557
26.4.6.3 Bus Snooping 557
26.4.7 IrDA 557
26.4.8 Wake-up 558
26.4.9 Flow Control 559
26.4.9.1 Hardware Flow Control 559
26.4.9.2 Software Flow Control 560
26.4.10 GDMA Mode 561
26.4.11 UART Interrupts 561
26.4.12 UHCI Interrupts 562
26.5 Programming Procedures 563

26.5.1 Register Type 563
26.5.1.1 Synchronous Registers 563
26.5.1.2 Static Registers 564
26.5.1.3 Immediate Registers 565
26.5.2 Detailed Steps 565
26.5.2.1 Initializing UART*n* 566
26.5.2.2 Configuring UART*n* Communication 566
26.5.2.3 Enabling UART*n* 566
26.6 Register Summary 568
26.7 Registers 571

27 SPI Controller (SPI) 608
27.1 Overview 608
27.2 Glossary 608
27.3 Features 609
27.4 Architectural Overview 610
27.5 Functional Description 610
27.5.1 Data Modes 611
27.5.2 FSPI Bus Signal Mapping 611
27.5.3 Bit Read/Write Order Control 614
27.5.4 Transfer Modes 614
27.5.5 CPU-Controlled Data Transfer 614
27.5.5.1 CPU-Controlled Master Mode 615
27.5.5.2 CPU-Controlled Slave Mode 616
27.5.6 DMA-Controlled Data Transfer 616
27.5.6.1 GDMA Configuration 617
27.5.6.2 GDMA TX/RX Buffer Length Control 618
27.5.7 Data Flow Control in GP-SPI2 Master and Slave Modes 618
27.5.7.1 GP-SPI2 Functional Blocks 618
27.5.7.2 Data Flow Control in Master Mode 619
27.5.7.3 Data Flow Control in Slave Mode 620
27.5.8 GP-SPI2 Works as a Master 621
27.5.8.1 State Machine 621
27.5.8.2 Register Configuration for State and Bit Mode Control 623
27.5.8.3 Full-Duplex Communication (1-bit Mode Only) 626
27.5.8.4 Half-Duplex Communication (1/2/4-bit Mode) 627
27.5.8.5 DMA-Controlled Configurable Segmented Transfer 629
27.5.9 GP-SPI2 Works as a Slave 633
27.5.9.1 Communication Formats 633
27.5.9.2 Supported CMD Values in Half-Duplex Communication 634
27.5.9.3 Slave Single Transfer and Slave Segmented Transfer 637
27.5.9.4 Configuration of Slave Single Transfer 637
27.5.9.5 Configuration of Slave Segmented Transfer in Half-Duplex 638
27.5.9.6 Configuration of Slave Segmented Transfer in Full-Duplex 638
27.6 CS Setup Time and Hold Time Control 639
27.7 GP-SPI2 Clock Control 640

27.7.1 Clock Phase and Polarity 640
27.7.2 Clock Control in Master Mode 642
27.7.3 Clock Control in Slave Mode 642
27.8 GP-SPI2 Timing Compensation 642
27.9 Interrupts 644
27.10 Register Summary 647
27.11 Registers 648

28 I2C Controller (I2C) 676
28.1 Overview 676
28.2 Features 676
28.3 I2C Architecture 677
28.4 Functional Description 679
28.4.1 Clock Configuration 679
28.4.2 SCL and SDA Noise Filtering 679
28.4.3 SCL Clock Stretching 680
28.4.4 Generating SCL Pulses in Idle State 680
28.4.5 Synchronization 680
28.4.6 Open-Drain Output 681
28.4.7 Timing Parameter Configuration 682
28.4.8 Timeout Control 683
28.4.9 Command Configuration 684
28.4.10 TX/RX RAM Data Storage 685
28.4.11 Data Conversion 686
28.4.12 Addressing Mode 686
28.4.13 $R/\overline{W}$ Bit Check in 10-bit Addressing Mode 687
28.4.14 To Start the I2C Controller 687
28.5 Programming Example 687
28.5.1 $I2C_{master}$ Writes to $I2C_{slave}$ with a 7-bit Address in One Command Sequence 687
28.5.1.1 Introduction 688
28.5.1.2 Configuration Example 688
28.5.2 $I2C_{master}$ Writes to $I2C_{slave}$ with a 10-bit Address in One Command Sequence 689
28.5.2.1 Introduction 690
28.5.2.2 Configuration Example 690
28.5.3 $I2C_{master}$ Writes to $I2C_{slave}$ with Two 7-bit Addresses in One Command Sequence 692
28.5.3.1 Introduction 692
28.5.3.2 Configuration Example 692
28.5.4 $I2C_{master}$ Writes to $I2C_{slave}$ with a 7-bit Address in Multiple Command Sequences 694
28.5.4.1 Introduction 694
28.5.4.2 Configuration Example 695
28.5.5 $I2C_{master}$ Reads $I2C_{slave}$ with a 7-bit Address in One Command Sequence 696
28.5.5.1 Introduction 696
28.5.5.2 Configuration Example 697
28.5.6 $I2C_{master}$ Reads $I2C_{slave}$ with a 10-bit Address in One Command Sequence 698
28.5.6.1 Introduction 698
28.5.6.2 Configuration Example 699

28.5.7 $I2C_{master}$ Reads $I2C_{slave}$ with Two 7-bit Addresses in One Command Sequence 700
28.5.7.1 Introduction 701
28.5.7.2 Configuration Example 701
28.5.8 $I2C_{master}$ Reads $I2C_{slave}$ with a 7-bit Address in Multiple Command Sequences 704
28.5.8.1 Introduction 704
28.5.8.2 Configuration Example 705
28.6 Interrupts 707
28.7 Register Summary 708
28.8 Registers 710

29 I2S Controller (I2S) 730
29.1 Overview 730
29.2 Terminology 730
29.3 Features 731
29.4 System Architecture 732
29.5 Supported Audio Standards 733
29.5.1 TDM Philips Standard 734
29.5.2 TDM MSB Alignment Standard 734
29.5.3 TDM PCM Standard 735
29.5.4 PDM Standard 735
29.6 I2S TX/RX Clock 736
29.7 I2S Reset 738
29.8 I2S Master/Slave Mode 738
29.8.1 Master/Slave TX Mode 738
29.8.2 Master/Slave RX Mode 739
29.9 Transmitting Data 739
29.9.1 Data Format Control 739
29.9.1.1 Bit Width Control of Channel Valid Data 739
29.9.1.2 Endian Control of Channel Valid Data 740
29.9.1.3 A-law/$\mu$-law Compression and Decompression 740
29.9.1.4 Bit Width Control of Channel TX Data 741
29.9.1.5 Bit Order Control of Channel Data 741
29.9.2 Channel Mode Control 742
29.9.2.1 I2S Channel Control in TDM TX Mode 742
29.9.2.2 I2S Channel Control in PDM TX Mode 743
29.10 Receiving Data 745
29.10.1 Channel Mode Control 745
29.10.1.1 I2S Channel Control in TDM RX Mode 746
29.10.1.2 I2S Channel Control in PDM RX Mode 746
29.10.2 Data Format Control 746
29.10.2.1 Bit Order Control of Channel Data 746
29.10.2.2 Bit Width Control of Channel Storage (Valid) Data 747
29.10.2.3 Bit Width Control of Channel RX Data 747
29.10.2.4 Endian Control of Channel Storage Data 747
29.10.2.5 A-law/$\mu$-law Compression and Decompression 748
29.11 Software Configuration Process 748

29.11.1 Configure I2S as TX Mode 748
29.11.2 Configure I2S as RX Mode 749
29.12 I2S Interrupts 749
29.13 Register Summary 749
29.14 Registers 751

30 USB Serial/JTAG Controller (USB_SERIAL_JTAG) 764
30.1 Overview 764
30.2 Features 764
30.3 Functional Description 765
30.3.1 CDC-ACM USB Interface Functional Description 765
30.3.2 CDC-ACM Firmware Interface Functional Description 767
30.3.3 USB-to-JTAG Interface 767
30.3.4 JTAG Command Processor 767
30.3.5 USB-to-JTAG Interface: CMD_REP usage example 768
30.3.6 USB-to-JTAG Interface: Response Capture Unit 769
30.3.7 USB-to-JTAG Interface: Control Transfer Requests 770
30.4 Recommended Operation 770
30.5 Register Summary 773
30.6 Registers 774

31 Two-wire Automotive Interface (TWAI) 788
31.1 Features 788
31.2 Functional Protocol 789
31.2.1 TWAI Properties 789
31.2.2 TWAI Messages 790
31.2.2.1 Data Frames and Remote Frames 790
31.2.2.2 Error and Overload Frames 792
31.2.2.3 Interframe Space 794
31.2.3 TWAI Errors 794
31.2.3.1 Error Types 794
31.2.3.2 Error States 795
31.2.3.3 Error Counters 795
31.2.4 TWAI Bit Timing 796
31.2.4.1 Nominal Bit 796
31.2.4.2 Hard Synchronization and Resynchronization 797
31.3 Architectural Overview 798
31.3.1 Registers Block 798
31.3.2 Bit Stream Processor 799
31.3.3 Error Management Logic 799
31.3.4 Bit Timing Logic 800
31.3.5 Acceptance Filter 800
31.3.6 Receive FIFO 800
31.4 Functional Description 800
31.4.1 Modes 800
31.4.1.1 Reset Mode 800

31.4.1.2 Operation Mode 800
31.4.2 Bit Timing 801
31.4.3 Interrupt Management 802
31.4.3.1 Receive Interrupt (RXI) 802
31.4.3.2 Transmit Interrupt (TXI) 802
31.4.3.3 Error Warning Interrupt (EWI) 802
31.4.3.4 Data Overrun Interrupt (DOI) 803
31.4.3.5 Error Passive Interrupt (TXI) 803
31.4.3.6 Arbitration Lost Interrupt (ALI) 803
31.4.3.7 Bus Error Interrupt (BEI) 803
31.4.3.8 Bus Status Interrupt (BSI) 804
31.4.4 Transmit and Receive Buffers 804
31.4.4.1 Overview of Buffers 804
31.4.4.2 Frame Information 805
31.4.4.3 Frame Identifier 805
31.4.4.4 Frame Data 806
31.4.5 Receive FIFO and Data Overruns 806
31.4.6 Acceptance Filter 807
31.4.6.1 Single Filter Mode 808
31.4.6.2 Dual Filter Mode 808
31.4.7 Error Management 809
31.4.7.1 Error Warning Limit 810
31.4.7.2 Error Passive 810
31.4.7.3 Bus-Off and Bus-Off Recovery 810
31.4.8 Error Code Capture 811
31.4.9 Arbitration Lost Capture 812
31.5 Register Summary 813
31.6 Registers 814

32 LED PWM Controller (LEDC) 827
32.1 Overview 827
32.2 Features 827
32.3 Functional Description 828
32.3.1 Architecture 828
32.3.2 Timers 828
32.3.2.1 Clock Source 828
32.3.2.2 Clock Divider Configuration 829
32.3.2.3 14-bit Counter 830
32.3.3 PWM Generators 831
32.3.4 Duty Cycle Fading 832
32.3.5 Interrupts 833
32.4 Register Summary 834
32.5 Registers 836

33 Remote Control Peripheral (RMT) 843
33.1 Overview 843

33.2 Features 843
33.3 Functional Description 843
33.3.1 RMT Architecture 844
33.3.2 RMT RAM 844
33.3.3 Clock 845
33.3.4 Transmitter 846
33.3.4.1 Normal TX Mode 846
33.3.4.2 Wrap TX Mode 846
33.3.4.3 TX Modulation 846
33.3.4.4 Continuous TX Mode 847
33.3.4.5 Simultaneous TX Mode 847
33.3.5 Receiver 847
33.3.5.1 Normal RX Mode 848
33.3.5.2 Wrap RX Mode 848
33.3.5.3 RX Filtering 848
33.3.5.4 RX Demodulation 849
33.3.6 Configuration Update 849
33.3.7 Interrupts 850
33.4 Register Summary 851
33.5 Registers 852

VI Analog Signal Processing 867

34 On-Chip Sensor and Analog Signal Processing 868
34.1 Overview 868
34.2 SAR ADCs 868
34.2.1 Overview 868
34.2.2 Features 868
34.2.3 Functional Description 869
34.2.3.1 Input Signals 870
34.2.3.2 ADC Conversion and Attenuation 870
34.2.3.3 DIG ADC Controller 870
34.2.3.4 DIG ADC Clock 871
34.2.3.5 DMA Support 872
34.2.3.6 DIG ADC FSM 872
34.2.3.7 ADC Filters 875
34.2.3.8 Threshold Monitoring 875
34.2.3.9 SAR ADC2 Arbiter 875
34.3 Temperature Sensor 876
34.3.1 Overview 876
34.3.2 Features 876
34.3.3 Functional Description 876
34.4 Interrupts 877
34.5 Register Summary 877
34.6 Register 878

VII Appendix 890

Related Documentation and Resources 891

Glossary 892
Abbreviations for Peripherals 892
Abbreviations Related to Registers 892
Access Types for Registers 892

Programming Reserved Register Field 895
Introduction 895
Programming Reserved Register Field 895

Interrupt Configuration Registers 896

Revision History 897

# List of Tables


1.3-1 CPU Address Map 32
1.5-1 ID wise map of Interrupt Trap-Vector Addresses 43
1.7-1 NAPOT encoding for maddress 52

2.4-1 Selecting Peripherals via Register Configuration 63
2.4-2 Descriptor Field Alignment Requirements 65

3.3-1 Internal Memory Address Mapping 93
3.3-2 External Memory Address Mapping 95
3.3-3 Module/Peripheral Address Mapping 98

4.3-1 Parameters in eFuse BLOCK0 101
4.3-2 Secure Key Purpose Values 104
4.3-3 Parameters in BLOCK1 to BLOCK10 105
4.3-4 Registers Information 109
4.3-5 Configuration of Default VDDQ Timing Parameters 111

5.8-1 Bits Used to Control IO MUX Functions in Light-sleep Mode 167
5.11-1 Peripheral Signals via GPIO Matrix 169
5.12-1 IO MUX Pin Functions 174
5.12-2 Power-Up Glitches on Pins 175
5.13-1 Analog Functions of IO MUX Pins 175

6.1-1 Reset Sources 193
6.2-1 CPU Clock Source 195
6.2-2 CPU Clock Frequency 195
6.2-3 Peripheral Clocks 196
6.2-4 APB_CLK Clock Frequency 197
6.2-5 CRYPTO_CLK Frequency 197

7.1-1 Default Configuration of Strapping Pins 198
7.2-1 Boot Mode Control 199
7.3-1 ROM Message Printing Control 200

8.3-1 CPU Peripheral Interrupt Configuration/Status Registers and Peripheral Interrupt Sources 204

9.3-1 Low-power Clocks 223
9.3-2 The Triggering Conditions for the RTC Timer 223
9.4-1 Predefined Power Modes 227
9.4-2 Wakeup Source 228

10.4-1 UNIT*n* Configuration Bits 272
10.4-2 Trigger Point 273
10.4-3 Synchronization Operation 273

11.2-1 Alarm Generation When Up-Down Counter Increments 290
11.2-2 Alarm Generation When Up-Down Counter Decrements 290

14.4-1 ROM Address 317
14.4-2 Access Configuration to ROM (ROM0 and ROM1) 318
14.4-3 SRAM Address 318
14.4-4 Access Configuration to Internal SRAM0 319
14.4-5 Internal SRAM1 Split Regions 320
14.4-6 Access Configuration to the Instruction Region of Internal SRAM1 322
14.4-7 Access Configuration to the Data Region of Internal SRAM1 322
14.4-8 RTC FAST Memory Address 322
14.4-9 Split RTC FAST Memory into the Higher Region and the Lower Region 323
14.4-10 Access Configuration to the RTC FAST Memory 323
14.5-1 Access Configuration of the Peripherals 324
14.5-2 Access Configuration of Peri Regions 325
14.6-1 Split the External Memory into Split Regions 326
14.6-2 Access Configuration of Flash Regions 326
14.6-3 Cache Virtual Address Region 327
14.6-4 Split IBUS Cache Virtual Address into 4 Regions 327
14.6-5 Split DBUS Cache Virtual Address into 4 Regions 327
14.6-6 Access Configuration of IBUS to Split Regions 328
14.6-7 Access Configuration of DBUS to Split Regions 328
14.7-1 Interrupt Registers for Unauthorized IBUS Access 329
14.7-2 Interrupt Registers for Unauthorized DBUS Access 329
14.7-3 Interrupt Registers for Unauthorized Access to External Memory 330
14.7-4 Interrupt Registers for Unauthorized Access to Internal Memory via GDMA 330
14.7-5 Interrupt Registers for Unauthorized PIF Access 331
14.7-6 All Possible Access Alignment and their Results 331
14.7-7 Interrupt Registers for Unauthorized Access Alignment 332
14.8-1 Lock Registers and Related Permission Control Registers 332

16.3-1 Memory Controlling Bit 428
16.3-2 Clock Gating and Reset Bits 430

17.4-1 CPU Packet Format 450
17.4-2 DMA Packet Format 450
17.4-3 DMA Source 450
17.4-4 Written Data Format 451

18.3-1 AES Accelerator Working Mode 474
18.3-2 Key Length and Encryption/Decryption 474
18.4-1 Working Status under Typical AES Working Mode 475
18.4-2 Text Endianness Type for Typical AES 475
18.4-3 Key Endianness Type for AES-128 Encryption and Decryption 476
18.4-4 Key Endianness Type for AES-256 Encryption and Decryption 476
18.5-1 Block Cipher Mode 477
18.5-2 Working Status under DMA-AES Working mode 478
18.5-3 TEXT-PADDING 478
18.5-4 Text Endianness for DMA-AES 479

19.2-1 HMAC Purposes and Configuration Value 488

20.3-1 Acceleration Performance 506
20.4-1 RSA Accelerator Memory Blocks 507

21.3-1 SHA Accelerator Working Mode 513
21.3-2 SHA Hash Algorithm Selection 514
21.4-1 The Storage and Length of Message Digest from Different Algorithms 518

23.4-1 $Key$ generated based on $Key_A$ 536
23.4-2 Mapping Between Offsets and Registers 537

26.5-1 UART*n* Synchronous Registers 563
26.5-2 UART*n* Static Registers 564

27.5-1 Data Modes Supported by GP-SPI2 611
27.5-2 Mapping of FSPI Bus Signals 611
27.5-3 Functional Description of FSPI Bus Signals 611
27.5-4 Signals Used in Various SPI Modes 613
27.5-5 Bit Order Control in GP-SPI2 Master and Slave Modes 614
27.5-6 Supported Transfers in Master and Slave Modes 614
27.5-7 Registers Used for State Control in 1/2/4-bit Modes 623
27.5-7 Registers Used for State Control in 1/2/4-bit Modes 624
27.5-8 GP-SPI2 Master BM Table for CONF State 631
27.5-9 An Example of CONF buffer*i* in Segment*i* 632
27.5-10 BM Bit Value v.s. Register to Be Updated in This Example 632
27.5-11 Supported CMD Values in SPI Mode 635
27.5-11 Supported CMD Values in SPI Mode 636
27.5-12 Supported CMD Values in QPI Mode 636
27.7-1 Clock Phase and Polarity Configuration in Master Mode 642
27.7-2 Clock Phase and Polarity Configuration in Slave Mode 642
27.9-1 GP-SPI2 Master Mode Interrupts 645
27.9-1 GP-SPI2 Master Mode Interrupts 646
27.9-2 GP-SPI2 Slave Mode Interrupts 646

28.4-1 I2C Synchronous Registers 681

29.4-1 I2S Signal Description 733
29.9-1 Bit Width of Channel Valid Data 740
29.9-2 Endian of Channel Valid Data 740
29.9-3 Data-Fetching Control in PDM TX Mode 743
29.9-4 I2S Channel Control in Normal PDM TX Mode 744
29.9-5 PCM-to-PDM TX Mode 744
29.10-1 Channel Storage Data Width 747
29.10-2 Channel Storage Data Endian 747

30.3-1 Standard CDC-ACM Control Requests 766
30.3-2 CDC-ACM Settings with RTS and DTR 766
30.3-3 Commands of a Nibble 768

30.3-4 USB-to-JTAG Control Requests 770
30.3-5 JTAG Capabilities Descriptor 770
30.4-1 Reset SoC into Download Mode 771
30.4-2 Reset SoC into Booting 772

31.2-1 Data Frames and Remote Frames in SFF and EFF 791
31.2-2 Error Frame 793
31.2-3 Overload Frame 793
31.2-4 Interframe Space 794
31.2-5 Segments of a Nominal Bit Time 797
31.4-1 Bit Information of TWAI_BUS_TIMING_0_REG (0x18) 801
31.4-2 Bit Information of TWAI_BUS_TIMING_1_REG (0x1c) 801
31.4-3 Buffer Layout for Standard Frame Format and Extended Frame Format 804
31.4-4 TX/RX Frame Information (SFF/EFF); TWAI Address 0x40 805
31.4-5 TX/RX Identifier 1 (SFF); TWAI Address 0x44 805
31.4-6 TX/RX Identifier 2 (SFF); TWAI Address 0x48 805
31.4-7 TX/RX Identifier 1 (EFF); TWAI Address 0x44 806
31.4-8 TX/RX Identifier 2 (EFF); TWAI Address 0x48 806
31.4-9 TX/RX Identifier 3 (EFF); TWAI Address 0x4c 806
31.4-10 TX/RX Identifier 4 (EFF); TWAI Address 0x50 806
31.4-11 Bit Information of TWAI_ERR_CODE_CAP_REG (0x30) 811
31.4-12 Bit Information of Bits SEG.4 - SEG.0 811
31.4-13 Bit Information of TWAI_ARB LOST CAP_REG (0x2c) 812

32.3-1 Commonly-used Frequencies and Resolutions 830
32.3-1 Commonly-used Frequencies and Resolutions 831

33.3-1 Configuration Update 849

34.2-1 SAR ADC Input Signals 870
34.3-1 Temperature Offset 877

34.6-4 Configuration of ENA/RAW/ST Registers 896

# List of Figures


1.1-1 CPU Block Diagram 31
1.6-1 Debug System Overview 47

2.1-1 Modules with GDMA Feature and GDMA Channels 60
2.3-1 GDMA Engine Architecture 61
2.4-1 Structure of a Linked List 62
2.4-2 Relationship among Linked Lists 64

3.2-1 System Structure and Address Mapping 92
3.3-1 Cache Structure 96
3.3-2 Peripherals/modules that can work with GDMA 97

4.3-1 Shift Register Circuit (first 32 output) 107
4.3-2 Shift Register Circuit (last 12 output) 107

5.3-1 Diagram of IO MUX and GPIO Matrix 160
5.3-2 Architecture of IO MUX and GPIO Matrix 160
5.3-3 Internal Structure of a Pad 161
5.4-1 GPIO Input Synchronized on APB Clock Rising Edge or on Falling Edge 162
5.4-2 Filter Timing of GPIO Input Signals 162

6.1-1 Reset Types 191
6.2-1 System Clock 194

8.2-1 Interrupt Matrix Structure 202

9.3-1 Low-power Management Schematics 220
9.3-2 Power Management Unit Workflow 221
9.3-3 RTC Clocks 222
9.3-4 Wireless Clock 223
9.3-5 Digital System Regulator 225
9.3-6 Low-power voltage regulator 225
9.3-7 Brown-out detector 226
9.6-1 ESP32-C3 Boot Flow 230

10.1-1 System Timer Structure 270
10.4-1 System Timer Alarm Generation 271

11.1-1 Timer Units within Groups 288
11.2-1 Timer Group Architecture 289

12.1-1 Watchdog Timers Overview 305
12.2-1 Watchdog Timers in ESP32-C3 307
12.3-1 Super Watchdog Controller Structure 310

13.1-1 XTAL32K Watchdog Timer 312

14.1-1 Permission Control Overview 316

14.4-1 Split Lines for Internal SRAM1 319
14.4-2 An illustration of Configuring the Category fields 321
14.6-1 Two Ways to Access External Memory 326

15.4-1 Switching From Secure World to Non-secure World 415
15.4-2 Switching From Non-secure World to Secure World 416
15.5-1 World Switch Log Register 417
15.5-2 Nested Interrupts Handling - Entry 9 418
15.5-3 Nested Interrupts Handling - Entry 1 419
15.5-4 Nested Interrupts Handling - Entry 4 419

19.3-1 HMAC SHA-256 Padding Diagram 491
19.3-2 HMAC Structure Schematic Diagram 492

22.3-1 Software Preparations and Hardware Working Process 525

23.3-1 Architecture of the External Memory Encryption and Decryption 535

24.3-1 Noise Source 544

25.2-1 XTAL_CLK Pulse Width 547

26.3-1 UART Architecture Overview 550
26.3-2 UART Structure 550
26.4-1 UART Controllers Sharing RAM 552
26.4-2 UART Controllers Division 554
26.4-3 The Timing Diagram of Weak UART Signals Along Falling Edges 554
26.4-4 Structure of UART Data Frame 555
26.4-5 AT_CMD Character Structure 556
26.4-6 Driver Control Diagram in RS485 Mode 557
26.4-7 The Timing Diagram of Encoding and Decoding in SIR mode 558
26.4-8 IrDA Encoding and Decoding Diagram 558
26.4-9 Hardware Flow Control Diagram 559
26.4-10 Connection between Hardware Flow Control Signals 560
26.4-11 Data Transfer in GDMA Mode 561
26.5-1 UART Programming Procedures 565

27.4-1 SPI Module Overview 610
27.5-1 Data Buffer Used in CPU-Controlled Transfer 615
27.5-2 GP-SPI2 Block Diagram 618
27.5-3 Data Flow Control in GP-SPI2 Master Mode 619
27.5-4 Data Flow Control in GP-SPI2 Slave Mode 620
27.5-5 GP-SPI2 State Machine in Master Mode 622
27.5-6 Full-Duplex Communication Between GP-SPI2 Master and a Slave 626
27.5-7 Connection of GP-SPI2 to Flash and External RAM in 4-bit Mode 629
27.5-8 SPI Quad I/O Read Command Sequence Sent by GP-SPI2 to Flash 629
27.5-9 Configurable Segmented Transfer in DMA-Controlled Master Mode 630
27.6-1 Recommended CS Timing and Settings When Accessing External RAM 639
27.6-2 Recommended CS Timing and Settings When Accessing Flash 640

27.7-1 SPI Clock Mode 0 or 2 641
27.7-2 SPI Clock Mode 1 or 3 641
27.8-1 Timing Compensation Control Diagram in GP-SPI2 Master Mode 643
27.8-2 Timing Compensation Example in GP-SPI2 Master Mode 644

28.3-1 I2C Master Architecture 677
28.3-2 I2C Slave Architecture 677
28.3-3 I2C Protocol Timing (Cited from Fig.31 in The I2C-bus specification Version 2.1) 678
28.3-4 I2C Timing Parameters (Cited from Table 5 in The I2C-bus specification Version 2.1) 679
28.4-1 I2C Timing Diagram 682
28.4-2 Structure of I2C Command Registers 684
28.5-1 $I2C_{master}$ Writing to $I2C_{slave}$ with a 7-bit Address 688
28.5-2 $I2C_{master}$ Writing to a Slave with a 10-bit Address 690
28.5-3 $I2C_{master}$ Writing to $I2C_{slave}$ with Two 7-bit Addresses 692
28.5-4 $I2C_{master}$ Writing to $I2C_{slave}$ with a 7-bit Address in Multiple Sequences 694
28.5-5 $I2C_{master}$ Reading $I2C_{slave}$ with a 7-bit Address 696
28.5-6 $I2C_{master}$ Reading $I2C_{slave}$ with a 10-bit Address 698
28.5-7 $I2C_{master}$ Reading N Bytes of Data from addrM of $I2C_{slave}$ with a 7-bit Address 701
28.5-8 $I2C_{master}$ Reading $I2C_{slave}$ with a 7-bit Address in Segments 704

29.4-1 ESP32-C3 I2S System Diagram 732
29.5-1 TDM Philips Standard Timing Diagram 734
29.5-2 TDM MSB Alignment Standard Timing Diagram 735
29.5-3 TDM PCM Standard Timing Diagram 735
29.5-4 PDM Standard Timing Diagram 736
29.6-1 I2S Clock 736
29.9-1 TX Data Format Control 741
29.9-2 TDM Channel Control 743
29.9-3 PDM Channel Control Example 745

30.2-1 USB Serial/JTAG High Level Diagram 765
30.2-2 USB Serial/JTAG Block Diagram 766

31.2-1 Bit Fields in Data Frames and Remote Frames 791
31.2-2 Fields of an Error Frame 793
31.2-3 Fields of an Overload Frame 793
31.2-4 The Fields within an Interframe Space 794
31.2-5 Layout of a Bit 797
31.3-1 TWAI Overview Diagram 798
31.4-1 Acceptance Filter 807
31.4-2 Single Filter Mode 808
31.4-3 Dual Filter Mode 809
31.4-4 Error State Transition 810
31.4-5 Positions of Arbitration Lost Bits 812

32.2-1 LED PWM Architecture 827
32.3-1 LED PWM Generator Diagram 828
32.3-2 Frequency Division When LEDC_CLK_DIV is a Non-Integer Value 829

32.3-3 LED_PWM Output Signal Diagram 832
32.3-4 Output Signal Diagram of Fading Duty Cycle 832

33.3-1 RMT Architecture 844
33.3-2 Format of Pulse Code in RAM 844

34.2-1 SAR ADCs Function Overview 869
34.2-2 Diagram of DIG ADC FSM 872
34.2-3 APB_SARADC_SAR_PATT_TAB1_REG and Pattern Table Entry 0 - Entry 3 873
34.2-4 APB_SARADC_SAR_PATT_TAB2_REG and Pattern Table Entry 4 - Entry 7 873
34.2-5 Pattern Table Entry 873
34.2-6 cmd0 Configuration 874
34.2-7 cmd1 configuration 874
34.2-8 DMA Data Format 874

# Part I

# Microprocessor and Master

This part covers the essential processing elements of the system. Details include controllers for Direct Memory Access (DMA) and RISC-V CPU.

# Chapter 1

# ESP-RISC-V CPU

## 1.1 Overview

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

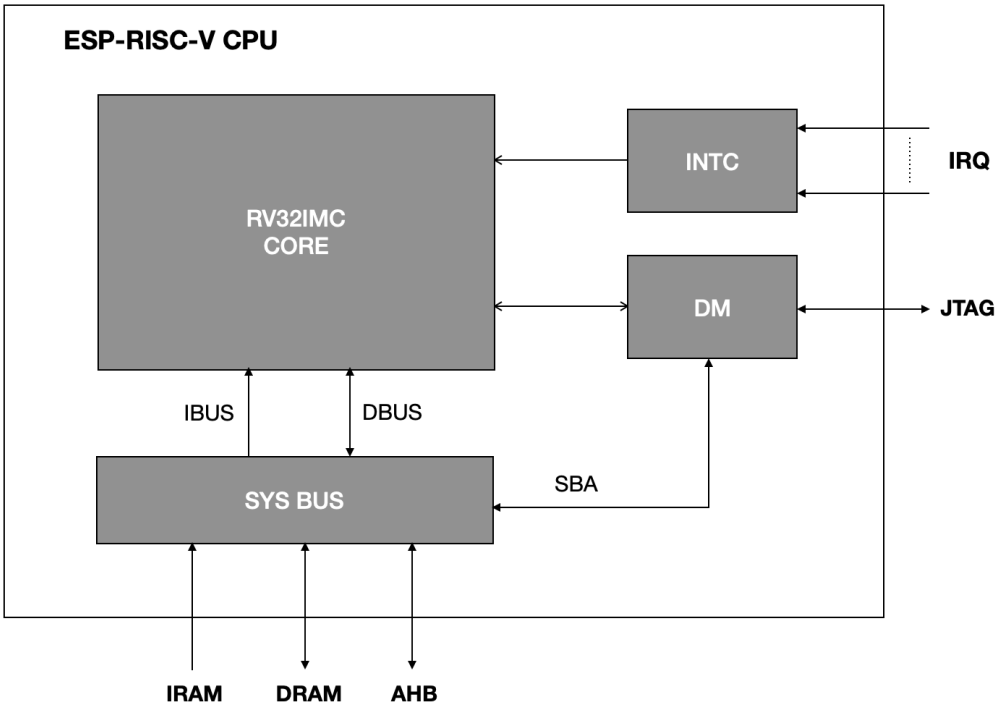



Figure 1.1-1. CPU Block Diagram

## 1.2 Features

- Operating clock frequency up to 160 MHz
- Zero wait cycle access to on-chip SRAM and Cache for program and data access over IRAM/DRAM interface
- Interrupt controller (INTC) with up to 31 vectored interrupts with programmable priority and threshold levels

- Debug module (DM) compliant with RISC-V debug specification v0.13 with external debugger support over an industry-standard JTAG/USB port
- Debugger direct system bus access (SBA) to memory and peripherals
- Hardware trigger compliant to RISC-V debug specification v0.13 with up to 8 breakpoints/watchpoints
- Physical memory protection (PMP) for up to 16 configurable regions
- 32-bit AHB system bus for peripheral access
- Configurable events for core performance metrics

## 1.3 Address Map

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

Table 1.3-1. CPU Address Map

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

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

## 1.4 Configuration and Status Registers (CSRs)

### 1.4.1 Register Summary

Below is a list of CSRs available to the CPU. Except for the custom performance counter CSRs and the tcontrol register (which complies with the RISC-V External Debug Support Version 0.13.2), all the implemented CSRs follow the standard mapping of bit fields as described in the RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10. It must be noted that even among the standard CSRs, not all bit fields have been implemented, limited by the subset of features implemented in the CPU. Refer to the next section for detailed description of the subset of fields implemented under each of these CSRs.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Machine Information CSRs** | | | |
| mvendorid | Machine Vendor ID | 0xF11 | RO |
| marchid | Machine Architecture ID | 0xF12 | RO |
| mimpid | Machine Implementation ID | 0xF13 | RO |
| mhartid | Machine Hart ID | 0xF14 | RO |
| **Machine Trap Setup CSRs** | | | |

| Name | Description | Address | Access |
|---|---|---|---|
| mstatus | Machine Mode Status | 0x300 | R/W |
| misa [1] | Machine ISA | 0x301 | R/W |
| mtvec [2] | Machine Trap Vector | 0x305 | R/W |
| **Machine Trap Handling CSRs** | | | |
| mscratch | Machine Scratch | 0x340 | R/W |
| mepc | Machine Trap Program Counter | 0x341 | R/W |
| mcause [3] | Machine Trap Cause | 0x342 | R/W |
| mtval | Machine Trap Value | 0x343 | R/W |
| **Physical Memory Protection (PMP) CSRs** | | | |
| pmpcfg0 | Physical memory protection configuration | 0x3A0 | R/W |
| pmpcfg1 | Physical memory protection configuration | 0x3A1 | R/W |
| pmpcfg2 | Physical memory protection configuration | 0x3A2 | R/W |
| pmpcfg3 | Physical memory protection configuration | 0x3A3 | R/W |
| pmpaddr0 | Physical memory protection address register | 0x3B0 | R/W |
| pmpaddr1 | Physical memory protection address register | 0x3B1 | R/W |
| .... | | | |
| pmpaddr15 | Physical memory protection address register | 0x3BF | R/W |
| **Trigger Module CSRs (shared with Debug Mode)** | | | |
| tselect | Trigger Select Register | 0x7A0 | R/W |
| tdata1 | Trigger Abstract Data 1 | 0x7A1 | R/W |
| tdata2 | Trigger Abstract Data 2 | 0x7A2 | R/W |
| tcontrol | Global Trigger Control | 0x7A5 | R/W |
| **Debug Mode CSRs** | | | |
| dcsr | Debug Control and Status | 0x7B0 | R/W |
| dpc | Debug PC | 0x7B1 | R/W |
| dscratch0 | Debug Scratch Register 0 | 0x7B2 | R/W |
| dscratch1 | Debug Scratch Register 1 | 0x7B3 | R/W |
| **Performance Counter CSRs (Custom)** [4] | | | |
| mpcer | Machine Performance Counter Event | 0x7E0 | R/W |
| mpcmr | Machine Performance Counter Mode | 0x7E1 | R/W |
| mpccr | Machine Performance Counter Count | 0x7E2 | R/W |
| **GPIO Access CSRs (Custom)** | | | |
| cpu_gpio_oen | GPIO Output Enable | 0x803 | R/W |
| cpu_gpio_in | GPIO Input Value | 0x804 | RO |
| cpu_gpio_out | GPIO Output Value | 0x805 | R/W |

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

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

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

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

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

### 1.4.2 Register Description

Register 1.1. mvendorid (0xF11)

| MVENDORID | |
|---|---|
| 31 … 0 | |
| 0x00000612 | Reset |

**MVENDORID** Vendor ID. (RO)

Register 1.2. marchid (0xF12)

| MARCHID | |
|---|---|
| 31 … 0 | |
| 0x80000001 | Reset |

**MARCHID** Architecture ID. (RO)

Register 1.3. mimpid (0xF13)

| MIMPID | |
|---|---|
| 31 … 0 | |
| 0x00000001 | Reset |

**MIMPID** Implementation ID. (RO)

Register 1.4. mhartid (0xF14)

| MHARTID | |
|---|---|
| 31 … 0 | |
| 0x00000000 | Reset |

**MHARTID** Hart ID. (RO)

**Register 1.5. mstatus (0x300)**

| (reserved) | TW | (reserved) | MPP | (reserved) | MPIE | (reserved) | MIE | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|
| 31 22 | 21 | 20 13 | 12 11 | 10 8 | 7 | 6 4 | 3 | 2 0 | |
| 0x000 | 0 | 0x00 | 0x0 | 0x0 | 0 | 0x0 | 0 | 0x0 | Reset |

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

**MPIE** Machine previous interrupt enable (before trap). (R/W)

**MPP** Machine previous privilege mode (before trap). (R/W)

Possible values:

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

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

**TW** Timeout wait. (R/W)

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

**Register 1.6. misa (0x301)**

| MXL | (reserved) | Z | Y | X | W | V | U | T | S | R | Q | P | O | N | M | L | K | J | I | H | G | F | E | D | C | B | A | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 30 | 29 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 | |
| 0x1 | 0x0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Reset |

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

**Z** Reserved = 0. (RO)

**Y** Reserved = 0. (RO)

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

**W** Reserved = 0. (RO)

**V** Reserved = 0. (RO)

**U** User mode implemented = 1. (RO)

**T** Reserved = 0. (RO)

**S** Supervisor mode implemented = 0. (RO)

**R** Reserved = 0. (RO)

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

**P** Reserved = 0. (RO)

**O** Reserved = 0. (RO)

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

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

**L** Reserved = 0. (RO)

**K** Reserved = 0. (RO)

**J** Reserved = 0. (RO)

**I** RV32I base ISA = 1. (RO)

**H** Hypervisor extension = 0. (RO)

**G** Additional standard extensions present = 0. (RO)

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

**E** RV32E base ISA = 0. (RO)

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

**C** Compressed Extension = 1. (RO)

**B** Reserved = 0. (RO)

**A** Atomic Extension = 0. (RO)

**Register 1.7. mtvec (0x305)**

| BASE | (reserved) | MODE | |
|---|---|---|---|
| 31 … 8 | 7 … 2 | 1 … 0 | |
| 0x000000 | 0x00 | 0x1 | Reset |

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

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

**Register 1.8. mscratch (0x340)**

| MSCRATCH | |
|---|---|
| 31 … 0 | |
| 0x00000000 | Reset |

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

**Register 1.9. mepc (0x341)**

| MEPC | |
|---|---|
| 31 … 0 | |
| 0x00000000 | Reset |

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

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

**Register 1.10. mcause (0x342)**

| Interrupt Flag | (reserved) | Exception Code | |
|---|---|---|---|
| 31 | 30 … 5 | 4 … 0 | |
| 0 | 0x0000000 | 0x00 | Reset |

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

Possible exception IDs are:

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

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

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

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

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

**Register 1.11. mtval (0x343)**

| MTVAL | |
|---|---|
| 31 … 0 | |
| 0x00000000 | Reset |

**MTVAL** Machine trap value. (R/W)

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

Data is to be interpreted depending upon exception IDs:

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

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

**Register 1.12. mpcer (0x7E0)**

| (reserved) | INST_COMP | (BRANCH_TAKEN | BRANCH | JMP_UNCOND | STORE | LOAD | IDLE | JMP_HAZARD | LD_HAZARD | INST | CYCLE | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0x000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**INST_COMP** Count Compressed Instructions. (R/W)

**BRANCH_TAKEN** Count Branches Taken. (R/W)

**BRANCH** Count Branches. (R/W)

**JMP_UNCOND** Count Unconditional Jumps. (R/W)

**STORE** Count Stores. (R/W)

**LOAD** Count Loads. (R/W)

**IDLE** Count IDLE Cycles. (R/W)

**JMP_HAZARD** Count Jump Hazards. (R/W)

**LD_HAZARD** Count Load Hazards. (R/W)

**INST** Count Instructions. (R/W)

**CYCLE** Count Clock Cycles. Cycle count does not increment during WFI mode. (R/W)

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

**Register 1.13. mpcmr (0x7E1)**

| (reserved) | COUNT_SAT | COUNT_EN | |
|---|---|---|---|
| 31 … 2 | 1 | 0 | |
| 0 | 1 | 1 | Reset |

**COUNT_SAT** Counter Saturation Control. (R/W)

Possible values:

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

**COUNT_EN** Counter Enable Control. (R/W)

Possible values:

- 0: Disabled
- 1: Enabled

**Register 1.14. mpccr (0x7E2)**

| MPCCR | |
|---|---|
| 31 … 0 | |
| 0x00000000 | Reset |

**MPCCR** Machine Performance Counter Value. (R/W)

**Register 1.15. cpu_gpio_oen (0x803)**

| (reserved) | CPU_GPIO_OEN[7] | CPU_GPIO_OEN[6] | CPU_GPIO_OEN[5] | CPU_GPIO_OEN[4] | CPU_GPIO_OEN[3] | CPU_GPIO_OEN[2] | CPU_GPIO_OEN[1] | CPU_GPIO_OEN[0] | |
|---|---|---|---|---|---|---|---|---|---|
| 31 … 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**CPU_GPIO_OEN** GPIOn (n=0 ~ 21) Output Enable. CPU_GPIO_OEN[7:0] correspond to output enable signals cpu_gpio_out_oen[7:0] in Table 5.11-1 *Peripheral Signals via GPIO Matrix*. CPU_GPIO_OEN value matches that of cpu_gpio_out_oen.

CPU_GPIO_OEN is the enable signal of CPU_GPIO_OUT. (R/W)

- 0: GPIO output disable
- 1: GPIO output enable

**Register 1.16. cpu_gpio_in (0x804)**

| (reserved) | CPU_GPIO_IN[7] | CPU_GPIO_IN[6] | CPU_GPIO_IN[5] | CPU_GPIO_IN[4] | CPU_GPIO_IN[3] | CPU_GPIO_IN[2] | CPU_GPIO_IN[1] | CPU_GPIO_IN[0] | |
|---|---|---|---|---|---|---|---|---|---|
| 31 … 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**CPU_GPIO_IN** GPIOn (n=0 ~ 21) Input Value. It is a CPU CSR to read input value (1=high, 0=low) from SoC GPIO pin.

CPU_GPIO_IN[7:0] correspond to input signals cpu_gpio_in[7:0] in Table 5.11-1 *Peripheral Signals via GPIO Matrix*.

CPU_GPIO_IN[7:0] can only be mapped to GPIO pins through GPIO matrix. For details please refer to Section 5.4 in Chapter *IO MUX and GPIO Matrix (GPIO, IO MUX)*. (RO)

**Register 1.17. cpu_gpio_out (0x805)**

| (reserved) | CPU_GPIO_OUT[7] | CPU_GPIO_OUT[6] | CPU_GPIO_OUT[5] | CPU_GPIO_OUT[4] | CPU_GPIO_OUT[3] | CPU_GPIO_OUT[2] | CPU_GPIO_OUT[1] | CPU_GPIO_OUT[0] | |
|---|---|---|---|---|---|---|---|---|---|
| 31 … 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**CPU_GPIO_OUT** GPIOn (n=0 ~ 21) Output Value. It is a CPU CSR to write value (1=high, 0=low) to SoC GPIO pin. The value takes effect only when CPU_GPIO_OEN is set.

CPU_GPIO_OUT[7:0] correspond to output signals cpu_gpio_out[7:0] in Table 5.11-1 *Peripheral Signals via GPIO Matrix*.

CPU_GPIO_OUT[7:0] can only be mapped to GPIO pins through GPIO matrix. For details please refer to Section 5.5 in Chapter *IO MUX and GPIO Matrix (GPIO, IO MUX)*. (R/W)

## 1.5 Interrupt Controller

### 1.5.1 Features

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

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

For the complete list of interrupt registers and detailed configuration information, please refer to Chapter 8 *Interrupt Matrix (INTERRUPT)*, section 8.4, register group "CPU Interrupt Registers".

### 1.5.2 Functional Description

Each interrupt ID has 5 properties associated with it:

1. Enable State (0-1):
   - Determines if an interrupt is enabled to be captured and serviced by the CPU.
   - Programmed by writing the corresponding bit in INTERRUPT_COREO_CPU_INT_ENABLE_REG.
2. Type (0-1):
   - Enables latching the state of an interrupt signal on its rising edge.
   - Programmed by writing the corresponding bit in INTERRUPT_COREO_CPU_INT_TYPE_REG.
   - An interrupt for which type is kept 0 is referred as a 'level' type interrupt.
   - An interrupt for which type is set to 1 is referred as an 'edge' type interrupt.
3. Priority (1-15):
   - Determines which interrupt, among multiple pending interrupts, the CPU will service first.
   - Programmed by writing to the INTERRUPT_COREO_CPU_INT_PRI_*n*_REG for a particular interrupt ID *n* in range (1-31).
   - Enabled interrupts with priorities zero or less than the threshold value in INTERRUPT_COREO_CPU_INT_THRESH_REG are masked.
   - Priority levels increase from 1 (lowest) to 15 (highest).
   - Interrupts with same priority are statically prioritized by their IDs, lowest ID having highest priority.
4. Pending State (0-1):
   - Reflects the captured state of an enabled and unmasked interrupt signal.

- For each interrupt ID, the corresponding bit in read-only INTERRUPT_COREO_CPU_INT_EIP_STATUS_REG gives its pending state.
- A pending interrupt will cause CPU to enter trap if no other pending interrupt has higher priority.
- A pending interrupt is said to be 'claimed' if it preempts the CPU and causes it to jump to the corresponding trap vector address.
- All pending interrupts which are yet to be serviced are termed as 'unclaimed'.

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

When CPU services a pending interrupt, it:

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

Table 1.5-1 shows the mapping of each interrupt ID with the corresponding trap-vector address. In short, the word aligned trap address for an interrupt with a certain $ID = i$ can be calculated as $(mtvec + 4i)$.

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

**Table 1.5-1. ID wise map of Interrupt Trap-Vector Addresses**

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

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

Upon execution of MRET instruction, the CPU:

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

It is possible to perform software assisted nesting of interrupts inside an ISR as explained in 1.5.3.

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

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

## 1.5.3 Suggested Operation

### 1.5.3.1 Latency Aspects

There is latency involved while configuring the Interrupt Controller.

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

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

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

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

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

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

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

#### 1.5.3.2 Configuration Procedure

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

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

1. save the state of MIE and clear MIE to 0
2. depending upon the type of the interrupt (edge/level), set/unset the *n*th bit of INTERRUPT_COREO_CPU_INT_TYPE_REG
3. set the priority by writing a value to INTERRUPT_COREO_CPU_INT_PRI_*n*_REG in range 1(lowest) to 15 (highest)
4. set the *n*th bit of INTERRUPT_COREO_CPU_INT_ENABLE_REG
5. execute FENCE instruction
6. restore the state of MIE

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

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

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

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

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

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

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

### 1.5.4 Register Summary

The addresses in this section are relative to Interrupt Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

For the complete list of interrupt registers and detailed configuration information, please refer to Chapter 8 *Interrupt Matrix (INTERRUPT)*, section 8.4, register group "CPU Interrupt Registers".

### 1.5.5 Register Description

The addresses in this section are relative to Interrupt Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

For the complete list of interrupt registers and detailed configuration information, please refer to Chapter 8 *Interrupt Matrix (INTERRUPT)*, section 8.4, register group "CPU Interrupt Registers".

# 1.6 Debug

## 1.6.1 Overview

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

Figure 1.6-1 below shows the main components of External Debug Support.

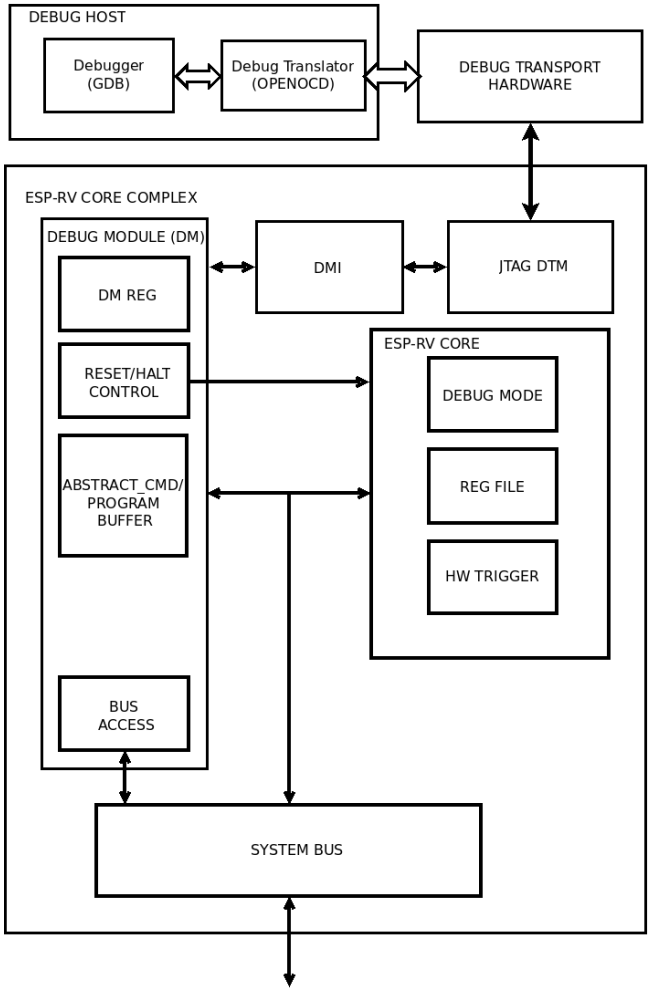



Figure 1.6-1. Debug System Overview

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

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

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

### 1.6.2 Features

Basic debug functionality supports below features.

- Provides necessary information about the implementation to the debugger.
- Allows the CPU core to be halted and resumed.
- CPU core registers (including CSR's) can be read/written by debugger.
- CPU can be debugged from the first instruction executed after reset.
- CPU core can be reset through debugger.
- CPU can be halted on software breakpoint (planted breakpoint instruction).
- Hardware single-stepping.
- Execute arbitrary instructions in the halted CPU by means of the program buffer. 16-word program buffer is supported.
- System bus access is supported through word aligned address access.
- Supports eight Hardware Triggers (can be used as breakpoints/watchpoints) as described in Section 1.7.

### 1.6.3 Functional Description

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

### 1.6.4 Register Summary

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

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

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

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

### 1.6.5 Register Description

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

**Register 1.18. dcsr (0x7B0)**

| xdebugver | reserved | ebreakm | reserved | ebreaku | reserved | stopcount | stoptime | cause | reserved | step | prv | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 28 | 27 16 | 15 | 14 13 | 12 | 11 | 10 | 9 | 8 6 | 5 3 | 2 | 1 0 | |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**xdebugver** Debug version. (RO)

- 4: External debug support exists

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

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

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

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

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

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

Other values are reserved for future use. (RO)

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

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

***Note**: Different from RISC-V Debug specification 0.13

**Register 1.19. dpc (0x7B1)**

| dpc | |
|---|---|
| 31 0 | |
| 0 | Reset |

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

Register 1.20. dscratch0 (0x7B2)

| dscratch0 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

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

Register 1.21. dscratch1 (0x7B3)

| dscratch1 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

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

## 1.7 Hardware Trigger

### 1.7.1 Features

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

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

### 1.7.2 Functional Description

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

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

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

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

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

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

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

Each trigger unit only supports match on address, although this address could either be that of a load/store access or the virtual address of an instruction. The address and size of a region are specified by writing to maddress (tdata2) CSR for the selected trigger unit. Larger than 1 byte region sizes are specified through NAPOT (naturally aligned power of two) encoding (see Table 1.7-1) and enabled by setting match bit in mcontrol. Note that for NAPOT encoded addresses, by definition, the start address is constrained to be aligned to (i.e. an integer multiple of) the region size.

**Table 1.7-1. NAPOT encoding for maddress**

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

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

### 1.7.3 Trigger Execution Flow

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

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

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

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

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

### 1.7.4 Register Summary

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

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

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

### 1.7.5 Register Description

**Register 1.22. tselect (0x7A0)**

| (reserved) | tselect |
|---|---|
| 31 … 3 | 2 … 0 |
| 0x00000000 | 0x0 |

Reset

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

**Register 1.23. tdata1 (0x7A1)**

| type | dmode | data |
|---|---|---|
| 31 … 28 | 27 | 26 … 0 |
| 0x2 | 0 | 0x3e00000 |

Reset

**type** Type of trigger. (RO)

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

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

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

** Note : Only writable from debug mode.*

**data** Abstract tdata1 content. (R/W)

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

**Register 1.24. tdata2 (0x7A2)**

| tdata2 | |
|---|---|
| 31 … 0 | |
| 0x00000000 | Reset |

**tdata2** Abstract tdata2 content. (R/W)

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

**Register 1.25. tcontrol (0x7A5)**

| (reserved) | mpte | (reserved) | mte | (reserved) | |
|---|---|---|---|---|---|
| 31 … 8 | 7 | 6 … 4 | 3 | 2 … 0 | |
| 0x000000 | 0 | 0 | 0 | 0 | Reset |

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

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

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

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

**Register 1.26. mcontrol (0x7A1)**

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

**dmode** Same as dmode in tdata1.

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

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

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

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

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

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

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

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

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

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

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

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

**Register 1.27. maddress (0x7A2)**

| maddress | |
|---|---|
| 31 … 0 | |
| 0x00000000 | Reset |

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

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

# 1.8 Memory Protection

## 1.8.1 Overview

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

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

## 1.8.2 Features

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

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

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

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

## 1.8.3 Functional Description

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

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

When any instruction is being fetched from memory region without execute permissions, exception is generated at processor level and exception cause is set as instruction access fault in mcause CSR. Similarly,

any load/store access without valid read/write permissions, will result in exception generation with mcause updated as load access and store access fault respectively. In case of load/store access faults, violating address is captured in mtval CSR.

### 1.8.4 Register Summary

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

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

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

### 1.8.5 Register Description

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

# Chapter 2

# GDMA Controller (GDMA)

## 2.1 Overview

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

The GDMA controller in ESP32-C3 has six independent channels, i.e. three transmit channels and three receive channels. These six channels are shared by peripherals with GDMA feature, namely SPI2, UHCI0 (UART0/UART1), I2S, AES, SHA, and ADC. Users can assign the six channels to any of these peripherals. UART0 and UART1 use UHCI0 together.

The GDMA controller uses fixed-priority and round-robin channel arbitration schemes to manage peripherals' needs for bandwidth.

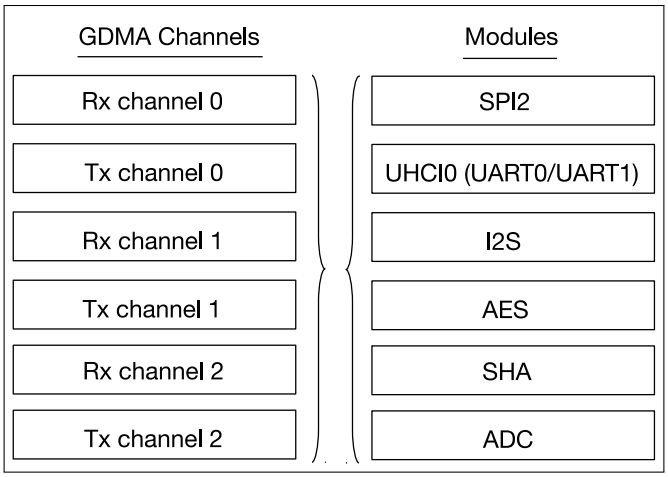



Figure 2.1-1. Modules with GDMA Feature and GDMA Channels

## 2.2 Features

The GDMA controller has the following features:

- AHB bus architecture
- Programmable length of data to be transferred in bytes
- Linked list of descriptors
- INCR burst transfer when accessing internal RAM
- Access to an address space of 384 KB at most in internal RAM

- Three transmit channels and three receive channels
- Software-configurable selection of peripheral requesting its service
- Fixed channel priority and round-robin channel arbitration

## 2.3 Architecture

In ESP32-C3, all modules that need high-speed data transfer support GDMA. The GDMA controller and CPU data bus have access to the same address space in internal RAM. Figure 2.3-1 shows the basic architecture of the GDMA engine.

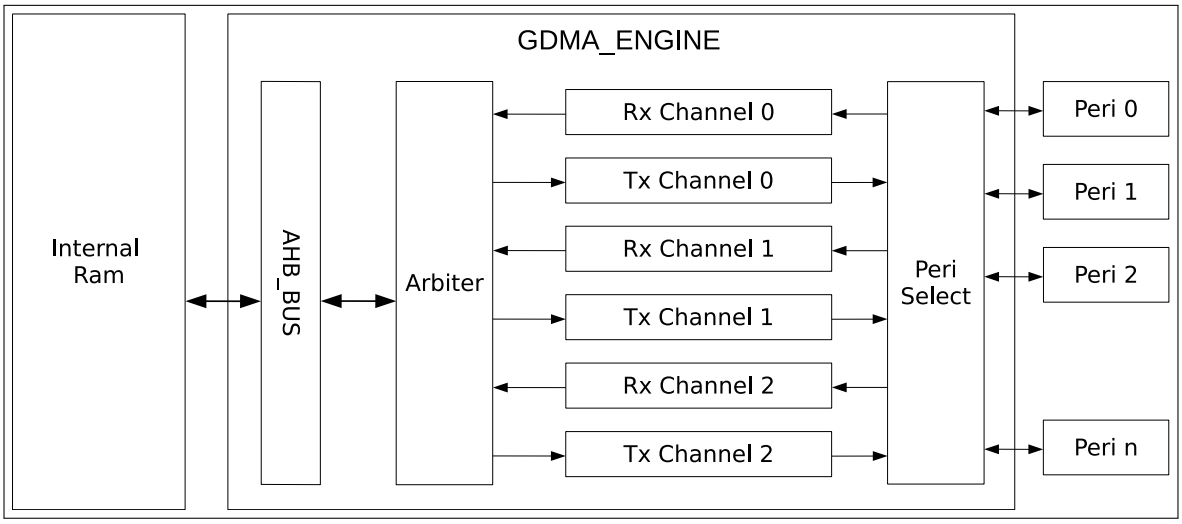



**Figure 2.3-1. GDMA Engine Architecture**

The GDMA controller has six independent channels, i.e. three transmit channels and three receive channels. Every channel can be connected to different peripherals. In other words, channels are general-purpose, shared by peripherals.

The GDMA engine reads data from or writes data to internal RAM via the AHB_BUS. Before this, the GDMA controller uses fixed-priority arbitration scheme for channels requesting read or write access. For available address range of Internal RAM, please see Chapter 3 *System and Memory*.

Software can use the GDMA engine through linked lists. These linked lists, stored in internal RAM, consist of outlink*n* and inlink*n*, where *n* indicates the channel number (ranging from 0 to 2). The GDMA controller reads an outlink*n* (i.e. a linked list of transmit descriptors) from internal RAM and transmits data in corresponding RAM according to the outlink*n*, or reads an inlink*n* (i.e. a linked list of receive descriptors) and stores received data into specific address space in RAM according to the inlink*n*.

## 2.4 Functional Description

### 2.4.1 Linked List

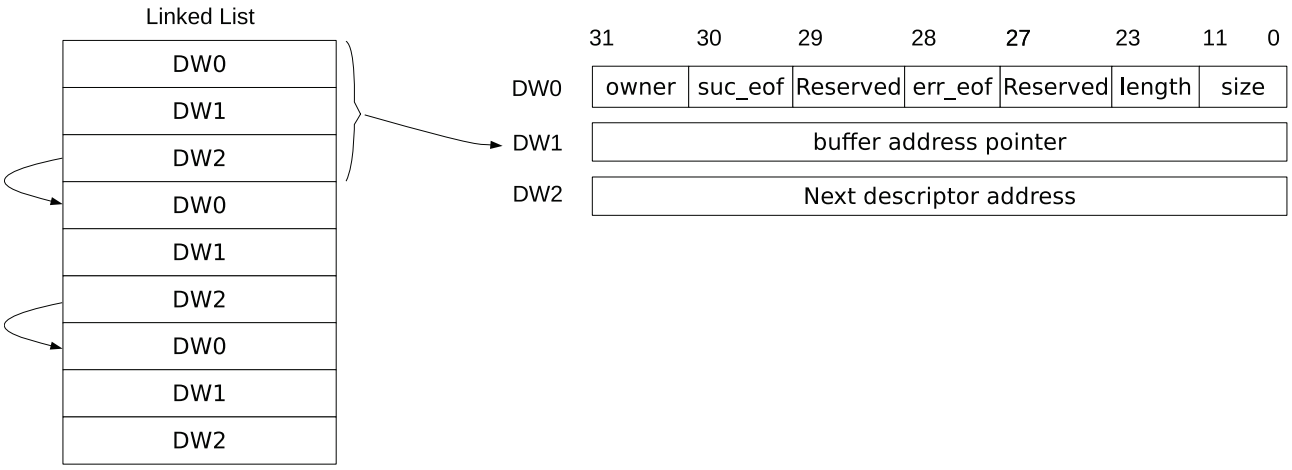



**Figure 2.4-1. Structure of a Linked List**

Figure 2.4-1 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 GDMA 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 GDMA controller can access the buffer.
  When the GDMA controller stops using the buffer, this bit in a receive descriptor is automatically cleared by hardware, and this bit in a transmit descriptor is automatically cleared by hardware only if GDMA_OUT_AUTO_WRBACK_CH*n* is set to 1. Software can disable automatic clearing by hardware by setting GDMA_OUT_LOOP_TEST_CH*n* or GDMA_IN_LOOP_TEST_CH*n* bit. When software loads a linked list, this bit should be set to 1.
  **Note:** GDMA_OUT is the prefix of transmit channel registers, and GDMA_IN is the prefix of receive channel registers.
- suc_eof (DW0) [30]: Specifies whether the GDMA_IN_SUC_EOF_CH*n*_INT or GDMA_OUT_EOF_CH*n*_INT interrupt will be triggered when the data corresponding to this descriptor has been received or transmitted.
  1'b0: No interrupt will be triggered after the current descriptor's successful transfer;
  1'b1: An interrupt will be triggered after the current descriptor's successful transfer.
  For receive descriptors, software needs to clear this bit to 0, and hardware will set it to 1 after receiving data containing the EOF flag.
  For transmit descriptors, software needs to set this bit to 1 as needed.
  If software configures this bit to 1 in a descriptor, the GDMA will include the EOF flag in the data sent to the corresponding peripheral, indicating to the peripheral that this data segment marks the end of one transfer phase.
- Reserved (DW0) [29]: Reserved. Value of this bit does not matter.
- err_eof (DW0) [28]: Specifies whether the received data has errors.
  This bit is used only when UHCI0 uses GDMA to receive data. When an error is detected in the received data segment corresponding to a descriptor, 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 valid bytes have been stored into the buffer.
- Size (DW0) [11:0]: Specifies the size of the buffer that this descriptor points to.
- Buffer address pointer (DW1): Address of the buffer. This field can only point to internal RAM.
- Next descriptor address (DW2): Address of the next descriptor. If the current descriptor is the last one, 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 GDMA controller will not use the available space of the buffer in the next transaction.

### 2.4.2 Peripheral-to-Memory and Memory-to-Peripheral Data Transfer

The GDMA controller can transfer data from memory to peripheral (transmit) and from peripheral to memory (receive). A transmit channel transfers data in the specified memory location to a peripheral's transmitter via an outlink*n*, whereas a receive channel transfers data received by a peripheral to the specified memory location via an inlink*n*.

Every transmit and receive channel can be connected to any peripheral with GDMA feature. Table 2.4-1 illustrates how to select the peripheral to be connected via registers. When a channel is connected to a peripheral, the rest channels can not be connected to that peripheral.

**Table 2.4-1. Selecting Peripherals via Register Configuration**

| GDMA_PERI_IN_SEL_CH*n*<br>GDMA_PERI_OUT_SEL_CH*n* | Peripheral |
|---|---|
| 0 | SPI2 |
| 1 | Reserved |
| 2 | UHCI0 |
| 3 | I2S |
| 4 | Reserved |
| 5 | Reserved |
| 6 | AES |
| 7 | SHA |
| 8 | ADC |
| 9 ~ 63 | Invalid |

### 2.4.3 Memory-to-Memory Data Transfer

The GDMA controller also allows memory-to-memory data transfer. Such data transfer can be enabled by setting GDMA_MEM_TRANS_EN_CH*n*, which connects the output of transmit channel *n* to the input of receive channel *n*. Note that a transmit channel is only connected to the receive channel with the same number (*n*).

### 2.4.4 Enabling GDMA

Software uses the GDMA controller through linked lists. When the GDMA controller receives data, software loads an inlink, configures GDMA_INLINK_ADDR_CH*n* field with address of the first receive descriptor, and sets GDMA_INLINK_START_CH*n* bit to enable GDMA. When the GDMA controller transmits data, software loads an outlink, prepares data to be transmitted, configures GDMA_OUTLINK_ADDR_CH*n* field with address of the first transmit descriptor, and sets GDMA_OUTLINK_START_CH*n* bit to enable GDMA. GDMA_INLINK_START_CH*n* bit and GDMA_OUTLINK_START_CH*n* bit are cleared automatically by hardware.

In some cases, you may want to append more descriptors to a DMA transfer that is already started. Naively, it would seem to be possible to do this by clearing the EOF bit of the final descriptor in the existing list and setting its next descriptor address pointer field (DW2) to the first descriptor of the to-be-added list. However, this strategy fails if the existing DMA transfer is almost or entirely finished. Instead, the GDMA engine has specialized logic to make sure a DMA transfer can be continued or restarted: if it is still ongoing, it will make sure to take the appended descriptors into account; if the transfer has already finished, it will restart with the new descriptors. This is implemented in the Restart function.

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, and set GDMA_INLINK_RESTART_CH*n* bit or GDMA_OUTLINK_RESTART_CH*n* bit (these two bits are cleared automatically by hardware). As shown in Figure 2.4-2, 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.

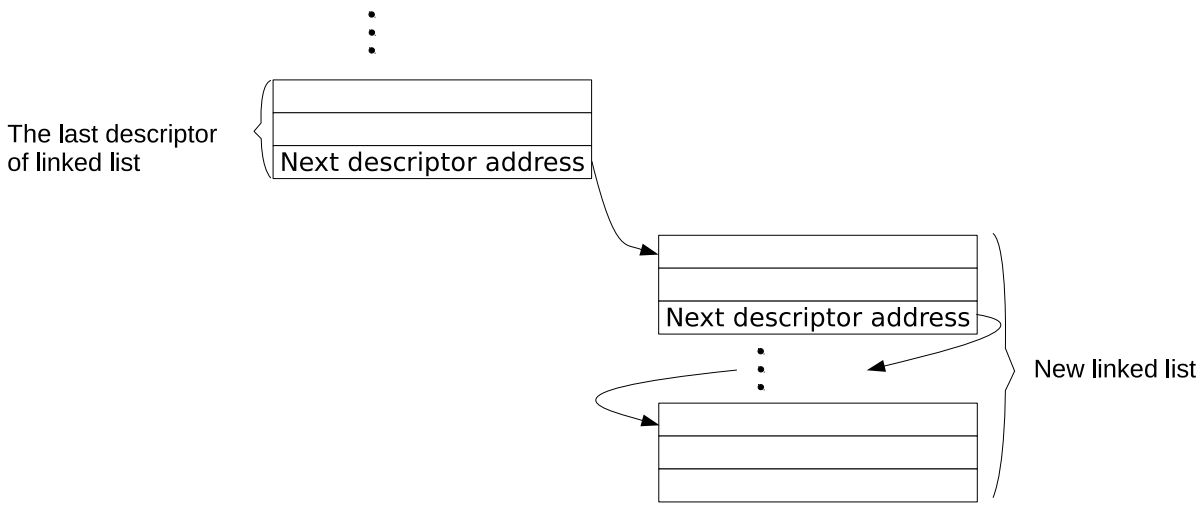



Figure 2.4-2. Relationship among Linked Lists

### 2.4.5 Linked List Reading Process

Once configured and enabled by software, the GDMA controller starts to read the linked list from internal RAM. The GDMA performs checks on descriptors in the linked list. Only if descriptors pass the checks, will the corresponding GDMA channel transfer data. If the descriptors fail any of the checks, hardware will trigger descriptor error interrupt (either GDMA_IN_DSCR_ERR_CH*n*_INT or GDMA_OUT_DSCR_ERR_CH*n*_INT), and the channel will halt.

The checks performed on descriptors are:

- Owner bit check when GDMA_IN_CHECK_OWNER_CH*n* or GDMA_OUT_CHECK_OWNER_CH*n* is set to 1.

If the owner bit is 0, the buffer is accessed by the CPU. In this case, the owner bit fails the check. The owner bit will not be checked if GDMA_IN_CHECK_OWNER_CH*n* or GDMA_OUT_CHECK_OWNER_CH*n* is 0;

- Buffer address pointer (DW1) check. If the buffer address pointer points to 0x3FC80000 ~ 0x3FCDFFFF (please refer to Section 2.4.7), it passes the check.

After software detects a descriptor error interrupt, it must reset the corresponding channel, and enable GDMA by setting GDMA_OUTLINK_START_CH*n* or GDMA_INLINK_START_CH*n* bit.

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

### 2.4.6 EOF

The GDMA controller uses EOF (end of frame) flags to indicate the end of data segment transfer corresponding to a specific descriptor.

Before the GDMA controller transmits data, GDMA_OUT_TOTAL_EOF_CH*n*_INT_ENA bit should be set to enable GDMA_OUT_TOTAL_EOF_CH*n*_INT interrupt. If data in the buffer pointed by the last descriptor (with EOF) has been transmitted, a GDMA_OUT_TOTAL_EOF_CH*n*_INT interrupt is generated.

Before the GDMA controller receives data, GDMA_IN_SUC_EOF_CH*n*_INT_ENA bit should be set to enable GDMA_IN_SUC_EOF_CH*n*_INT interrupt. If a data segment with an EOF flag has been received successfully, a GDMA_IN_SUC_EOF_CH*n*_INT interrupt is generated. In addition, when GDMA channel is connected to UHCI0, the GDMA controller also supports GDMA_IN_ERR_CH*n*_EOF_INT interrupt. This interrupt is enabled by setting GDMA_IN_ERR_EOF_CH*n*_INT_ENA bit, and it indicates that a data segment corresponding to a descriptor has been received with errors.

When detecting a GDMA_OUT_TOTAL_EOF_CH*n*_INT or a GDMA_IN_SUC_EOF_CH*n*_INT interrupt, software can record the value of GDMA_OUT_EOF_DES_ADDR_CH*n* or GDMA_IN_SUC_EOF_DES_ADDR_CH*n* field, i.e. address of the last descriptor. Therefore, software can tell which descriptors have been used and reclaim them.

**Note:** In this chapter, EOF of transmit descriptors refers to suc_eof, while EOF of receive descriptors refers to both suc_eof and err_eof.

### 2.4.7 Accessing Internal RAM

Any transmit and receive channels of GDMA can access 0x3FC80000 ~ 0x3FCDFFFF in internal RAM. To improve data transfer efficiency, GDMA can send data in burst mode, which is disabled by default. This mode is enabled for receive channels by setting GDMA_IN_DATA_BURST_EN_CH*n*, and enabled for transmit channels by setting GDMA_OUT_DATA_BURST_EN_CH*n*.

**Table 2.4-2. Descriptor Field Alignment Requirements**

| Inlink/Outlink | Burst Mode | Size | Length | Buffer Address Pointer |
|---|---|---|---|---|
| Inlink | 0 | — | — | — |
| | 1 | Word-aligned | — | Word-aligned |
| Outlink | 0 | — | — | — |
| | 1 | — | — | — |

Table 2.4-2 lists the requirements for descriptor field alignment when accessing internal RAM.

When burst mode is disabled, size, length, and buffer address pointer in both transmit and receive descriptors do not need to be word-aligned. That is to say, GDMA can read data of specified length (1 ~ 4095 bytes) from any start addresses in the accessible address range, or write received data of the specified length (1 ~ 4095 bytes) to any contiguous addresses in the accessible address range.

When burst mode is enabled, size, length, and buffer address pointer in transmit descriptors are also not necessarily word-aligned. However, size and buffer address pointer in receive descriptors except length should be word-aligned.

### 2.4.8 Arbitration

To ensure timely response to peripherals running at a high speed with low latency (such as SPI), the GDMA controller implements a fixed-priority channel arbitration scheme. That is to say, each channel can be assigned a priority from 0 ~ 9. The larger the number, the higher the priority, and the more timely the response. When several channels are assigned the same priority, the GDMA controller adopts a round-robin arbitration scheme.

Please note that the overall throughput of peripherals with GDMA feature cannot exceed the maximum bandwidth of the GDMA, so that requests from low-priority peripherals can be responded to.

## 2.5 GDMA Interrupts

- DMA_INFIFO_OVF_CH*n*_INT: Triggered when the RX FIFO of GDMA overflows.
- GDMA_INFIFO_UDF_CH*n*_INT: Triggered when the RX FIFO of GDMA underflows.
- GMA_OUTFIFO_OVF_CH*n*_INT: Triggered when the TX FIFO of GDMA overflows.
- GDMA_OUTFIFO_UDF_CH*n*_INT: Triggered when the TX FIFO of GDMA underflows.
- GDMA_OUT_TOTAL_EOF_CH*n*_INT: Triggered when all data corresponding to a linked list (including multiple descriptors) has been sent via transmit channel *n*.
- GDMA_IN_DSCR_EMPTY_CH*n*_INT: Triggered when the size of the buffer pointed by receive descriptors is smaller than the length of data to be received via receive channel *n*.
- GDMA_OUT_DSCR_ERR_CH*n*_INT: Triggered when an error is detected in a transmit descriptor on transmit channel *n*.
- GDMA_IN_DSCR_ERR_CH*n*_INT: Triggered when an error is detected in a receive descriptor on receive channel *n*.
- GDMA_OUT_EOF_CH*n*_INT: Triggered when EOF in a transmit descriptor is 1 and data corresponding to this descriptor has been sent via transmit channel *n*. If GDMA_OUT_EOF_MODE_CH*n* is 0, this interrupt will be triggered when the last byte of data corresponding to this descriptor enters GDMA's transmit channel; if GDMA_OUT_EOF_MODE_CH*n* is 1, this interrupt is triggered when the last byte of data is taken from GDMA's transmit channel.

- GDMA_OUT_DONE_CH*n*_INT: Triggered when all data corresponding to a transmit descriptor has been sent via transmit channel *n*.
- GDMA_IN_ERR_EOF_CH*n*_INT: Triggered when an error is detected in the data segment corresponding to a descriptor received via receive channel *n*. This interrupt is used only for UHCI0 peripheral (UART0 or UART1).
- GDMA_IN_SUC_EOF_CH*n*_INT: Triggered when the suc_eof bit in a receive descriptor is 1 and the data corresponding to this receive descriptor has been received via receive channel *n*.
- GDMA_IN_DONE_CH*n*_INT: Triggered when all data corresponding to a receive descriptor has been received via receive channel *n*.

## 2.6 Programming Procedures

### 2.6.1 Programming Procedure for GDMA Clock and Reset

GDMA's clock and reset should be configured as follows:

1. Set SYSTEM_DMA_CLK_EN to enable GDMA's clock;
2. Clear SYSTEM_DMA_RST to reset GDMA.

### 2.6.2 Programming Procedures for GDMA's Transmit Channel

To transmit data, GDMA's transmit channel should be configured by software as follows:

1. Set GDMA_OUT_RST_CH*n* first to 1 and then to 0, to reset the state machine of GDMA's transmit channel and FIFO pointer;
2. Load an outlink, and configure GDMA_OUTLINK_ADDR_CH*n* with address of the first transmit descriptor;
3. Configure GDMA_PERI_OUT_SEL_CH*n* with the value corresponding to the peripheral to be connected, as shown in Table 2.4-1;
4. Set GDMA_OUTLINK_START_CH*n* to enable GDMA's transmit channel for data transfer;
5. Configure and enable the corresponding peripheral (SPI2, UHCI0 (UART0 or UART1), I2S, AES, SHA, and ADC). See details in individual chapters of these peripherals;
6. Wait for GDMA_OUT_TOTAL_EOF_CH*n*_INT interrupt, which indicates the completion of data transfer.

### 2.6.3 Programming Procedures for GDMA's Receive Channel

To receive data, GDMA's receive channel should be configured by software as follows:

1. Set GDMA_IN_RST_CH*n* first to 1 and then to 0, to reset the state machine of GDMA's receive channel and FIFO pointer;
2. Load an inlink, and configure GDMA_INLINK_ADDR_CH*n* with address of the first receive descriptor;
3. Configure GDMA_PERI_IN_SEL_CH*n* with the value corresponding to the peripheral to be connected, as shown in Table 2.4-1;
4. Set GDMA_INLINK_START_CH*n* to enable GDMA's receive channel for data transfer;

5. Configure and enable the corresponding peripheral (SPI2, UHCI0 (UART0 or UART1), I2S, AES, SHA, and ADC). See details in individual chapters of these peripherals;

### 2.6.4 Programming Procedures for Memory-to-Memory Transfer

To transfer data from one memory location to another, GDMA should be configured by software as follows:

1. Set GDMA_OUT_RST_CH*n* first to 1 and then to 0, to reset the state machine of GDMA's transmit channel and FIFO pointer;
2. Set GDMA_IN_RST_CH*n* first to 1 and then to 0, to reset the state machine of GDMA's receive channel and FIFO pointer;
3. Load an outlink, and configure GDMA_OUTLINK_ADDR_CH*n* with address of the first transmit descriptor;
4. Load an inlink, and configure GDMA_INLINK_ADDR_CH*n* with address of the first receive descriptor;
5. Set GDMA_MEM_TRANS_EN_CH*n* to enable memory-to-memory transfer;
6. Set GDMA_OUTLINK_START_CH*n* to enable GDMA's transmit channel for data transfer;
7. Set GDMA_INLINK_START_CH*n* to enable GDMA's receive channel for data transfer;
8. If the suc_eof bit is set in a transmit descriptor, a GDMA_IN_SUC_EOF_CH*n*_INT interrupt will be triggered when the data segment corresponding to this descriptor has been transmitted.

## 2.7 Register Summary

The addresses in this section are relative to GDMA base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Interrupt Registers** | | | |
| GDMA_INT_RAW_CH0_REG | Raw status interrupt of RX channel 0 | 0x0000 | R/WTC/SS |
| GDMA_INT_ST_CH0_REG | Masked interrupt of RX channel 0 | 0x0004 | RO |
| GDMA_INT_ENA_CH0_REG | Interrupt enable bits of RX channel 0 | 0x0008 | R/W |
| GDMA_INT_CLR_CH0_REG | Interrupt clear bits of RX channel 0 | 0x000C | WT |
| GDMA_INT_RAW_CH1_REG | Raw status interrupt of RX channel 1 | 0x0010 | R/WTC/SS |
| GDMA_INT_ST_CH1_REG | Masked interrupt of RX channel 1 | 0x0014 | RO |
| GDMA_INT_ENA_CH1_REG | Interrupt enable bits of RX channel 1 | 0x0018 | R/W |
| GDMA_INT_CLR_CH1_REG | Interrupt clear bits of RX channel 1 | 0x001C | WT |
| GDMA_INT_RAW_CH2_REG | Raw status interrupt of RX channel 2 | 0x0020 | R/WTC/SS |
| GDMA_INT_ST_CH2_REG | Masked interrupt of RX channel 2 | 0x0024 | RO |
| GDMA_INT_ENA_CH2_REG | Interrupt enable bits of RX channel 2 | 0x0028 | R/W |
| GDMA_INT_CLR_CH2_REG | Interrupt clear bits of RX channel 2 | 0x002C | WT |
| **Configuration Register** | | | |
| GDMA_MISC_CONF_REG | Miscellaneous register | 0x0044 | R/W |
| **Version Registers** | | | |
| GDMA_DATE_REG | Version control register | 0x0048 | R/W |
| **Configuration Registers** | | | |
| GDMA_IN_CONF0_CH0_REG | Configuration register 0 of RX channel 0 | 0x0070 | R/W |
| GDMA_IN_CONF1_CH0_REG | Configuration register 1 of RX channel 0 | 0x0074 | R/W |
| GDMA_IN_POP_CH0_REG | Pop control register of RX channel 0 | 0x007C | varies |
| GDMA_IN_LINK_CH0_REG | Link descriptor configuration and control register of RX channel 0 | 0x0080 | varies |
| GDMA_OUT_CONF0_CH0_REG | Configuration register 0 of TX channel 0 | 0x00D0 | R/W |
| GDMA_OUT_CONF1_CH0_REG | Configuration register 1 of TX channel 0 | 0x00D4 | R/W |
| GDMA_OUT_PUSH_CH0_REG | Push control register of TX channel 0 | 0x00DC | varies |
| GDMA_OUT_LINK_CH0_REG | Link descriptor configuration and control register of TX channel 0 | 0x00E0 | varies |
| GDMA_IN_CONF0_CH1_REG | Configuration register 0 of RX channel 1 | 0x0130 | R/W |
| GDMA_IN_CONF1_CH1_REG | Configuration register 1 of RX channel 1 | 0x0134 | R/W |
| GDMA_IN_POP_CH1_REG | Pop control register of RX channel 1 | 0x013C | varies |
| GDMA_IN_LINK_CH1_REG | Link descriptor configuration and control register of RX channel 1 | 0x0140 | varies |
| GDMA_OUT_CONF0_CH1_REG | Configuration register 0 of TX channel 1 | 0x0190 | R/W |
| GDMA_OUT_CONF1_CH1_REG | Configuration register 1 of TX channel 1 | 0x0194 | R/W |
| GDMA_OUT_PUSH_CH1_REG | Push control register of TX channel 1 | 0x019C | varies |

| Name | Description | Address | Access |
|---|---|---|---|
| GDMA_OUT_LINK_CH1_REG | Link descriptor configuration and control register of TX channel 1 | 0x01A0 | varies |
| GDMA_IN_CONF0_CH2_REG | Configuration register 0 of RX channel 2 | 0x01F0 | R/W |
| GDMA_IN_CONF1_CH2_REG | Configuration register 1 of RX channel 2 | 0x01F4 | R/W |
| GDMA_IN_POP_CH2_REG | Pop control register of RX channel 2 | 0x01FC | varies |
| GDMA_IN_LINK_CH2_REG | Link descriptor configuration and control register of RX channel 2 | 0x0200 | varies |
| GDMA_OUT_CONF0_CH2_REG | Configuration register 0 of TX channel 2 | 0x0250 | R/W |
| GDMA_OUT_CONF1_CH2_REG | Configuration register 1 of TX channel 2 | 0x0254 | R/W |
| GDMA_OUT_PUSH_CH2_REG | Push control register of TX channel 2 | 0x025C | varies |
| GDMA_OUT_LINK_CH2_REG | Link descriptor configuration and control register of TX channel 2 | 0x0260 | varies |
| **Status Registers** | | | |
| GDMA_INFIFO_STATUS_CH0_REG | RX FIFO status of RX channel 0 | 0x0078 | RO |
| GDMA_IN_STATE_CH0_REG | Receive status of RX channel 0 | 0x0084 | RO |
| GDMA_IN_SUC_EOF_DES_ADDR_CH0_REG | Inlink descriptor address when EOF occurs of RX channel 0 | 0x0088 | RO |
| GDMA_IN_ERR_EOF_DES_ADDR_CH0_REG | Inlink descriptor address when errors occur of RX channel 0 | 0x008C | RO |
| GDMA_IN_DSCR_CH0_REG | Address of the next receive descriptor pointed by the current pre-read receive descriptor on RX channel 0 | 0x0090 | RO |
| GDMA_IN_DSCR_BF0_CH0_REG | Address of the current pre-read receive descriptor on RX channel 0 | 0x0094 | RO |
| GDMA_IN_DSCR_BF1_CH0_REG | Address of the previous pre-read receive descriptor on RX channel 0 | 0x0098 | RO |
| GDMA_OUTFIFO_STATUS_CH0_REG | TX FIFO status of TX channel 0 | 0x00D8 | RO |
| GDMA_OUT_STATE_CH0_REG | Transmit status of TX channel 0 | 0x00E4 | RO |
| GDMA_OUT_EOF_DES_ADDR_CH0_REG | Outlink descriptor address when EOF occurs of TX channel 0 | 0x00E8 | RO |
| GDMA_OUT_EOF_BFR_DES_ADDR_CH0_REG | The last outlink descriptor address when EOF occurs of TX channel 0 | 0x00EC | RO |
| GDMA_OUT_DSCR_CH0_REG | Address of the next transmit descriptor pointed by the current pre-read transmit descriptor on TX channel 0 | 0x00F0 | RO |
| GDMA_OUT_DSCR_BF0_CH0_REG | Address of the current pre-read transmit descriptor on TX channel 0 | 0x00F4 | RO |
| GDMA_OUT_DSCR_BF1_CH0_REG | Address of the previous pre-read transmit descriptor on TX channel 0 | 0x00F8 | RO |
| GDMA_INFIFO_STATUS_CH1_REG | RX FIFO status of RX channel 1 | 0x0138 | RO |
| GDMA_IN_STATE_CH1_REG | Receive status of RX channel 1 | 0x0144 | RO |
| GDMA_IN_SUC_EOF_DES_ADDR_CH1_REG | Inlink descriptor address when EOF occurs of RX channel 1 | 0x0148 | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| GDMA_IN_ERR_EOF_DES_ADDR_CH1_REG | Inlink descriptor address when errors occur of RX channel 1 | 0x014C | RO |
| GDMA_IN_DSCR_CH1_REG | Address of the next receive descriptor pointed by the current pre-read receive descriptor on RX channel 1 | 0x0150 | RO |
| GDMA_IN_DSCR_BF0_CH1_REG | Address of the current pre-read receive descriptor on RX channel 1 | 0x0154 | RO |
| GDMA_IN_DSCR_BF1_CH1_REG | Address of the previous pre-read receive descriptor on RX channel 1 | 0x0158 | RO |
| GDMA_OUTFIFO_STATUS_CH1_REG | TX FIFO status of TX channel 1 | 0x0198 | RO |
| GDMA_OUT_STATE_CH1_REG | Transmit status of TX channel 1 | 0x01A4 | RO |
| GDMA_OUT_EOF_DES_ADDR_CH1_REG | Outlink descriptor address when EOF occurs of TX channel 1 | 0x01A8 | RO |
| GDMA_OUT_EOF_BFR_DES_ADDR_CH1_REG | The last outlink descriptor address when EOF occurs of TX channel 1 | 0x01AC | RO |
| GDMA_OUT_DSCR_CH1_REG | Address of the next transmit descriptor pointed by the current pre-read transmit descriptor on TX channel 1 | 0x01B0 | RO |
| GDMA_OUT_DSCR_BF0_CH1_REG | Address of the current pre-read transmit descriptor on TX channel 1 | 0x01B4 | RO |
| GDMA_OUT_DSCR_BF1_CH1_REG | Address of the previous pre-read transmit descriptor on TX channel 1 | 0x01B8 | RO |
| GDMA_INFIFO_STATUS_CH2_REG | RX FIFO status of RX channel 2 | 0x01F8 | RO |
| GDMA_IN_STATE_CH2_REG | Receive status of RX channel 2 | 0x0204 | RO |
| GDMA_IN_SUC_EOF_DES_ADDR_CH2_REG | Inlink descriptor address when EOF occurs of RX channel 2 | 0x0208 | RO |
| GDMA_IN_ERR_EOF_DES_ADDR_CH2_REG | Inlink descriptor address when errors occur of RX channel 2 | 0x020C | RO |
| GDMA_IN_DSCR_CH2_REG | Address of the next receive descriptor pointed by the current pre-read receive descriptor on RX channel 2 | 0x0210 | RO |
| GDMA_IN_DSCR_BF0_CH2_REG | Address of the current pre-read receive descriptor on RX channel 2 | 0x0214 | RO |
| GDMA_IN_DSCR_BF1_CH2_REG | Address of the previous pre-read receive descriptor on RX channel 2 | 0x0218 | RO |
| GDMA_OUTFIFO_STATUS_CH2_REG | TX FIFO status of TX channel 2 | 0x0258 | RO |
| GDMA_OUT_STATE_CH2_REG | Transmit status of TX channel 2 | 0x0264 | RO |
| GDMA_OUT_EOF_DES_ADDR_CH2_REG | Outlink descriptor address when EOF occurs of TX channel 2 | 0x0268 | RO |
| GDMA_OUT_EOF_BFR_DES_ADDR_CH2_REG | The last outlink descriptor address when EOF occurs of TX channel 2 | 0x026C | RO |
| GDMA_OUT_DSCR_CH2_REG | Address of the next transmit descriptor pointed by the current pre-read transmit descriptor on TX channel 2 | 0x0270 | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| GDMA_OUT_DSCR_BF0_CH2_REG | Address of the current pre-read transmit descriptor on TX channel 2 | 0x0274 | RO |
| GDMA_OUT_DSCR_BF1_CH2_REG | Address of the previous pre-read transmit descriptor on TX channel 2 | 0x0278 | RO |
| **Priority Registers** | | | |
| GDMA_IN_PRI_CH0_REG | Priority register of RX channel 0 | 0x009C | R/W |
| GDMA_OUT_PRI_CH0_REG | Priority register of TX channel 0 | 0x00FC | R/W |
| GDMA_IN_PRI_CH1_REG | Priority register of RX channel 1 | 0x015C | R/W |
| GDMA_OUT_PRI_CH1_REG | Priority register of TX channel 1 | 0x01BC | R/W |
| GDMA_IN_PRI_CH2_REG | Priority register of RX channel 2 | 0x021C | R/W |
| GDMA_OUT_PRI_CH2_REG | Priority register of TX channel 2 | 0x027C | R/W |
| **Peripheral Select Registers** | | | |
| GDMA_IN_PERI_SEL_CH0_REG | Peripheral selection of RX channel 0 | 0x00A0 | R/W |
| GDMA_OUT_PERI_SEL_CH0_REG | Peripheral selection of TX channel 0 | 0x0100 | R/W |
| GDMA_IN_PERI_SEL_CH1_REG | Peripheral selection of RX channel 1 | 0x0160 | R/W |
| GDMA_OUT_PERI_SEL_CH1_REG | Peripheral selection of TX channel 1 | 0x01C0 | R/W |
| GDMA_IN_PERI_SEL_CH2_REG | Peripheral selection of RX channel 2 | 0x0220 | R/W |
| GDMA_OUT_PERI_SEL_CH2_REG | Peripheral selection of TX channel 2 | 0x0280 | R/W |

## 2.8 Registers

The addresses in this section are relative to GDMA base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 2.1. GDMA_INT_RAW_CH*n*_REG (*n*: 0-2) (0x0000+16**n*)**

| 31 | | | | | | | | | | | | | | | | | | | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | GDMA_OUTFIFO_UDF_CH0_INT_RAW | GDMA_OUTFIFO_OVF_CH0_INT_RAW | GDMA_INFIFO_UDF_CH0_INT_RAW | GDMA_INFIFO_OVF_CH0_INT_RAW | GDMA_OUT_TOTAL_EOF_CH0_INT_RAW | GDMA_IN_DSCR_EMPTY_CH0_INT_RAW | GDMA_OUT_DSCR_ERR_CH0_INT_RAW | GDMA_IN_DSCR_ERR_CH0_INT_RAW | GDMA_OUT_EOF_CH0_INT_RAW | GDMA_OUT_DONE_CH0_INT_RAW | GDMA_IN_ERR_EOF_CH0_INT_RAW | GDMA_IN_SUC_EOF_CH0_INT_RAW | GDMA_IN_DONE_CH0_INT_RAW | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**GDMA_IN_DONE_CH*n*_INT_RAW** The raw interrupt bit turns to high level when the last data pointed by one receive descriptor has been received for RX channel 0. (R/WTC/SS)

**GDMA_IN_SUC_EOF_CH*n*_INT_RAW** The raw interrupt bit turns to high level for RX channel 0 when the last data pointed by one receive descriptor has been received and the suc_eof bit in this descriptor is 1. For UHCI0, the raw interrupt bit turns to high level when the last data pointed by one receive descriptor has been received and no data error is detected for RX channel 0. (R/WTC/SS)

**GDMA_IN_ERR_EOF_CH*n*_INT_RAW** The raw interrupt bit turns to high level when data error is detected only in the case that the peripheral is UHCI0 for RX channel 0. For other peripherals, this raw interrupt is reserved. (R/WTC/SS)

**GDMA_OUT_DONE_CH*n*_INT_RAW** The raw interrupt bit turns to high level when the last data pointed by one transmit descriptor has been transmitted to peripherals for TX channel 0. (R/WTC/SS)

**GDMA_OUT_EOF_CH*n*_INT_RAW** The raw interrupt bit turns to high level when the last data pointed by one transmit descriptor has been read from memory for TX channel 0. (R/WTC/SS)

**GDMA_IN_DSCR_ERR_CH*n*_INT_RAW** The raw interrupt bit turns to high level when detecting receive descriptor error, including owner error, the second and third word error of receive descriptor for RX channel 0. (R/WTC/SS)

**GDMA_OUT_DSCR_ERR_CH*n*_INT_RAW** The raw interrupt bit turns to high level when detecting transmit descriptor error, including owner error, the second and third word error of transmit descriptor for TX channel 0. (R/WTC/SS)

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

**Register 2.1. GDMA_INT_RAW_CH*n*_REG (*n*: 0-2) (0x0000+16**n*)**

**Continued from the previous page...**

**GDMA_IN_DSCR_EMPTY_CH*n*_INT_RAW** The raw interrupt bit turns to high level when RX buffer pointed by inlink is full and receiving data is not completed, but there is no more inlink for RX channel 0. (R/WTC/SS)

**GDMA_OUT_TOTAL_EOF_CH*n*_INT_RAW** The raw interrupt bit turns to high level when data corresponding a outlink (includes one descriptor or few descriptors) is transmitted out for TX channel 0. (R/WTC/SS)

**GDMA_INFIFO_OVF_CH*n*_INT_RAW** This raw interrupt bit turns to high level when level 1 FIFO of RX channel 0 is overflow. (R/WTC/SS)

**GDMA_INFIFO_UDF_CH*n*_INT_RAW** This raw interrupt bit turns to high level when level 1 FIFO of RX channel 0 is underflow. (R/WTC/SS)

**GDMA_OUTFIFO_OVF_CH*n*_INT_RAW** This raw interrupt bit turns to high level when level 1 FIFO of TX channel 0 is overflow. (R/WTC/SS)

**GDMA_OUTFIFO_UDF_CH*n*_INT_RAW** This raw interrupt bit turns to high level when level 1 FIFO of TX channel 0 is underflow. (R/WTC/SS)

**Register 2.2. GDMA_INT_ST_CH*n*_REG (*n*: 0-2) (0x0004+16**n*)**

| 31 | | | | | | | | | | | | | | | | | | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | GDMA_OUTFIFO_UDF_CH0_INT_ST | GDMA_OUTFIFO_OVF_CH0_INT_ST | GDMA_INFIFO_UDF_CH0_INT_ST | GDMA_INFIFO_OVF_CH0_INT_ST | GDMA_OUT_TOTAL_EOF_CH0_INT_ST | GDMA_IN_DSCR_EMPTY_CH0_INT_ST | GDMA_OUT_DSCR_ERR_CH0_INT_ST | GDMA_IN_DSCR_ERR_CH0_INT_ST | GDMA_OUT_EOF_CH0_INT_ST | GDMA_OUT_DONE_CH0_INT_ST | GDMA_IN_ERR_EOF_CH0_INT_ST | GDMA_IN_SUC_EOF_CH0_INT_ST | GDMA_IN_DONE_CH0_INT_ST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**GDMA_IN_DONE_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_IN_DONE_CH_INT interrupt. (RO)

**GDMA_IN_SUC_EOF_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_IN_SUC_EOF_CH_INT interrupt. (RO)

**GDMA_IN_ERR_EOF_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_IN_ERR_EOF_CH_INT interrupt. (RO)

**GDMA_OUT_DONE_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_OUT_DONE_CH_INT interrupt. (RO)

**GDMA_OUT_EOF_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_OUT_EOF_CH_INT interrupt. (RO)

**GDMA_IN_DSCR_ERR_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_IN_DSCR_ERR_CH_INT interrupt. (RO)

**GDMA_OUT_DSCR_ERR_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_OUT_DSCR_ERR_CH_INT interrupt. (RO)

**GDMA_IN_DSCR_EMPTY_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_IN_DSCR_EMPTY_CH_INT interrupt. (RO)

**GDMA_OUT_TOTAL_EOF_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_OUT_TOTAL_EOF_CH_INT interrupt. (RO)

**GDMA_INFIFO_OVF_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_INFIFO_OVF_L1_CH_INT interrupt. (RO)

**GDMA_INFIFO_UDF_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_INFIFO_UDF_L1_CH_INT interrupt. (RO)

**GDMA_OUTFIFO_OVF_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_OUTFIFO_OVF_L1_CH_INT interrupt. (RO)

**GDMA_OUTFIFO_UDF_CH*n*_INT_ST** The raw interrupt status bit for the GDMA_OUTFIFO_UDF_L1_CH_INT interrupt. (RO)

**Register 2.3. GDMA_INT_ENA_CH*n*_REG (*n*: 0-2) (0x0008+16**n*)**

| 31 | | | | | | | | | | | | | | | | | | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | GDMA_OUTFIFO_UDF_CH0_INT_ENA | GDMA_OUTFIFO_OVF_CH0_INT_ENA | GDMA_INFIFO_UDF_CH0_INT_ENA | GDMA_INFIFO_OVF_CH0_INT_ENA | GDMA_OUT_TOTAL_EOF_CH0_INT_ENA | GDMA_IN_DSCR_EMPTY_CH0_INT_ENA | GDMA_OUT_DSCR_ERR_CH0_INT_ENA | GDMA_IN_DSCR_ERR_CH0_INT_ENA | GDMA_OUT_EOF_CH0_INT_ENA | GDMA_OUT_DONE_CH0_INT_ENA | GDMA_IN_ERR_EOF_CH0_INT_ENA | GDMA_IN_SUC_EOF_CH0_INT_ENA | GDMA_IN_DONE_CH0_INT_ENA | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**GDMA_IN_DONE_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_IN_DONE_CH_INT interrupt. (R/W)

**GDMA_IN_SUC_EOF_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_IN_SUC_EOF_CH_INT interrupt. (R/W)

**GDMA_IN_ERR_EOF_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_IN_ERR_EOF_CH_INT interrupt. (R/W)

**GDMA_OUT_DONE_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_OUT_DONE_CH_INT interrupt. (R/W)

**GDMA_OUT_EOF_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_OUT_EOF_CH_INT interrupt. (R/W)

**GDMA_IN_DSCR_ERR_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_IN_DSCR_ERR_CH_INT interrupt. (R/W)

**GDMA_OUT_DSCR_ERR_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_OUT_DSCR_ERR_CH_INT interrupt. (R/W)

**GDMA_IN_DSCR_EMPTY_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_IN_DSCR_EMPTY_CH_INT interrupt. (R/W)

**GDMA_OUT_TOTAL_EOF_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_OUT_TOTAL_EOF_CH_INT interrupt. (R/W)

**GDMA_INFIFO_OVF_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_INFIFO_OVF_L1_CH_INT interrupt. (R/W)

**GDMA_INFIFO_UDF_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_INFIFO_UDF_L1_CH_INT interrupt. (R/W)

**GDMA_OUTFIFO_OVF_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_OUTFIFO_OVF_L1_CH_INT interrupt. (R/W)

**GDMA_OUTFIFO_UDF_CH*n*_INT_ENA** The interrupt enable bit for the GDMA_OUTFIFO_UDF_L1_CH_INT interrupt. (R/W)

**Register 2.4. GDMA_INT_CLR_CH*n*_REG (*n*: 0-2) (0x000C+16**n*)**

| 31 | | | | | | | | | | | | | | | | | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | GDMA_OUTFIFO_UDF_CH0_INT_CLR | GDMA_OUTFIFO_OVF_CH0_INT_CLR | GDMA_INFIFO_UDF_CH0_INT_CLR | GDMA_INFIFO_OVF_CH0_INT_CLR | GDMA_OUT_TOTAL_EOF_CH0_INT_CLR | GDMA_IN_DSCR_EMPTY_CH0_INT_CLR | GDMA_OUT_DSCR_ERR_CH0_INT_CLR | GDMA_IN_DSCR_ERR_CH0_INT_CLR | GDMA_OUT_EOF_CH0_INT_CLR | GDMA_OUT_DONE_CH0_INT_CLR | GDMA_IN_ERR_EOF_CH0_INT_CLR | GDMA_IN_SUC_EOF_CH0_INT_CLR | GDMA_IN_DONE_CH0_INT_CLR | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**GDMA_IN_DONE_CH*n*_INT_CLR** Set this bit to clear the GDMA_IN_DONE_CH_INT interrupt. (WT)

**GDMA_IN_SUC_EOF_CH*n*_INT_CLR** Set this bit to clear the GDMA_IN_SUC_EOF_CH_INT interrupt. (WT)

**GDMA_IN_ERR_EOF_CH*n*_INT_CLR** Set this bit to clear the GDMA_IN_ERR_EOF_CH_INT interrupt. (WT)

**GDMA_OUT_DONE_CH*n*_INT_CLR** Set this bit to clear the GDMA_OUT_DONE_CH_INT interrupt. (WT)

**GDMA_OUT_EOF_CH*n*_INT_CLR** Set this bit to clear the GDMA_OUT_EOF_CH_INT interrupt. (WT)

**GDMA_IN_DSCR_ERR_CH*n*_INT_CLR** Set this bit to clear the GDMA_IN_DSCR_ERR_CH_INT interrupt. (WT)

**GDMA_OUT_DSCR_ERR_CH*n*_INT_CLR** Set this bit to clear the GDMA_OUT_DSCR_ERR_CH_INT interrupt. (WT)

**GDMA_IN_DSCR_EMPTY_CH*n*_INT_CLR** Set this bit to clear the GDMA_IN_DSCR_EMPTY_CH_INT interrupt. (WT)

**GDMA_OUT_TOTAL_EOF_CH*n*_INT_CLR** Set this bit to clear the GDMA_OUT_TOTAL_EOF_CH_INT interrupt. (WT)

**GDMA_INFIFO_OVF_CH*n*_INT_CLR** Set this bit to clear the GDMA_INFIFO_OVF_L1_CH_INT interrupt. (WT)

**GDMA_INFIFO_UDF_CH*n*_INT_CLR** Set this bit to clear the GDMA_INFIFO_UDF_L1_CH_INT interrupt. (WT)

**GDMA_OUTFIFO_OVF_CH*n*_INT_CLR** Set this bit to clear the GDMA_OUTFIFO_OVF_L1_CH_INT interrupt. (WT)

**GDMA_OUTFIFO_UDF_CH*n*_INT_CLR** Set this bit to clear the GDMA_OUTFIFO_UDF_L1_CH_INT interrupt. (WT)

**Register 2.5. GDMA_MISC_CONF_REG (0x0044)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | GDMA_CLK_EN | GDMA_ARB_PRI_DIS | (reserved) | GDMA_AHBM_RST_INTER | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**GDMA_AHBM_RST_INTER** Set this bit, then clear this bit to reset the internal ahb FSM. (R/W)

**GDMA_ARB_PRI_DIS** Set this bit to disable priority arbitration function. (R/W)

**GDMA_CLK_EN** 0: Enable the clock only when application writes registers. 1: Force the clock on for registers. (R/W)

**Register 2.6. GDMA_DATE_REG (0x0048)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GDMA_DATE | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x2008250 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Reset |

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

**Register 2.7. GDMA_IN_CONF0_CH*n*_REG (*n*: 0-2) (0x0070+192**n*)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | GDMA_MEM_TRANS_EN_CH0 | GDMA_IN_DATA_BURST_EN_CH0 | GDMA_INDSCR_BURST_EN_CH0 | GDMA_IN_LOOP_TEST_CH0 | GDMA_IN_RST_CH0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**GDMA_IN_RST_CH*n*** This bit is used to reset GDMA channel 0 RX FSM and RX FIFO pointer. (R/W)

**GDMA_IN_LOOP_TEST_CH*n*** This bit is used to fill the owner bit of receive descriptor by hardware of receive descriptor. (R/W)

**GDMA_INDSCR_BURST_EN_CH*n*** Set this bit to 1 to enable INCR burst transfer for RX channel 0 reading descriptor when accessing internal RAM. (R/W)

**GDMA_IN_DATA_BURST_EN_CH*n*** Set this bit to 1 to enable INCR burst transfer for RX channel 0 receiving data when accessing internal RAM. (R/W)

**GDMA_MEM_TRANS_EN_CH*n*** Set this bit 1 to enable automatic transmitting data from memory to memory via GDMA. (R/W)

**Register 2.8. GDMA_IN_CONF1_CH*n*_REG (*n*: 0-2) (0x0074+192**n*)**

| 31 | | | | | | | | | | | | | | | | | | 13 | 12 | 11 | | | | | | | | | | | | | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | GDMA_IN_CHECK_OWNER_CH0 | (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 | 0 | Reset |

**GDMA_IN_CHECK_OWNER_CH*n*** Set this bit to enable checking the owner attribute of the descriptor. (R/W)

**Register 2.9. GDMA_IN_POP_CH*n*_REG (*n*: 0-2) (0x007C+192**n*)**

| (reserved) | GDMA_INFIFO_POP_CH0 | GDMA_INFIFO_RDATA_CH0 | |
|---|---|---|---|
| 31 … 13 | 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0x800 | Reset |

**GDMA_INFIFO_RDATA_CH*n*** This register stores the data popping from GDMA FIFO (intended for debugging). (RO)

**GDMA_INFIFO_POP_CH*n*** Set this bit to pop data from GDMA FIFO (intended for debugging). (R/W/SC)

**Register 2.10. GDMA_IN_LINK_CH*n*_REG (*n*: 0-2) (0x0080+192**n*)**

| (reserved) | GDMA_INLINK_PARK_CH0 | GDMA_INLINK_RESTART_CH0 | GDMA_INLINK_START_CH0 | GDMA_INLINK_STOP_CH0 | GDMA_INLINK_AUTO_RET_CH0 | GDMA_INLINK_ADDR_CH0 | |
|---|---|---|---|---|---|---|---|
| 31 … 25 | 24 | 23 | 22 | 21 | 20 | 19 … 0 | |
| 0 0 0 0 0 0 0 | 1 | 0 | 0 | 0 | 1 | 0x000 | Reset |

**GDMA_INLINK_ADDR_CH*n*** This register stores the 20 least significant bits of the first receive descriptor's address. (R/W)

**GDMA_INLINK_AUTO_RET_CH*n*** Set this bit to return to current receive descriptor's address, when there are some errors in current receiving data. (R/W)

**GDMA_INLINK_STOP_CH*n*** Set this bit to stop GDMA's receive channel from receiving data. (R/W/SC)

**GDMA_INLINK_START_CH*n*** Set this bit to enable GDMA's receive channel from receiving data. (R/W/SC)

**GDMA_INLINK_RESTART_CH*n*** Set this bit to mount a new receive descriptor. (R/W/SC)

**GDMA_INLINK_PARK_CH*n*** 1: the receive descriptor's FSM is in idle state; 0: the receive descriptor's FSM is working. (RO)

**Register 2.11. GDMA_OUT_CONF0_CH*n*_REG (*n*: 0-2) (0x00D0+192**n*)**

| (reserved) | GDMA_OUT_DATA_BURST_EN_CH0 | GDMA_OUTDSCR_BURST_EN_CH0 | GDMA_OUT_EOF_MODE_CH0 | GDMA_OUT_AUTO_WRBACK_CH0 | GDMA_OUT_LOOP_TEST_CH0 | GDMA_OUT_RST_CH0 | |
|---|---|---|---|---|---|---|---|
| 31 … 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 | 1 | 0 | 0 | 0 | Reset |

**GDMA_OUT_RST_CH*n*** This bit is used to reset GDMA channel 0 TX FSM and TX FIFO pointer. (R/W)

**GDMA_OUT_LOOP_TEST_CH*n*** Reserved. (R/W)

**GDMA_OUT_AUTO_WRBACK_CH*n*** Set this bit to enable automatic outlink-writeback when all the data in TX buffer has been transmitted. (R/W)

**GDMA_OUT_EOF_MODE_CH*n*** EOF flag generation mode when transmitting data. 1: EOF flag for TX channel 0 is generated when data need to transmit has been popped from FIFO in GDMA. (R/W)

**GDMA_OUTDSCR_BURST_EN_CH*n*** Set this bit to 1 to enable INCR burst transfer for TX channel 0 reading descriptor when accessing internal RAM. (R/W)

**GDMA_OUT_DATA_BURST_EN_CH*n*** Set this bit to 1 to enable INCR burst transfer for TX channel 0 transmitting data when accessing internal RAM. (R/W)

**Register 2.12. GDMA_OUT_CONF1_CH*n*_REG (*n*: 0-2) (0x00D4+192**n*)**

| (reserved) | GDMA_OUT_CHECK_OWNER_CH0 | (reserved) | |
|---|---|---|---|
| 31 … 13 | 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 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 |

**GDMA_OUT_CHECK_OWNER_CH*n*** Set this bit to enable checking the owner attribute of the descriptor. (R/W)

**Register 2.13. GDMA_OUT_PUSH_CH*n*_REG (*n*: 0-2) (0x00DC+192**n*)**

| (reserved) | GDMA_OUTFIFO_PUSH_CH0 | GDMA_OUTFIFO_WDATA_CH0 | |
|---|---|---|---|
| 31 … 10 | 9 | 8 … 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 |

**GDMA_OUTFIFO_WDATA_CH*n*** This register stores the data that need to be pushed into GDMA FIFO. (R/W)

**GDMA_OUTFIFO_PUSH_CH*n*** Set this bit to push data into GDMA FIFO. (R/W/SC)

**Register 2.14. GDMA_OUT_LINK_CH*n*_REG (*n*: 0-2) (0x00E0+192**n*)**

| (reserved) | GDMA_OUTLINK_PARK_CH0 | GDMA_OUTLINK_RESTART_CH0 | GDMA_OUTLINK_START_CH0 | GDMA_OUTLINK_STOP_CH0 | GDMA_OUTLINK_ADDR_CH0 | |
|---|---|---|---|---|---|---|
| 31 … 24 | 23 | 22 | 21 | 20 | 19 … 0 | |
| 0 0 0 0 0 0 0 0 | 1 | 0 | 0 | 0 | 0x000 | Reset |

**GDMA_OUTLINK_ADDR_CH*n*** This register stores the 20 least significant bits of the first transmit descriptor's address. (R/W)

**GDMA_OUTLINK_STOP_CH*n*** Set this bit to stop GDMA's transmit channel from transferring data. (R/W/SC)

**GDMA_OUTLINK_START_CH*n*** Set this bit to enable GDMA's transmit channel for data transfer. (R/W/SC)

**GDMA_OUTLINK_RESTART_CH*n*** Set this bit to restart a new outlink from the last address. (R/W/SC)

**GDMA_OUTLINK_PARK_CH*n*** 1: the transmit descriptor's FSM is in idle state; 0: the transmit descriptor's FSM is working. (RO)

**Register 2.15. GDMA_INFIFO_STATUS_CH*n*_REG (*n*: 0-2) (0x0078+192**n*)**

| (reserved) | GDMA_IN_BUF_HUNGRY_CH0 | GDMA_IN_REMAIN_UNDER_4B_CH0 | GDMA_IN_REMAIN_UNDER_3B_CH0 | GDMA_IN_REMAIN_UNDER_2B_CH0 | GDMA_IN_REMAIN_UNDER_1B_CH0 | (reserved) | GDMA_INFIFO_CNT_CH0 | GDMA_INFIFO_EMPTY_CH0 | GDMA_INFIFO_FULL_CH0 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 28 | 27 | 26 | 25 | 24 | 23 | 22 8 | 7 2 | 1 | 0 | |
| 0 0 0 0 | 0 | 1 | 1 | 1 | 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 1 | 1 | Reset |

**GDMA_INFIFO_FULL_CH*n*** L1 RX FIFO full signal for RX channel 0. (RO)

**GDMA_INFIFO_EMPTY_CH*n*** L1 RX FIFO empty signal for RX channel 0. (RO)

**GDMA_INFIFO_CNT_CH*n*** The register stores the byte number of the data in L1 RX FIFO for RX channel 0. (RO)

**GDMA_IN_REMAIN_UNDER_1B_CH*n*** Reserved. (RO)

**GDMA_IN_REMAIN_UNDER_2B_CH*n*** Reserved. (RO)

**GDMA_IN_REMAIN_UNDER_3B_CH*n*** Reserved. (RO)

**GDMA_IN_REMAIN_UNDER_4B_CH*n*** Reserved. (RO)

**GDMA_IN_BUF_HUNGRY_CH*n*** Reserved. (RO)

**Register 2.16. GDMA_IN_STATE_CH*n*_REG (*n*: 0-2) (0x0084+192**n*)**

| (reserved) | GDMA_IN_STATE_CH0 | GDMA_IN_DSCR_STATE_CH0 | GDMA_INLINK_DSCR_ADDR_CH0 | |
|---|---|---|---|---|
| 31 23 | 22 20 | 19 18 | 17 0 | |
| 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | Reset |

**GDMA_INLINK_DSCR_ADDR_CH*n*** This register stores the lower 18 bits of the next receive descriptor address that is pre-read (but not processed yet). If the current receive descriptor is the last descriptor, then this field represents the address of the current receive descriptor. (RO)

**GDMA_IN_DSCR_STATE_CH*n*** Reserved. (RO)

**GDMA_IN_STATE_CH*n*** Reserved. (RO)

**Register 2.17. GDMA_IN_SUC_EOF_DES_ADDR_CH*n*_REG (*n*: 0-2) (0x0088+192**n*)**

| GDMA_IN_SUC_EOF_DES_ADDR_CH0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**GDMA_IN_SUC_EOF_DES_ADDR_CH*n*** This register stores the address of the receive descriptor when the EOF bit in this descriptor is 1. (RO)

**Register 2.18. GDMA_IN_ERR_EOF_DES_ADDR_CH*n*_REG (*n*: 0-2) (0x008C+192**n*)**

| GDMA_IN_ERR_EOF_DES_ADDR_CH0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**GDMA_IN_ERR_EOF_DES_ADDR_CH*n*** This register stores the address of the receive descriptor when there are some errors in current receiving data. Only used when peripheral is UHCI0. (RO)

**Register 2.19. GDMA_IN_DSCR_CH*n*_REG (*n*: 0-2) (0x0090+192**n*)**

| GDMA_INLINK_DSCR_CH0 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**GDMA_INLINK_DSCR_CH*n*** Represents the address of the next receive descriptor x+1 pointed by the current receive descriptor that is pre-read. (RO)

**Register 2.20. GDMA_IN_DSCR_BF0_CH*n*_REG (*n*: 0-2) (0x0094+192**n*)**

| GDMA_INLINK_DSCR_BF0_CH0 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**GDMA_INLINK_DSCR_BF0_CH*n*** Represents the address of the current receive descriptor x that is pre-read. (RO)

**Register 2.21. GDMA_IN_DSCR_BF1_CH*n*_REG (*n*: 0-2) (0x0098+192**n*)**

| GDMA_INLINK_DSCR_BF1_CH0 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**GDMA_INLINK_DSCR_BF1_CH*n*** Represents the address of the previous receive descriptor x-1 that is pre-read. (RO)

**Register 2.22. GDMA_OUTFIFO_STATUS_CH*n*_REG (*n*: 0-2) (0x00D8+192**n*)**

| (reserved) | GDMA_OUT_REMAIN_UNDER_4B_CH0 | GDMA_OUT_REMAIN_UNDER_3B_CH0 | GDMA_OUT_REMAIN_UNDER_2B_CH0 | GDMA_OUT_REMAIN_UNDER_1B_CH0 | (reserved) | GDMA_OUTFIFO_CNT_CH0 | GDMA_OUTFIFO_EMPTY_CH0 | GDMA_OUTFIFO_FULL_CH0 | |
|---|---|---|---|---|---|---|---|---|---|
| 31 27 | 26 | 25 | 24 | 23 | 22 8 | 7 2 | 1 | 0 | |
| 0 0 0 0 0 | 1 | 1 | 1 | 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 1 | 0 | Reset |

**GDMA_OUTFIFO_FULL_CH*n*** L1 TX FIFO full signal for TX channel 0. (RO)

**GDMA_OUTFIFO_EMPTY_CH*n*** L1 TX FIFO empty signal for TX channel 0. (RO)

**GDMA_OUTFIFO_CNT_CH*n*** The register stores the byte number of the data in L1 TX FIFO for TX channel 0. (RO)

**GDMA_OUT_REMAIN_UNDER_1B_CH*n*** Reserved. (RO)

**GDMA_OUT_REMAIN_UNDER_2B_CH*n*** Reserved. (RO)

**GDMA_OUT_REMAIN_UNDER_3B_CH*n*** Reserved. (RO)

**GDMA_OUT_REMAIN_UNDER_4B_CH*n*** Reserved. (RO)

**Register 2.23. GDMA_OUT_STATE_CH*n*_REG (*n*: 0-2) (0x00E4+192**n*)**

| (reserved) | GDMA_OUT_STATE_CH0 | GDMA_OUT_DSCR_STATE_CH0 | GDMA_OUTLINK_DSCR_ADDR_CH0 | |
|---|---|---|---|---|
| 31 23 | 22 20 | 19 18 | 17 0 | |
| 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | Reset |

**GDMA_OUTLINK_DSCR_ADDR_CH*n*** This register stores the lower 18 bits of the next receive descriptor address that is pre-read (but not processed yet). If the current receive descriptor is the last descriptor, then this field represents the address of the current receive descriptor. (RO)

**GDMA_OUT_DSCR_STATE_CH*n*** Reserved. (RO)

**GDMA_OUT_STATE_CH*n*** Reserved. (RO)

**Register 2.24. GDMA_OUT_EOF_DES_ADDR_CH*n*_REG (*n*: 0-2) (0x00E8+192**n*)**

| 31 | 0 | |
|---|---|---|
| GDMA_OUT_EOF_DES_ADDR_CH0 | | |
| 0x000000 | | Reset |

**GDMA_OUT_EOF_DES_ADDR_CH*n*** This register stores the address of the transmit descriptor when the EOF bit in this descriptor is 1. (RO)

**Register 2.25. GDMA_OUT_EOF_BFR_DES_ADDR_CH*n*_REG (*n*: 0-2) (0x00EC+192**n*)**

| 31 | 0 | |
|---|---|---|
| GDMA_OUT_EOF_BFR_DES_ADDR_CH0 | | |
| 0x000000 | | Reset |

**GDMA_OUT_EOF_BFR_DES_ADDR_CH*n*** This register stores the address of the transmit descriptor before the last transmit descriptor. (RO)

**Register 2.26. GDMA_OUT_DSCR_CH*n*_REG (*n*: 0-2) (0x00F0+192**n*)**

| 31 | 0 | |
|---|---|---|
| GDMA_OUTLINK_DSCR_CH0 | | |
| 0 | | Reset |

**GDMA_OUTLINK_DSCR_CH*n*** Represents the address of the next transmit descriptor y+1 pointed by the current transmit descriptor that is pre-read. (RO)

**Register 2.27. GDMA_OUT_DSCR_BF0_CH*n*_REG (*n*: 0-2) (0x00F4+192**n*)**

| 31 … 0 | |
|---|---|
| GDMA_OUTLINK_DSCR_BF0_CH0 | |
| 0 | Reset |

**GDMA_OUTLINK_DSCR_BF0_CH*n*** Represents the address of the current transmit descriptor y that is pre-read. (RO)

**Register 2.28. GDMA_OUT_DSCR_BF1_CH*n*_REG (*n*: 0-2) (0x00F8+192**n*)**

| 31 … 0 | |
|---|---|
| GDMA_OUTLINK_DSCR_BF1_CH0 | |
| 0 | Reset |

**GDMA_OUTLINK_DSCR_BF1_CH*n*** Represents the address of the previous transmit descriptor y-1 that is pre-read. (RO)

**Register 2.29. GDMA_IN_PRI_CH*n*_REG (*n*: 0-2) (0x009C+192**n*)**

| 31 … 4 | 3 … 0 | |
|---|---|---|
| (reserved) | GDMA_RX_PRI_CH0 | |
| 0 0 0 0 0 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 |

**GDMA_RX_PRI_CH*n*** The priority of RX channel 0. The larger the value, the higher the priority. (R/W)

**Register 2.30. GDMA_OUT_PRI_CH*n*_REG (*n*: 0-2) (0x00FC+192**n*)**

| (reserved) | GDMA_TX_PRI_CH0 | |
|---|---|---|
| 31 … 4 | 3 … 0 | |
| 0 0 0 0 0 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 |

**GDMA_TX_PRI_CH*n*** The priority of TX channel 0. The larger the value, the higher the priority. (R/W)

**Register 2.31. GDMA_IN_PERI_SEL_CH*n*_REG (*n*: 0-2) (0x00A0+192**n*)**

| (reserved) | GDMA_PERI_IN_SEL_CH0 | |
|---|---|---|
| 31 … 6 | 5 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x3f | Reset |

**GDMA_PERI_IN_SEL_CH*n*** This register is used to select peripheral for RX channel 0. 0: SPI2. 1: reserved. 2: UHCI0. 3: I2S. 4: reserved. 5: reserved. 6: AES. 7: SHA. 8: ADC; 9 ~ 63: Invalid. (R/W)

**Register 2.32. GDMA_OUT_PERI_SEL_CH*n*_REG (*n*: 0-2) (0x0100+192**n*)**

| (reserved) | GDMA_PERI_OUT_SEL_CH0 | |
|---|---|---|
| 31 … 6 | 5 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x3f | Reset |

**GDMA_PERI_OUT_SEL_CH*n*** This register is used to select peripheral for TX channel 0. 0: SPI2. 1: reserved. 2: UHCI0. 3: I2S. 4: reserved. 5: reserved. 6: AES. 7: SHA. 8: ADC; 9 ~ 63: Invalid. (R/W)

# Part II

# Memory Organization

This part provides insights into the system's memory structure, discussing the organization and mapping of RAM, ROM, eFuse, and external memories, offering a framework for understanding memory-related subsystems.

# Chapter 3

# System and Memory

## 3.1 Overview

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

## 3.2 Features

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

Figure 3.2-1 illustrates the system structure and address mapping.

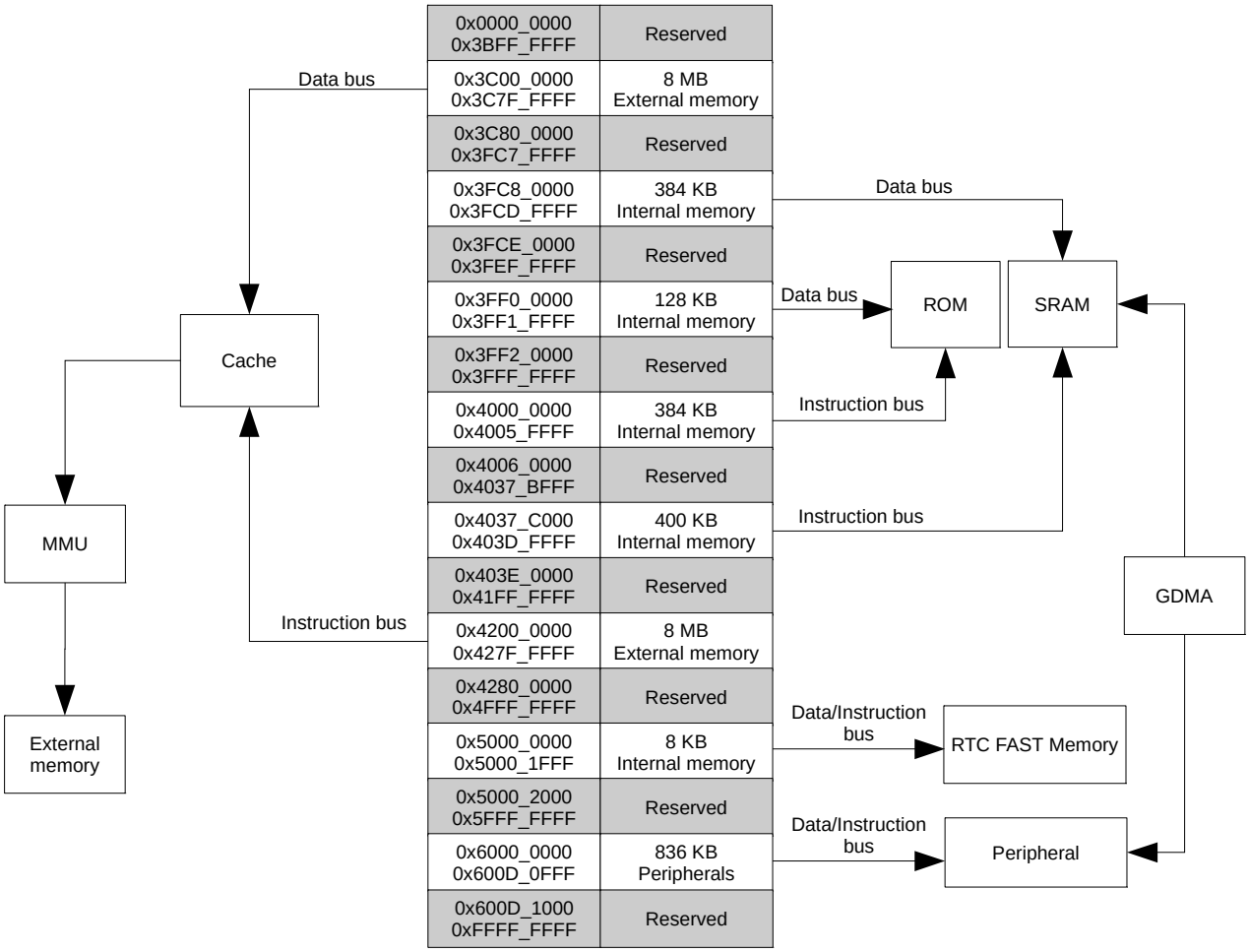



Figure 3.2-1. System Structure and Address Mapping

**Note:**

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

# 3.3 Functional Description

## 3.3.1 Address Mapping

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

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

The CPU can:

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

Figure 3.2-1 lists the address ranges on the data bus and instruction bus and their corresponding target memory.

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

### 3.3.2 Internal Memory

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

- Internal ROM (384 KB): The Internal ROM of the ESP32-C3 is a Mask ROM, meaning it is strictly read-only and cannot be reprogrammed. Internal ROM contains the ROM code (software instructions and some software read-only data) of some low level system software.
- Internal SRAM (400 KB): The Internal Static RAM (SRAM) is a volatile memory that can be quickly accessed by the CPU (generally within a single CPU clock cycle).
  - A part of the SRAM can be configured to operate as a cache for external memory access.
  - Some parts of the SRAM can only be accessed via the CPU's instruction bus.
  - Some parts of the SRAM can be accessed via both the CPU's instruction bus and the CPU's data bus.
- RTC Memory (8 KB): The RTC (Real Time Clock) memory implemented as Static RAM (SRAM) thus is volatile. However, RTC memory has the added feature of being persistent in deep sleep (i.e., the RTC memory retains its values throughout deep sleep).
  - RTC FAST Memory (8 KB): RTC FAST memory can only be accessed by the CPU and can be generally used to store instructions and data that needs to persist across a deep sleep.

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

However, within each segment, there may be different bus access restrictions (e.g., some parts of the segment may only be accessible by the CPU's Data bus). Therefore, each some segments are also further divided into parts. Table 3.3-1 describes each part of internal memory and their address ranges on the data bus and/or instruction bus.

**Table 3.3-1. Internal Memory Address Mapping**

| Bus Type | Boundary Address | | Size (KB) | Target |
|---|---|---|---|---|
| | Low Address | High Address | | |
| Data bus | 0x3FF0_0000 | 0x3FF1_FFFF | 128 | Internal ROM 1 |
| | 0x3FC8_0000 | 0x3FCD_FFFF | 384 | Internal SRAM 1 |
| Instruction bus | 0x4000_0000 | 0x4003_FFFF | 256 | Internal ROM 0 |
| | 0x4004_0000 | 0x4005_FFFF | 128 | Internal ROM 1 |
| | 0x4037_C000 | 0x4037_FFFF | 16 | Internal SRAM 0 |

Cont'd on next page

Table 3.3-1 – cont'd from previous page

| Bus Type | Boundary Address | | Size (KB) | Target |
|---|---|---|---|---|
| | Low Address | High Address | | |
| | 0x4038_0000 | 0x403D_FFFF | 384 | Internal SRAM 1 |
| Data/Instruction bus | 0x5000_0000 | 0x5000_1FFF | 8 | RTC FAST Memory |

**Note:**

All of the internal memories are managed by Permission Control module. An internal memory can only be accessed when it is allowed by Permission Control, then the internal memory can be available to the CPU. For more information about Permission Control, please refer to Chapter 14 *Permission Control (PMS)*.

**1. Internal ROM 0**

Internal ROM 0 is a 256 KB, read-only memory space, addressed by the CPU only through the instruction bus via 0x4000_0000 ~ 0x4003_FFFF, as shown in Table 3.3-1.

**2. Internal ROM 1**

Internal ROM 1 is a 128 KB, read-only memory space, addressed by the CPU through the instruction bus via 0x4004_0000 ~ 0x4005_FFFF or through the data bus via 0x3FF0_0000 ~ 0x3FF1_FFFF in the same order, as shown in Table 3.3-1.

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

**3. Internal SRAM 0**

Internal SRAM 0 is a 16 KB, read-and-write memory space, addressed by the CPU through the instruction bus via the range described in Table 3.3-1.

This memory managed by Permission Control, can be configured as instruction cache to store cache instructions or read-only data of the external memory. In this case, the memory cannot be accessed by the CPU. For more information about Permission Control, please refer to Chapter 14 *Permission Control (PMS)*.

**4. Internal SRAM 1**

Internal SRAM 1 is a 384 KB, read-and-write memory space, addressed by the CPU through the data bus or instruction bus, in the same order, via the ranges described in Table 3.3-1.

**5. RTC FAST Memory**

RTC FAST Memory is a 8 KB, read-and-write SRAM, addressed by the CPU through the data/instruction bus via the shared address 0x5000_0000 ~ 0x5000_1FFF, as described in Table 3.3-1.

### 3.3.3 External Memory

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

#### 3.3.3.1 External Memory Address Mapping

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

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

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

Table 3.3-2 lists the mapping between the cache and the corresponding address ranges on the data bus and instruction bus.

Table 3.3-2. External Memory Address Mapping

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

**Note:**
Only if the CPU obtains permission for accessing the external memory, can it be responded for memory access. For more detailed information about permission control, please refer to Chapter 14 *Permission Control (PMS)*.

#### 3.3.3.2 Cache

As shown in Figure 3.3-1, ESP32-C3 has a read-only uniform cache which is eight-way set-associative, its size is 16 KB and its block size is 32 bytes. When cache is active, some internal memory space will be occupied by cache (see Internal SRAM 0 in Section 3.3.2).

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

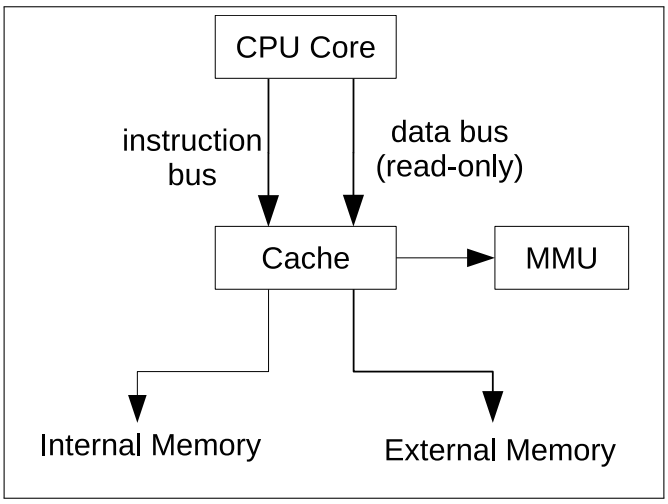



Figure 3.3-1. Cache Structure

#### 3.3.3.3 Cache Operations

ESP32-C3 cache support the following operations:

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

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

### 3.3.4 GDMA Address Space

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

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

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

There are 7 peripherals/modules that can work together with GDMA.

As shown in Figure 3.3-2, these 7 vertical lines in turn correspond to these 7 peripherals/modules with GDMA function, the horizontal line represents a certain channel of GDMA (can be any channel), and the intersection of the vertical line and the horizontal line indicates that a peripheral/module has the ability to access the corresponding channel of GDMA. If there are multiple intersections on the same line, it means that these peripherals/modules cannot enable the GDMA function at the same time.

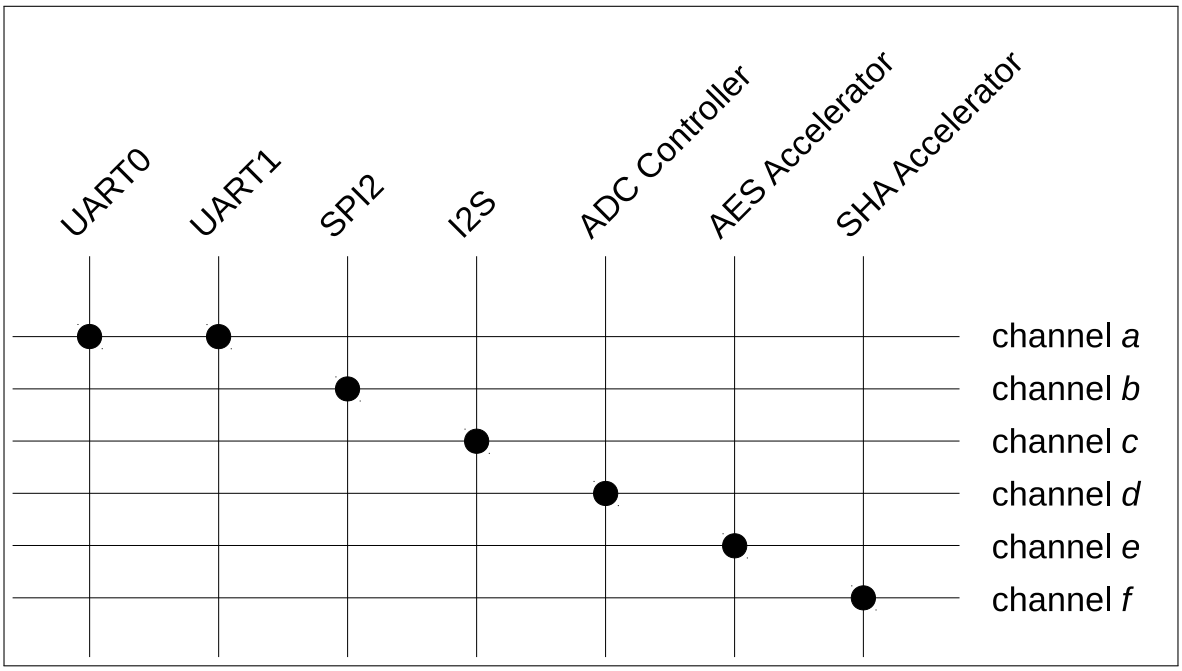



**Figure 3.3-2. Peripherals/modules that can work with GDMA**

These peripherals/modules can access any memory available to GDMA. For more information, please refer to Chapter 2 *GDMA Controller (GDMA)*.

> **Note:**
> When accessing a memory via GDMA, a corresponding access permission is needed, otherwise this access may fail. For more information about permission control, please refer to Chapter 14 *Permission Control (PMS)*.

### 3.3.5 Modules/Peripherals

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

### 3.3.5.1 Module/Peripheral Address Mapping

Table 3.3-3 lists all the modules/peripherals and their respective address ranges. Note that the address space of specific modules/peripherals is defined by "Boundary Address" (including both Low Address and High Address).

Table 3.3-3. Module/Peripheral Address Mapping

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

Cont'd on next page

Table 3.3-3 – cont'd from previous page

| Target | Boundary Address | | Size (KB) | Notes |
|---|---|---|---|---|
| | Low Address | High Address | | |
| Digital Signature | 0x6003_D000 | 0x6003_DFFF | 4 | |
| HMAC Accelerator | 0x6003_E000 | 0x6003_EFFF | 4 | |
| GDMA Controller | 0x6003_F000 | 0x6003_FFFF | 4 | |
| ADC Controller | 0x6004_0000 | 0x6004_0FFF | 4 | |
| Reserved | 0x6004_1000 | 0x6002_FFFF | | |
| USB Serial/JTAG Controller | 0x6004_3000 | 0x6004_3FFF | 4 | |
| Reserved | 0x6004_4000 | 0x600B_FFFF | | |
| System Registers | 0x600C_0000 | 0x600C_0FFF | 4 | |
| PMS Registers | 0x600C_1000 | 0x600C_1FFF | 4 | |
| Interrupt Matrix | 0x600C_2000 | 0x600C_2FFF | 4 | |
| Reserved | 0x600C_3000 | 0x600C_3FFF | | |
| Reserved | 0x600C_4000 | 0x600C_BFFF | | |
| External Memory Encryption and Decryption | 0x600C_C000 | 0x600C_CFFF | 4 | |
| Reserved | 0x600C_D000 | 0x600C_DFFF | | |
| Assist Debug | 0x600C_E000 | 0x600C_EFFF | 4 | |
| Reserved | 0x600C_F000 | 0x600C_FFFF | | |
| World Controller | 0x600D_0000 | 0x600D_0FFF | 4 | |

# Chapter 4

# eFuse Controller (EFUSE)

## 4.1 Overview

ESP32-C3 contains a 4096-bit eFuse controller to store parameters. Once an eFuse bit is programmed to 1, it can never be reverted to 0. The eFuse controller programs individual bits of parameters in eFuse according to user configurations. From outside the chip, eFuse data can only be read via the eFuse Controller. If read-protection for some data is not enabled, that data is readable from outside the chip. If read-protection is enabled, that data can not be read from outside the chip. In all cases, however, some keys stored in eFuse can still be used internally by hardware cryptography modules such as Digital Signature, HMAC, etc., without exposing this data to the outside world.

## 4.2 Features

- 4096-bit One-time programmable storage
- Configurable write protection
- Configurable read protection
- Various hardware encoding schemes against data corruption

## 4.3 Functional Description

### 4.3.1 Structure

eFuse data is organized in 11 blocks (BLOCK0 ~ BLOCK10).

BLOCK0, which holds most parameters, has 9 bits that are readable but useless to users, and 60 further bits are reserved for future use.

Table 4.3-1 lists all the parameters accessible (readable and usable) to users in BLOCK0 and their offsets, bit widths, as well as information on whether their configuration is directly accessible by hardware, and whether they are protected from programming.

The **EFUSE_WR_DIS** parameter is used to disable the writing of other parameters, while **EFUSE_RD_DIS** is used to disable users from reading BLOCK4 ~ BLOCK10. For more information on these two parameters, please see Section 4.3.1.1 and Section 4.3.1.2.

Table 4.3-1. Parameters in eFuse BLOCK0

| Parameters | Bit Width | Accessible by Hardware | Programming-Protection by EFUSE_WR_DIS Bit Number | Description |
|---|---|---|---|---|
| **EFUSE_WR_DIS** | 32 | Y | N/A | Represents whether writing of individual eFuses is disabled. |
| **EFUSE_RD_DIS** | 7 | Y | 0 | Represents whether users' reading from BLOCK4 ~ 10 is disabled. |
| EFUSE_DIS_ICACHE | 1 | Y | 2 | Represents whether iCache is disabled. |
| EFUSE_DIS_USB_JTAG | 1 | Y | 2 | Represents whether the USB-to-JTAG function is disabled. |
| EFUSE_DIS_DOWNLOAD_ICACHE | 1 | Y | 2 | Represents whether iCache is disabled in Download mode. |
| EFUSE_DIS_USB_SERIAL_JTAG | 1 | Y | 2 | Represents whether the usb_serial_jtag peripheral is disabled. |
| EFUSE_DIS_FORCE_DOWNLOAD | 1 | Y | 2 | Represents whether the function to force the chip into Download mode is disabled. |
| EFUSE_DIS_TWAI | 1 | Y | 2 | Represents whether the TWAI controller is disabled. |
| EFUSE_JTAG_SEL_ENABLE | 1 | Y | 2 | Represents whether to use JTAG directly. |
| EFUSE_SOFT_DIS_JTAG | 3 | Y | 31 | Represents whether JTAG is disabled in the soft way. |
| EFUSE_DIS_PAD_JTAG | 1 | Y | 2 | Represents whether JTAG is disabled in the hard way (permanently). |
| EFUSE_DIS_DOWNLOAD_ MANUAL_ENCRYPT | 1 | Y | 2 | Represents whether flash encryption is disabled in Download boot mode. |
| EFUSE_USB_EXCHG_PINS | 1 | Y | 30 | Represents whether the D+ and D- pins are exchanged. |
| EFUSE_VDD_SPI_AS_GPIO | 1 | N | 30 | Represents whether the VDD_SPI pin is used as a regular GPIO. |
| EFUSE_WDT_DELAY_SEL | 2 | Y | 3 | Represents whether RTC watchdog timeout threshold is selected. |
| EFUSE_SPI_BOOT_CRYPT_CNT | 3 | Y | 4 | Represents whether SPI boot encryption/decryption is enabled. |

Cont'd on next page

Table 4.3-1 – cont'd from previous page

| Parameters | Bit Width | Accessible by Hardware | Programming-Protection by EFUSE_WR_DIS Bit Number | Description |
|---|---|---|---|---|
| EFUSE_SECURE_BOOT_KEY_ REVOKE0 | 1 | N | 5 | Represents whether revoking the first Secure Boot key is enabled. |
| EFUSE_SECURE_BOOT_KEY_ REVOKE1 | 1 | N | 6 | Represents whether revoking the second Secure Boot key is enabled.. |
| EFUSE_SECURE_BOOT_KEY_ REVOKE2 | 1 | N | 7 | Represents whether revoking the third Secure Boot key is enabled. |
| EFUSE_KEY_PURPOSE_0 | 4 | Y | 8 | Represents Key0 purpose, see Table 4.3-2. |
| EFUSE_KEY_PURPOSE_1 | 4 | Y | 9 | Represents Key1 purpose, see Table 4.3-2. |
| EFUSE_KEY_PURPOSE_2 | 4 | Y | 10 | Represents Key2 purpose, see Table 4.3-2. |
| EFUSE_KEY_PURPOSE_3 | 4 | Y | 11 | Represents Key3 purpose, see Table 4.3-2. |
| EFUSE_KEY_PURPOSE_4 | 4 | Y | 12 | Represents Key4 purpose, see Table 4.3-2. |
| EFUSE_KEY_PURPOSE_5 | 4 | Y | 13 | Represents Key5 purpose, see Table 4.3-2. |
| EFUSE_SECURE_BOOT_EN | 1 | N | 15 | Represents whether Secure Boot is enabled. |
| EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE | 1 | N | 16 | Represents whether aggressive revocation of Secure Boot is enabled. |
| EFUSE_FLASH_TPUW | 4 | N | 18 | Represents the flash waiting time after power-up. |
| EFUSE_DIS_DOWNLOAD_MODE | 1 | N | 18 | Represents whether all download modes are disabled. |
| EFUSE_USB_PRINT_CHANNEL | 1 | N | 18 | Represents whether USB printing is disabled. |
| EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE | 1 | N | 18 | Represents whether the USB-Serial-JTAG download function is disabled. |
| EFUSE_ENABLE_SECURITY_DOWNLOAD | 1 | N | 18 | Represents whether UART secure download mode is enabled. |
| EFUSE_UART_PRINT_CONTROL | 2 | N | 18 | Represents the UART boot message output mode. |
| EFUSE_FORCE_SEND_RESUME | 1 | N | 18 | Represents whether ROM code is forced to send a resume command during SPI boot. |

Cont'd on next page

Table 4.3-1 – cont'd from previous page

| Parameters | Bit Width | Accessible by Hardware | Programming-Protection by EFUSE_WR_DIS Bit Number | Description |
|---|---|---|---|---|
| EFUSE_SECURE_VERSION | 16 | N | 18 | Represents the version used by ESP-IDF anti-rollback feature. |
| EFUSE_ERR_RST_ENABLE | 1 | N | 19 | Represents whether to use BLOCK0 to check error record registers. |

Table 4.3-2 lists all key purpose and their values. Setting the eFuse parameter EFUSE_KEY_PURPOSE_*n* declares the purpose of KEY*n* (*n*: 0 ~ 5).

**Table 4.3-2. Secure Key Purpose Values**

| Key Purpose Values | Purposes |
|---|---|
| 0 | User purposes |
| 1 | Reserved |
| 2 | Reserved |
| 3 | Reserved |
| 4 | XTS_AES_128_KEY (flash/SRAM encryption and decryption) |
| 5 | HMAC Downstream mode (both JTAG and DS) |
| 6 | JTAG in HMAC Downstream mode |
| 7 | Digital Signature peripheral in 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 4.3-3 provides the details of parameters in BLOCK1 ~ BLOCK10.

Table 4.3-3. Parameters in BLOCK1 to BLOCK10

| BLOCK | Parameters | Bit Width | Accessible by Hardware | Write Protection by EFUSE_WR_DIS Bit Number | Read Protection byEFUSE_RD_DIS Bit Number | Description |
|---|---|---|---|---|---|---|
| BLOCK1 | EFUSE_MAC | 48 | N | 20 | N/A | MAC address |
| | EFUSE_SPI_PAD_ CONFIGURE | [0:5] | N | 20 | N/A | CLK |
| | | [6:11] | N | 20 | N/A | Q (D1) |
| | | [12:17] | N | 20 | N/A | D (D0) |
| | | [18:23] | N | 20 | N/A | CS |
| | | [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 |
| | EFUSE_SYS_DATA_PART0 | 78 | N | 20 | N/A | System data |
| BLOCK2 | EFUSE_SYS_DATA_PART1 | 256 | N | 21 | N/A | System data |
| BLOCK3 | EFUSE_USR_DATA | 256 | N | 22 | N/A | User data |
| BLOCK4 | EFUSE_KEY0_DATA | 256 | Y | 23 | 0 | KEY0 or user data |
| BLOCK5 | EFUSE_KEY1_DATA | 256 | Y | 24 | 1 | KEY1 or user data |
| BLOCK6 | EFUSE_KEY2_DATA | 256 | Y | 25 | 2 | KEY2 or user data |
| BLOCK7 | EFUSE_KEY3_DATA | 256 | Y | 26 | 3 | KEY3 or user data |
| BLOCK8 | EFUSE_KEY4_DATA | 256 | Y | 27 | 4 | KEY4 or user data |
| BLOCK9 | EFUSE_KEY5_DATA | 256 | Y | 28 | 5 | KEY5 or user data |
| BLOCK10 | EFUSE_SYS_DATA_PART2 | 256 | N | 29 | 6 | System data |

Among these blocks, BLOCK4 ~ 9 stores KEY0 ~ 5, respectively. Up to six 256-bit keys can be written into eFuse. Whenever a key is written, its purpose value should also be written (see table 4.3-2). For example, when a key for the JTAG function in HMAC Downstream mode is written to KEY3 (i.e., BLOCK7), its key purpose value 6 should also be written to EFUSE_KEY_PURPOSE_3.

**Note:**
Do not program the XTS-AES key into the KEY5 block, i.e., BLOCK9. Otherwise, the key may be unreadable. Instead, program it into the preceding blocks, i.e., BLOCK4 ~ BLOCK8. The last block, BLOCK9, is used to program other keys.

BLOCK1 ~ BLOCK10 use the RS coding scheme, so there are some restrictions on writing to these parameters. For more detailed information, please refer to Section 4.3.1.3 and Section 4.3.2.

#### 4.3.1.1 EFUSE_WR_DIS

Parameter EFUSE_WR_DIS determines whether individual eFuse parameters are write-protected. After EFUSE_WR_DIS has been programmed, execute an eFuse read operation so the new values would take effect.

Column "Write Protection by EFUSE_WR_DIS Bit Number" in Table 4.3-1 and Table 4.3-3 list the specific bits in EFUSE_WR_DIS that disable writing.

When the write protection bit of a parameter is set to 0, it means that this parameter is not write-protected and can be programmed, unless it has been programmed before.

When the write protection bit of a parameter is set to 1, it means that this parameter is write-protected and none of its bits can be modified, with non-programmed bits always remaining 0 while programmed bits always remain 1.

#### 4.3.1.2 EFUSE_RD_DIS

Only the eFuse blocks in BLOCK4 ~ BLOCK10 can be individually read protected to prevent any access from outside the chip, as shown in column "Read Protection by EFUSE_RD_DIS Bit Number" of Table 4.3-3. After EFUSE_RD_DIS has been programmed, execute an eFuse read operation so the new values would take effect.

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

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

When BLOCK4 ~ BLOCK10 are set to be read-protected, the data in these blocks are not readable by users, but they can still be read by hardware cryptography modules, if the EFUSE_KEY_PURPOSE_*n* bit is set accordingly.

#### 4.3.1.3 Data Storage

Internally, eFuses use hardware encoding schemes to protect data from corruption, which are invisible for users.

All BLOCK0 parameters except for EFUSE_WR_DIS are stored with four backups, meaning each bit is stored four times. This backup scheme is not visible to users.

BLOCK1 ~ BLOCK10 use RS (44, 32) coding scheme that supports up to 6 bytes of automatic error correction. The primitive polynomial of RS (44, 32) is $p(x) = x^8 + x^4 + x^3 + x^2 + 1$.

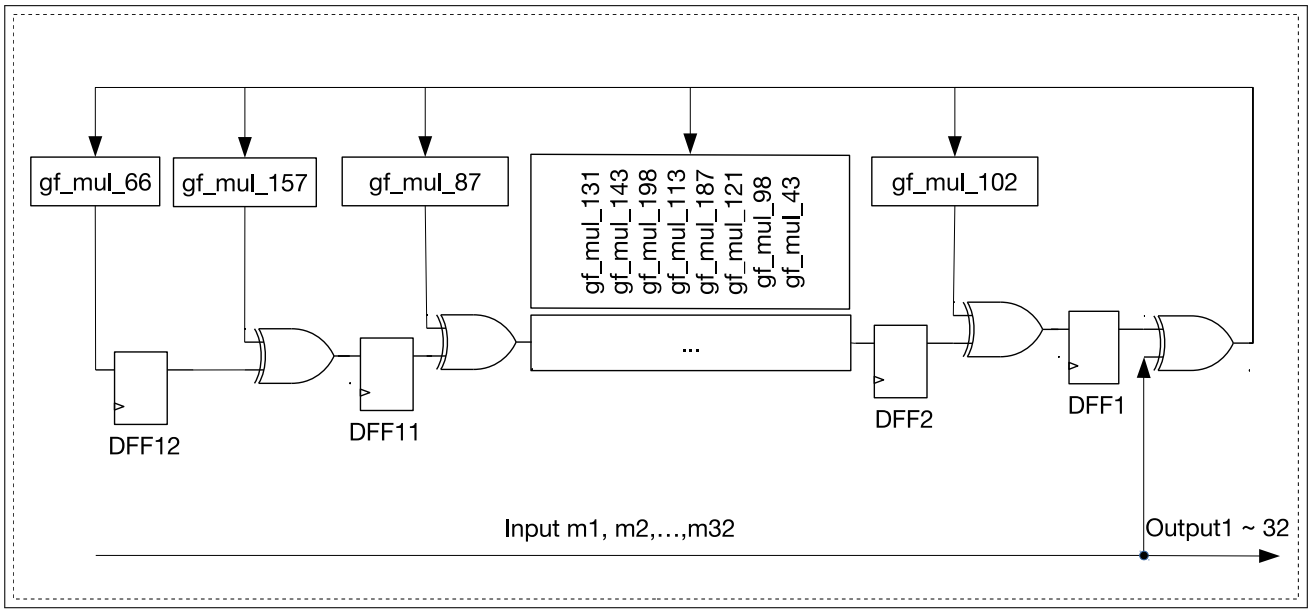



Figure 4.3-1. Shift Register Circuit (first 32 output)

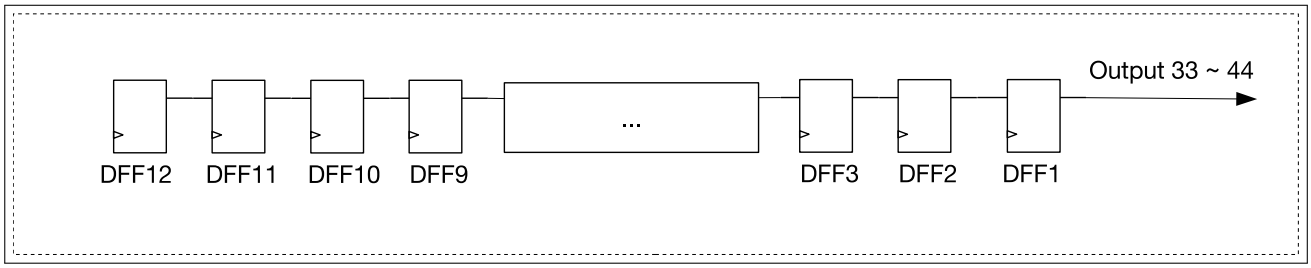



Figure 4.3-2. Shift Register Circuit (last 12 output)

The shift register circuit shown in Figure 4.3-1 and 4.3-2 processes 32 data bytes using RS (44, 32). This coding scheme encodes 32 bytes of data into 44 bytes:

- Bytes [0:31] are the data bytes itself
- Bytes [32:43] are the encoded parity bytes stored in 8-bit flip-flops DFF1, DFF2, ..., DFF12 (gf_mul_n, where n is an integer, is the result of multiplying a byte of data ...)

After that, the hardware burns into eFuse the 44-byte codeword consisting of the data bytes followed by the parity bytes.

When the eFuse block is read back, the eFuse controller automatically decodes the codeword and applies error correction if needed.

Because the RS check codes are generated on the entire 256-bit eFuse block, each block can only be written once.

### 4.3.2 Programming of Parameters

The eFuse controller can only program eFuse parameters in one block at a time. BLOCK0 ~ BLOCK10 share the same address range to store the parameters to be programmed. Configure parameter EFUSE_BLK_NUM

to indicate which block should be programmed.

#### Programming BLOCK0

When EFUSE_BLK_NUM is set to 0, BLOCK0 will be programmed. Register EFUSE_PGM_DATA0_REG stores EFUSE_WR_DIS. Registers EFUSE_PGM_DATA1_REG ~ EFUSE_PGM_DATA5_REG store the information of parameters to be programmed. Note that 9 BLOCK0 bits are readable but useless to users and must always be set to 0 in the programming registers. The specific bits are:

- EFUSE_PGM_DATA1_REG[24:21]
- EFUSE_PGM_DATA1_REG[31:27]

Data in registers 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 is set to 1, registers EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA5_REG store the BLOCK1 parameters to be programmed. Registers EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_DATA2_REG store the corresponding RS check codes. Data in registers EFUSE_PGM_DATA6_REG ~ EFUSE_PGM_DATA7_REG are ignored when programming BLOCK1, and the RS check codes will be calculated with these bits all treated as 0.

#### Programming BLOCK2 ~ 10

When EFUSE_BLK_NUM is set to 2 ~ 10, registers EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG store the parameters to be programmed to this block. Registers 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. Configure the value of parameter EFUSE_BLK_NUM to determine the block to be programmed.
2. Write parameters to be programmed to registers EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG and EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG.
3. Make sure the eFuse programming voltage VDDQ is configured correctly as described in Section 4.3.4.
4. Configure the field EFUSE_OP_CODE of register EFUSE_CONF_REG to 0x5A5A.
5. Configure the field EFUSE_PGM_CMD of register EFUSE_CMD_REG to 1.
6. Poll register EFUSE_CMD_REG until it is 0x0, or wait for a PGM_DONE interrupt. For more information on how to identify a PGM/READ_DONE interrupt, please see the end of Section 4.3.3.
7. Clear the parameters in EFUSE_PGM_DATA0_REG ~ EFUSE_PGM_DATA7_REG and EFUSE_PGM_CHECK_VALUE0_REG ~ EFUSE_PGM_CHECK_VALUE2_REG.
8. Trigger an eFuse read operation (see Section 4.3.3) to update eFuse registers with the new values.
9. Check error record registers. If the values read in error record registers are not 0, the programming process should be performed again following above steps 1 ~ 7. Please check the following error record registers for different eFuse blocks:

- BLOCK0: EFUSE_RD_REPEAT_ERR0_REG ~ EFUSE_RD_REPEAT_ERR4_REG
- BLOCK1: EFUSE_RD_RS_ERR0_REG[2:0], EFUSE_RD_RS_ERR0_REG[7]
- BLOCK2: EFUSE_RD_RS_ERR0_REG[6:4], EFUSE_RD_RS_ERR0_REG[11]
- BLOCK3: EFUSE_RD_RS_ERR0_REG[10:8], EFUSE_RD_RS_ERR0_REG[15]
- BLOCK4: EFUSE_RD_RS_ERR0_REG[14:12], EFUSE_RD_RS_ERR0_REG[19]
- BLOCK5: EFUSE_RD_RS_ERR0_REG[18:16], EFUSE_RD_RS_ERR0_REG[23]
- BLOCK6: EFUSE_RD_RS_ERR0_REG[22:20], EFUSE_RD_RS_ERR0_REG[27]
- BLOCK7: EFUSE_RD_RS_ERR0_REG[26:24], EFUSE_RD_RS_ERR0_REG[31]
- BLOCK8: EFUSE_RD_RS_ERR0_REG[30:28], EFUSE_RD_RS_ERR1_REG[3]
- BLOCK9: EFUSE_RD_RS_ERR1_REG[2:0], EFUSE_RD_RS_ERR1_REG[2:0][7]
- BLOCK10: EFUSE_RD_RS_ERR1_REG[2:0][6:4]

**Limitations**

In BLOCK0, each bit can be programmed separately. However, we recommend to 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.

### 4.3.3 User Read of Parameters

Users cannot read eFuse bits directly. The eFuse Controller hardware reads all eFuse bits and stores the results to their corresponding registers in its memory space. Then, users can read eFuse bits by reading the registers that start with EFUSE_RD_. Details are provided in Table 4.3-4.

**Table 4.3-4. Registers Information**

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

**Updating eFuse read registers**

The eFuse Controller reads internal eFuses to update corresponding registers. This read operation happens on system reset and can also be triggered manually by users as needed (e.g., if new eFuse values have been programmed). The process of triggering a read operation by users is as follows:

1. Configure the field EFUSE_OP_CODE in register EFUSE_CONF_REG to 0x5AA5.
2. Configure the field EFUSE_READ_CMD in register EFUSE_CMD_REG to 1.
3. Poll register EFUSE_CMD_REG until it is 0x0, or wait for a READ_DONE interrupt. Information on how to identify a PGM/READ_DONE interrupt is provided below in this section.
4. Read the values of each parameter from memory.

The eFuse read registers will hold all values until the next read operation.

#### Error detection

Error record registers allow users to detect if there are any inconsistencies in the stored backup eFuse parameters.

Registers EFUSE_RD_REPEAT_ERR0 ~ 3_REG indicate if there are any errors of programmed parameters (except for EFUSE_WR_DIS) in BLOCK0 (value 1 indicates an error is detected, and the bit becomes 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.

The values of above registers will be updated every time after the eFuse read registers have been updated.

#### Identifying program/read operation

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

- Method one:
  1. Poll bit 1/0 in register EFUSE_INT_RAW_REG until it becomes 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 the eFuse Controller to post a PGM/READ_DONE interrupt.
  2. Configure the Interrupt Matrix to enable the CPU to respond to eFuse interrupt signals, see Chapter *8 Interrupt Matrix (INTERRUPT)*.
  3. Wait for the PGM/READ_DONE interrupt.
  4. Set bit 1/0 in register EFUSE_INT_CLR_REG to 1 to clear the PGM/READ_DONE interrupt.

#### Note

When eFuse controller updating its registers, it will use EFUSE_PGM_DATA*n*_REG (n=0, 1, .., 7) again to store data. So please do not write important data into these registers before this updating process initiated. During the chip boot process, eFuse controller will update eFuse data into registers which can be accessed by users automatically. Users can get programmed eFuse data by reading corresponding registers. Thus, it is no need to update eFuse read registers in such case.

### 4.3.4 eFuse VDDQ Timing

The eFuse Controller operates with 20 MHz of clock frequency, and its programming voltage VDDQ should be configured as follows:

- EFUSE_DAC_NUM (the rising period of VDDQ): The default value of VDDQ is 2.5 V and the voltage increases by 0.01 V in each clock cycle. Thus, the default value of this parameter is 255;
- EFUSE_DAC_CLK_DIV (the clock divisor of VDDQ): The clock period to program VDDQ should be larger than 1 $\mu$s;
- EFUSE_PWR_ON_NUM (the power-up time for VDDQ): The programming voltage should be stabilized after this time, which means the value of this parameter should be configured to exceed the result of EFUSE_DAC_CLK_DIV times EFUSE_DAC_NUM;
- EFUSE_PWR_OFF_NUM (the power-out time for VDDQ): The value of this parameter should be larger than 10 $\mu$s.

**Table 4.3-5. Configuration of Default VDDQ Timing Parameters**

| EFUSE_DAC_NUM | EFUSE_DAC_CLK_DIV | EFUSE_PWR_ON_NUM | EFUSE_PWR_OFF_NUM |
|---|---|---|---|
| 0xFF | 0x28 | 0x3000 | 0x190 |

### 4.3.5 The Use of Parameters by Hardware Modules

Some hardware modules are directly connected to the eFuse peripheral in order to use the parameters listed in Table 4.3-1 and Table 4.3-3, specifically those marked with "Y" in columns "Accessible by Hardware". Users cannot intervene in this process.

### 4.3.6 Interrupts

- PGM_DONE interrupt: Triggered when eFuse programming has finished. To enable this interrupt, set the EFUSE_PGM_DONE_INT_ENA field of register EFUSE_INT_ENA_REG to 1;
- READ_DONE interrupt: Triggered when eFuse reading has finished. To enable this interrupt, set the EFUSE_READ_DONE_INT_ENA field of register EFUSE_INT_ENA_REG to 1.

## 4.4 Register Summary

The addresses in this section are relative to eFuse Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **PGM Data Register** | | | |
| EFUSE_PGM_DATA0_REG | Register 0 that stores data to be programmed | 0x0000 | R/W |
| EFUSE_PGM_DATA1_REG | Register 1 that stores data to be programmed | 0x0004 | R/W |
| EFUSE_PGM_DATA2_REG | Register 2 that stores data to be programmed | 0x0008 | R/W |
| EFUSE_PGM_DATA3_REG | Register 3 that stores data to be programmed | 0x000C | R/W |
| EFUSE_PGM_DATA4_REG | Register 4 that stores data to be programmed | 0x0010 | R/W |
| EFUSE_PGM_DATA5_REG | Register 5 that stores data to be programmed | 0x0014 | R/W |
| EFUSE_PGM_DATA6_REG | Register 6 that stores data to be programmed | 0x0018 | R/W |
| EFUSE_PGM_DATA7_REG | Register 7 that stores data to be programmed | 0x001C | R/W |
| EFUSE_PGM_CHECK_VALUE0_REG | Register 0 that stores the RS code to be programmed | 0x0020 | R/W |
| EFUSE_PGM_CHECK_VALUE1_REG | Register 1 that stores the RS code to be programmed | 0x0024 | R/W |
| EFUSE_PGM_CHECK_VALUE2_REG | Register 2 that stores the RS code to be programmed | 0x0028 | R/W |
| **Read Data Register** | | | |
| EFUSE_RD_WR_DIS_REG | BLOCK0 data register 0 | 0x002C | RO |
| EFUSE_RD_REPEAT_DATA0_REG | BLOCK0 data register 1 | 0x0030 | RO |
| EFUSE_RD_REPEAT_DATA1_REG | BLOCK0 data register 2 | 0x0034 | RO |
| EFUSE_RD_REPEAT_DATA2_REG | BLOCK0 data register 3 | 0x0038 | RO |
| EFUSE_RD_REPEAT_DATA3_REG | BLOCK0 data register 4 | 0x003C | RO |
| EFUSE_RD_REPEAT_DATA4_REG | BLOCK0 data register 5 | 0x0040 | RO |
| EFUSE_RD_MAC_SPI_SYS_0_REG | BLOCK1 data register 0 | 0x0044 | RO |
| EFUSE_RD_MAC_SPI_SYS_1_REG | BLOCK1 data register 1 | 0x0048 | RO |
| EFUSE_RD_MAC_SPI_SYS_2_REG | BLOCK1 data register 2 | 0x004C | RO |
| EFUSE_RD_MAC_SPI_SYS_3_REG | BLOCK1 data register 3 | 0x0050 | RO |
| EFUSE_RD_MAC_SPI_SYS_4_REG | BLOCK1 data register 4 | 0x0054 | RO |
| EFUSE_RD_MAC_SPI_SYS_5_REG | BLOCK1 data register 5 | 0x0058 | RO |
| EFUSE_RD_SYS_PART1_DATA0_REG | Register 0 of BLOCK2 (system) | 0x005C | RO |
| EFUSE_RD_SYS_PART1_DATA1_REG | Register 1 of BLOCK2 (system) | 0x0060 | RO |
| EFUSE_RD_SYS_PART1_DATA2_REG | Register 2 of BLOCK2 (system) | 0x0064 | RO |
| EFUSE_RD_SYS_PART1_DATA3_REG | Register 3 of BLOCK2 (system) | 0x0068 | RO |
| EFUSE_RD_SYS_PART1_DATA4_REG | Register 4 of BLOCK2 (system) | 0x006C | RO |
| EFUSE_RD_SYS_PART1_DATA5_REG | Register 5 of BLOCK2 (system) | 0x0070 | RO |
| EFUSE_RD_SYS_PART1_DATA6_REG | Register 6 of BLOCK2 (system) | 0x0074 | RO |
| EFUSE_RD_SYS_PART1_DATA7_REG | Register 7 of BLOCK2 (system) | 0x0078 | RO |
| EFUSE_RD_USR_DATA0_REG | Register 0 of BLOCK3 (user) | 0x007C | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| 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 |
| 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 |

| Name | Description | Address | Access |
|---|---|---|---|
| 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_PART2_DATA0_REG | Register 0 of BLOCK10 (system) | 0x015C | RO |
| EFUSE_RD_SYS_PART2_DATA1_REG | Register 1 of BLOCK10 (system) | 0x0160 | RO |
| EFUSE_RD_SYS_PART2_DATA2_REG | Register 2 of BLOCK10 (system) | 0x0164 | RO |
| EFUSE_RD_SYS_PART2_DATA3_REG | Register 3 of BLOCK10 (system) | 0x0168 | RO |
| EFUSE_RD_SYS_PART2_DATA4_REG | Register 4 of BLOCK10 (system) | 0x016C | RO |
| EFUSE_RD_SYS_PART2_DATA5_REG | Register 5 of BLOCK10 (system) | 0x0170 | RO |
| EFUSE_RD_SYS_PART2_DATA6_REG | Register 6 of BLOCK10 (system) | 0x0174 | RO |
| EFUSE_RD_SYS_PART2_DATA7_REG | Register 7 of BLOCK10 (system) | 0x0178 | RO |
| **Report Register** | | | |
| 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 |
| **Configuration Register** | | | |
| EFUSE_CLK_REG | eFuse clock configuration register | 0x01C8 | R/W |
| EFUSE_CONF_REG | eFuse operation mode configuration register | 0x01CC | R/W |
| EFUSE_CMD_REG | eFuse command register | 0x01D4 | varies |
| EFUSE_DAC_CONF_REG | Controls the eFuse programming voltage | 0x01E8 | R/W |
| EFUSE_RD_TIM_CONF_REG | Configures read timing parameters | 0x01EC | R/W |
| EFUSE_WR_TIM_CONF1_REG | Configuration register 1 of eFuse programming timing parameters | 0x01F0 | R/W |
| EFUSE_WR_TIM_CONF2_REG | Configuration register 2 of eFuse programming timing parameters | 0x01F4 | R/W |
| **Status Register** | | | |
| EFUSE_STATUS_REG | eFuse status register | 0x01D0 | RO |
| **Interrupt Register** | | | |

| Name | Description | Address | Access |
|---|---|---|---|
| EFUSE_INT_RAW_REG | eFuse raw interrupt register | 0x01D8 | R/WC/SS |
| 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 |
| **Version Register** | | | |
| EFUSE_DATE_REG | Version control register | 0x01FC | R/W |

## 4.5 Registers

The addresses in this section are relative to eFuse Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 4.1. EFUSE_PGM_DATA0_REG (0x0000)**

| EFUSE_PGM_DATA_0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_PGM_DATA_0** The content of the 0th 32-bit data to be programmed. (R/W)

**Register 4.2. EFUSE_PGM_DATA1_REG (0x0004)**

| EFUSE_PGM_DATA_1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_PGM_DATA_1** The content of the 1st 32-bit data to be programmed. (R/W)

**Register 4.3. EFUSE_PGM_DATA2_REG (0x0008)**

| EFUSE_PGM_DATA_2 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_PGM_DATA_2** The content of the 2nd 32-bit data to be programmed. (R/W)

**Register 4.4. EFUSE_PGM_DATA3_REG (0x000C)**

| EFUSE_PGM_DATA_3 (31 … 0) | Reset |
|---|---|
| 0x000000 | |

**EFUSE_PGM_DATA_3** The content of the 3rd 32-bit data to be programmed. (R/W)

**Register 4.5. EFUSE_PGM_DATA4_REG (0x0010)**

| EFUSE_PGM_DATA_4 (31 … 0) | Reset |
|---|---|
| 0x000000 | |

**EFUSE_PGM_DATA_4** The content of the 4th 32-bit data to be programmed. (R/W)

**Register 4.6. EFUSE_PGM_DATA5_REG (0x0014)**

| EFUSE_PGM_DATA_5 (31 … 0) | Reset |
|---|---|
| 0x000000 | |

**EFUSE_PGM_DATA_5** The content of the 5th 32-bit data to be programmed. (R/W)

**Register 4.7. EFUSE_PGM_DATA6_REG (0x0018)**

| EFUSE_PGM_DATA_6 (31 … 0) | Reset |
|---|---|
| 0x000000 | |

**EFUSE_PGM_DATA_6** The content of the 6th 32-bit data to be programmed. (R/W)

Register 4.8. EFUSE_PGM_DATA7_REG (0x001C)

| EFUSE_PGM_DATA_7 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_PGM_DATA_7** The content of the 7th 32-bit data to be programmed. (R/W)

Register 4.9. EFUSE_PGM_CHECK_VALUE0_REG (0x0020)

| EFUSE_PGM_RS_DATA_0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

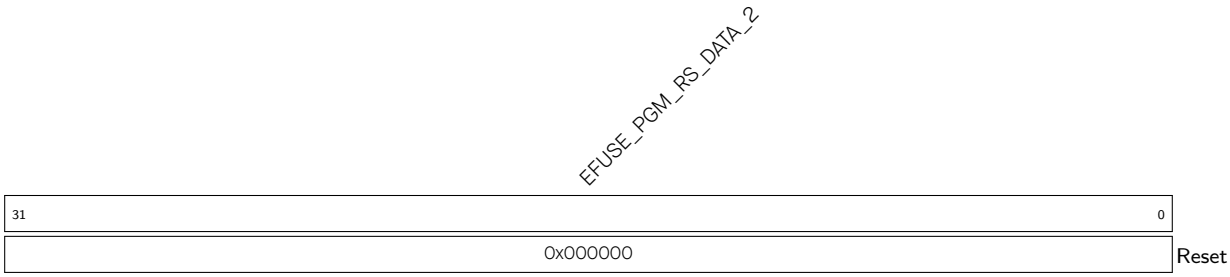
**EFUSE_PGM_RS_DATA_0** The content of the 0th 32-bit RS code to be programmed. (R/W)

Register 4.10. EFUSE_PGM_CHECK_VALUE1_REG (0x0024)

| EFUSE_PGM_RS_DATA_1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

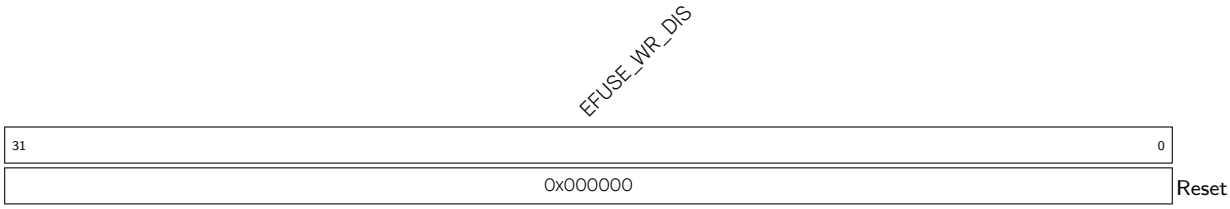
**EFUSE_PGM_RS_DATA_1** The content of the 1st 32-bit RS code to be programmed. (R/W)

**Register 4.11. EFUSE_PGM_CHECK_VALUE2_REG (0x0028)**




**EFUSE_PGM_RS_DATA_2** The content of the 2nd 32-bit RS code to be programmed. (R/W)

**Register 4.12. EFUSE_RD_WR_DIS_REG (0x002C)**




**EFUSE_WR_DIS** Represents whether programming of corresponding eFuse part is disabled or enabled. 1: Disabled. 0: Enabled. (RO)

**Register 4.13. EFUSE_RD_REPEAT_DATA0_REG (0x0030)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | EFUSE_VDD_SPI_AS_GPIO | EFUSE_USB_EXCHG_PINS | (reserved) | | | | EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT | EFUSE_DIS_PAD_JTAG | EFUSE_SOFT_DIS_JTAG | | | | EFUSE_JTAG_SEL_ENABLE | EFUSE_DIS_TWAI | EFUSE_RPT4_RESERVED6 | EFUSE_DIS_FORCE_DOWNLOAD | EFUSE_DIS_USB_SERIAL_JTAG | EFUSE_DIS_DOWNLOAD_ICACHE | EFUSE_DIS_USB_JTAG | EFUSE_DIS_ICACHE | EFUSE_DIS_RTC_RAM_BOOT | EFUSE_RD_DIS | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x0 | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x0 | | | | | | | Reset |

**EFUSE_RD_DIS** Represents whether users' reading from BLOCK4 ~ 10 is disabled or enabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_DIS_RTC_RAM_BOOT** Reserved (used for four backups method). (RO)

**EFUSE_DIS_ICACHE** Represents whether iCache is disabled or enabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_DIS_USB_JTAG** Represents whether the USB-to-JTAG function is disabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_DIS_DOWNLOAD_ICACHE** Represents whether iCache is disabled in download mode (boot_mode[3:0] is 0, 1, 2, 3, 6, 7). 1: Disabled. 0: Enabled. (RO)

**EFUSE_DIS_USB_SERIAL_JTAG** Represents whether USB-Serial-JTAG is disabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_DIS_FORCE_DOWNLOAD** Represents whether the function that forces chip into download mode is disabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_RPT4_RESERVED6** Reserved (used for four backups method). (RO)

**EFUSE_DIS_TWAI** Represents whether TWAI function is disabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_JTAG_SEL_ENABLE** Represents whether to use JTAG directly. 1: Use directly. 0: Not use directly. (RO)

**EFUSE_SOFT_DIS_JTAG** Represents whether JTAG is disabled in the soft way. Odd count of bits with a value of 1: Disabled. It can still be restarted via HMAC. Even count of bits with a value of 1: Enabled. (RO)

**EFUSE_DIS_PAD_JTAG** Represents whether JTAG is disabled in the hard way (permanently). 1: Disabled. 0: Enabled. (RO)

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

**Register 4.13. EFUSE_RD_REPEAT_DATA0_REG (0x0030)**

**Continued from the previous page...**

**EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT** Represents whether flash encryption is disabled (except in SPI boot mode). 1: Disabled. 0: Enabled. (RO)

**EFUSE_USB_EXCHG_PINS** Represents whether or not USB D+ and D- pins are swapped. 1: Swapped. 0: Not swapped. (RO)

Note: The eFuse has a design flaw and does **not** move the pullup (needed to detect USB speed), resulting in the PC thinking the chip is a low-speed device, which stops communication. For detailed information, please refer to Chapter 30 *USB Serial/JTAG Controller (USB_SERIAL_JTAG)*.

**EFUSE_VDD_SPI_AS_GPIO** Represents whether the VDD_SPI pin is used as a regular GPIO. 1: Used as a regular GPIO. 0: Not used as a regular GPIO. (RO)

**Register 4.14. 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 | EFUSE_RPT4_RESERVED2 | |
|---|---|---|---|---|---|---|---|---|
| 31 28 | 27 24 | 23 | 22 | 21 | 20 18 | 17 16 | 15 0 | |
| 0x0 | 0x0 | 0 | 0 | 0 | 0x0 | 0x0 | 0x00 | Reset |

**EFUSE_RPT4_RESERVED2** Reserved (used for four backups method). (RO)

**EFUSE_WDT_DELAY_SEL** Represents RTC watchdog timeout threshold. Measurement unit: slow clock cycle. 00: 40000, 01: 80000, 10: 160000, 11:320000. (RO)

**EFUSE_SPI_BOOT_CRYPT_CNT** Represents whether SPI boot encrypt/decrypt is disabled or enabled. Odd count of bits with a value of 1: Enabled. Even count of bits with a value of 1: Disabled. (RO)

**EFUSE_SECURE_BOOT_KEY_REVOKE0** Represents whether or not the first secure boot key is revoked. 1: Revoked. 0: Not revoked. (RO)

**EFUSE_SECURE_BOOT_KEY_REVOKE1** Represents whether or not the second secure boot key is revoked. 1: Revoked. 0: Not revoked. (RO)

**EFUSE_SECURE_BOOT_KEY_REVOKE2** Represents whether or not the third secure boot key is revoked. 1: Revoked. 0: Not revoked. (RO)

**EFUSE_KEY_PURPOSE_0** Represents purpose of Key0. (RO)

**EFUSE_KEY_PURPOSE_1** Represents purpose of Key1. (RO)

**Register 4.15. EFUSE_RD_REPEAT_DATA2_REG (0x0038)**

| EFUSE_FLASH_TPUW | EFUSE_RPT4_RESERVED0 | EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE | EFUSE_SECURE_BOOT_EN | EFUSE_RPT4_RESERVED3 | EFUSE_KEY_PURPOSE_5 | EFUSE_KEY_PURPOSE_4 | EFUSE_KEY_PURPOSE_3 | EFUSE_KEY_PURPOSE_2 | |
|---|---|---|---|---|---|---|---|---|---|
| 31 28 | 27 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 4 | 3 0 | |
| 0x0 | 0x0 | 0 | 0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | Reset |

**EFUSE_KEY_PURPOSE_2** Represents purpose of Key2. (RO)

**EFUSE_KEY_PURPOSE_3** Represents purpose of Key3. (RO)

**EFUSE_KEY_PURPOSE_4** Represents purpose of Key4. (RO)

**EFUSE_KEY_PURPOSE_5** Represents purpose of Key5. (RO)

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

**EFUSE_SECURE_BOOT_EN** Represents whether secure boot is enabled or disabled. 1: Enabled. 0: Disabled. (RO)

**EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE** Represents whether aggressive revoke of secure boot keys is enabled or disabled. 1: Enabled. 0: Disabled. (RO)

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

**EFUSE_FLASH_TPUW** Represents flash waiting time after power-up. Measurement unit: ms. If the value is less than 15, the waiting time is the configurable value. Otherwise, the waiting time is always 30 ms. (RO)

**Register 4.16. EFUSE_RD_REPEAT_DATA3_REG (0x003C)**

| 31 | 30 | 29 | 14 | 13 | 12 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EFUSE_ERR_RST_ENABLE | EFUSE_RPT4_RESERVED1 | EFUSE_SECURE_VERSION | | EFUSE_FORCE_SEND_RESUME | EFUSE_RPT4_RESERVED5 | | EFUSE_UART_PRINT_CONTROL | | EFUSE_ENABLE_SECURITY_DOWNLOAD | EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE | EFUSE_RPT4_RESERVED7 | EFUSE_USB_PRINT_CHANNEL | EFUSE_RPT4_RESERVED8 | EFUSE_DIS_DOWNLOAD_MODE | |
| 0 | 0 | 0x00 | | 0 | 0 | | 0x0 | | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**EFUSE_DIS_DOWNLOAD_MODE** Represents whether download mode (boot_mode[3:0] = 0, 1, 2, 3, 6, 7) is disabled or enabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_RPT4_RESERVED8** Reserved (used for four backups method). (RO)

**EFUSE_USB_PRINT_CHANNEL** Represents whether USB printing is disabled or enabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_RPT4_RESERVED7** Reserved (used for four backups method). (RO)

**EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE** Represents whether download through USB-Serial-JTAG is disabled or enabled. 1: Disabled. 0: Enabled. (RO)

**EFUSE_ENABLE_SECURITY_DOWNLOAD** Represents whether secure UART download mode is enabled or disabled (read/write flash only). 1: Enabled. 0: Disabled. (RO)

**EFUSE_UART_PRINT_CONTROL** Represents the UART boot message output mode. 00: Enabled. 01: Enabled when GPIO8 is low at reset. 10: Enabled when GPIO8 is high at reset. 11: Disabled. (RO)

**EFUSE_RPT4_RESERVED5** Reserved (used for four backups method). (RO)

**EFUSE_FORCE_SEND_RESUME** Represents whether or not to force ROM code to send a resume command during SPI boot. 1: Send. 0: Not send. (RO)

**EFUSE_SECURE_VERSION** Represents the values of version control register (used by ESP-IDF anti-rollback feature). (RO)

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

**EFUSE_ERR_RST_ENABLE** Represents whether to enable the check for error registers of block0. 1: Enabled. 0: Disabled. (RO)

**Register 4.17. EFUSE_RD_REPEAT_DATA4_REG (0x0040)**

| (reserved) | EFUSE_RPT4_RESERVED4 | |
|---|---|---|
| 31 24 | 23 0 | |
| 0 0 0 0 0 0 0 0 | 0x0000 | Reset |

**EFUSE_RPT4_RESERVED4** Reserved (used for four backups method). (RO)

**Register 4.18. EFUSE_RD_MAC_SPI_SYS_0_REG (0x0044)**

| EFUSE_MAC_0 | |
|---|---|
| 31 0 | |
| 0x000000 | Reset |

**EFUSE_MAC_0** Stores the low 32 bits of MAC address. (RO)

**Register 4.19. EFUSE_RD_MAC_SPI_SYS_1_REG (0x0048)**

| EFUSE_SPI_PAD_CONF_0 | EFUSE_MAC_1 | |
|---|---|---|
| 31 16 | 15 0 | |
| 0x00 | 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 4.20. 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 4.21. EFUSE_RD_MAC_SPI_SYS_3_REG (0x0050)

| EFUSE_SYS_DATA_PART0_0 | EFUSE_SPI_PAD_CONF_2 | |
|---|---|---|
| 31 … 18 | 17 … 0 | |
| 0x00 | 0x000 | Reset |

**EFUSE_SPI_PAD_CONF_2** Stores the second part of SPI_PAD_CONF. (RO)

**EFUSE_SYS_DATA_PART0_0** Stores the first 14 bits of the zeroth part of system data. (RO)

Register 4.22. 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 32 bits of the zeroth part of system data. (RO)

**Register 4.23. 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 32 bits of the zeroth part of system data. (RO)

**Register 4.24. EFUSE_RD_SYS_PART1_DATA0_REG (0x005C)**

| EFUSE_SYS_DATA_PART1_0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

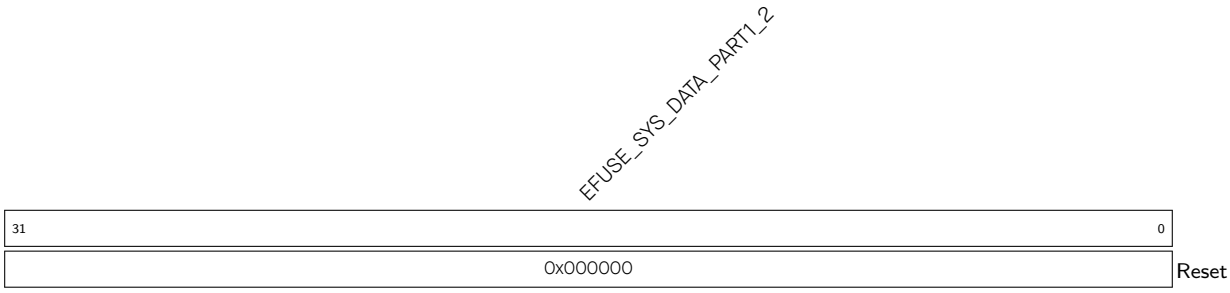
**EFUSE_SYS_DATA_PART1_0** Stores the zeroth 32 bits of the first part of system data. (RO)

**Register 4.25. EFUSE_RD_SYS_PART1_DATA1_REG (0x0060)**

| EFUSE_SYS_DATA_PART1_1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

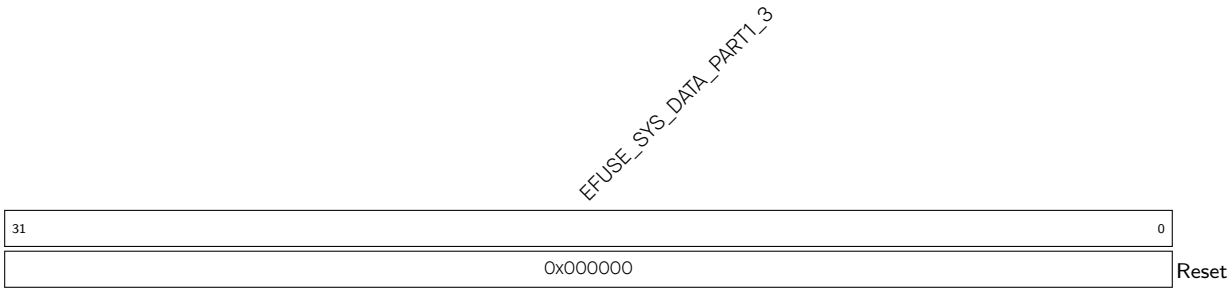
**EFUSE_SYS_DATA_PART1_1** Stores the first 32 bits of the first part of system data. (RO)

**Register 4.26. EFUSE_RD_SYS_PART1_DATA2_REG (0x0064)**



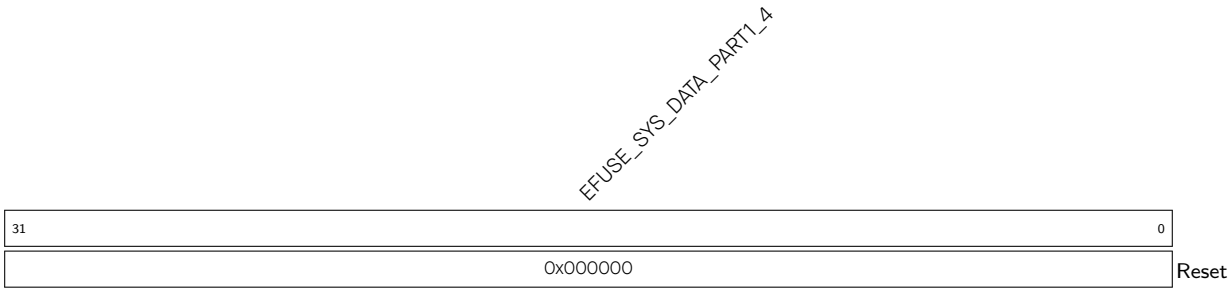

**EFUSE_SYS_DATA_PART1_2** Stores the second 32 bits of the first part of system data. (RO)

**Register 4.27. EFUSE_RD_SYS_PART1_DATA3_REG (0x0068)**



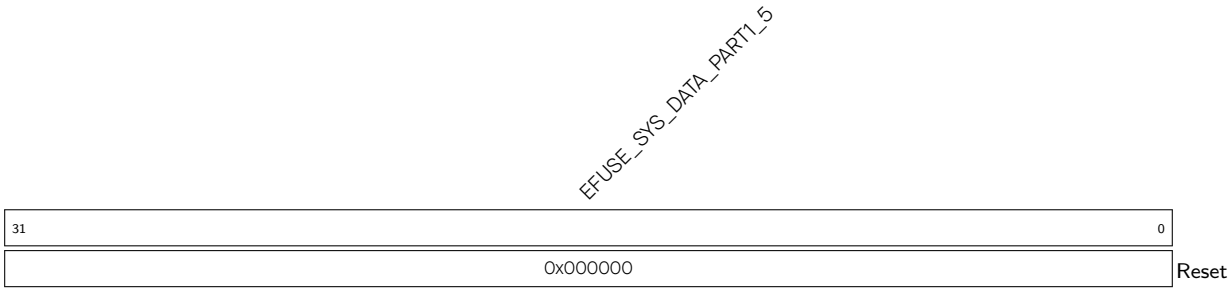

**EFUSE_SYS_DATA_PART1_3** Stores the third 32 bits of the first part of system data. (RO)

**Register 4.28. EFUSE_RD_SYS_PART1_DATA4_REG (0x006C)**



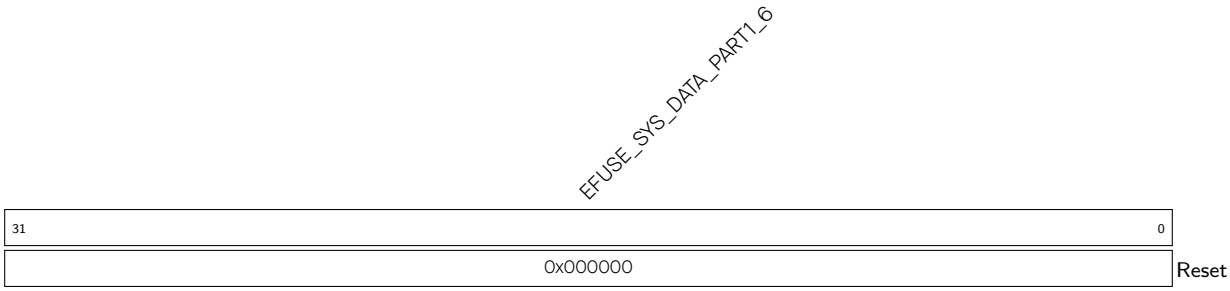

**EFUSE_SYS_DATA_PART1_4** Stores the fourth 32 bits of the first part of system data. (RO)

**Register 4.29. EFUSE_RD_SYS_PART1_DATA5_REG (0x0070)**



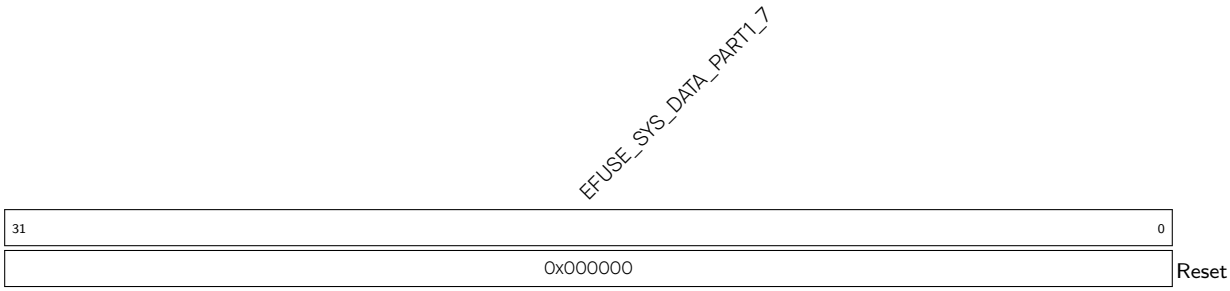

**EFUSE_SYS_DATA_PART1_5** Stores the fifth 32 bits of the first part of system data. (RO)

**Register 4.30. EFUSE_RD_SYS_PART1_DATA6_REG (0x0074)**



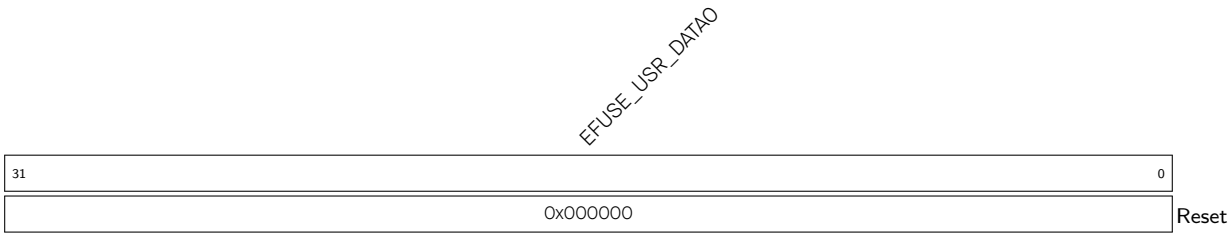

**EFUSE_SYS_DATA_PART1_6** Stores the sixth 32 bits of the first part of system data. (RO)

**Register 4.31. EFUSE_RD_SYS_PART1_DATA7_REG (0x0078)**



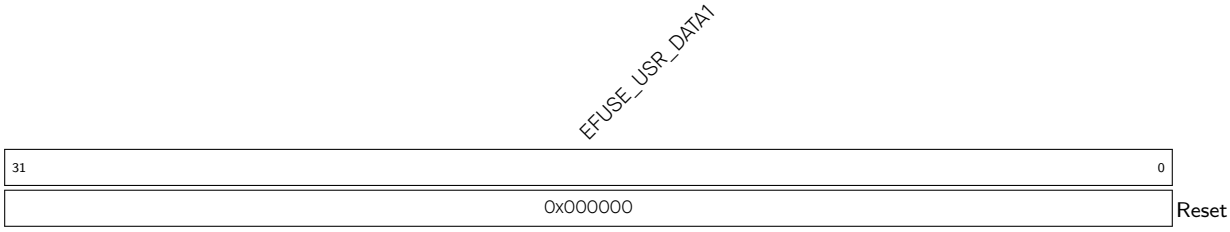

**EFUSE_SYS_DATA_PART1_7** Stores the seventh 32 bits of the first part of system data. (RO)

**Register 4.32. EFUSE_RD_USR_DATA0_REG (0x007C)**



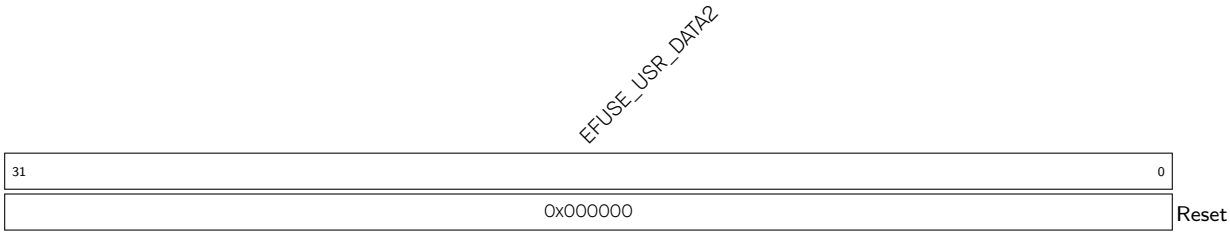

**EFUSE_USR_DATA0** Stores the zeroth 32 bits of BLOCK3 (user). (RO)

**Register 4.33. EFUSE_RD_USR_DATA1_REG (0x0080)**



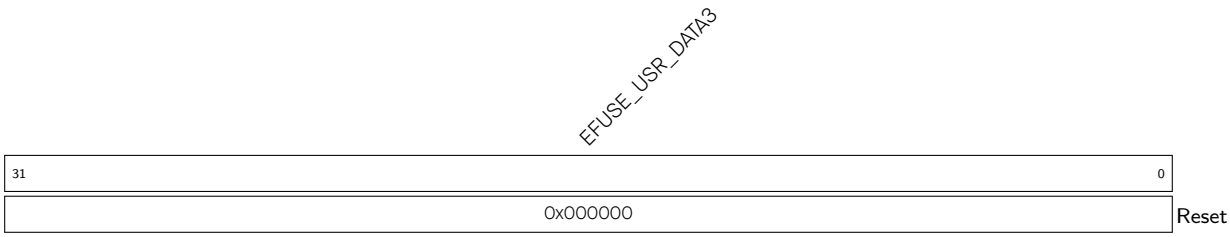

**EFUSE_USR_DATA1** Stores the first 32 bits of BLOCK3 (user). (RO)

**Register 4.34. EFUSE_RD_USR_DATA2_REG (0x0084)**



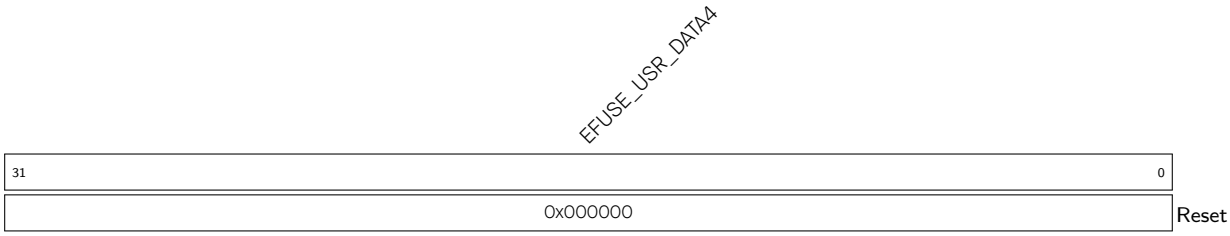

**EFUSE_USR_DATA2** Stores the second 32 bits of BLOCK3 (user). (RO)

**Register 4.35. EFUSE_RD_USR_DATA3_REG (0x0088)**



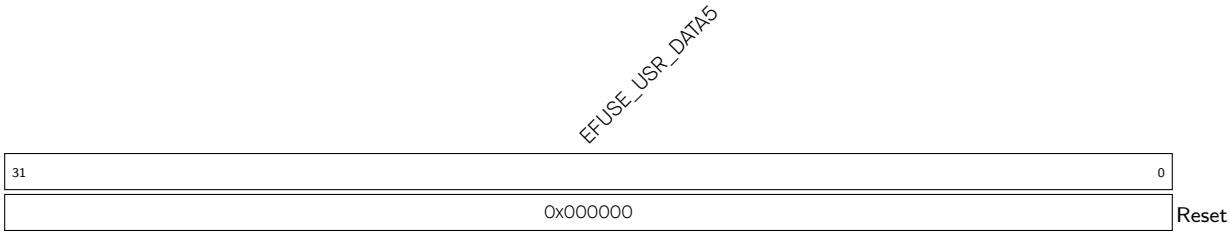

**EFUSE_USR_DATA3** Stores the third 32 bits of BLOCK3 (user). (RO)

**Register 4.36. EFUSE_RD_USR_DATA4_REG (0x008C)**



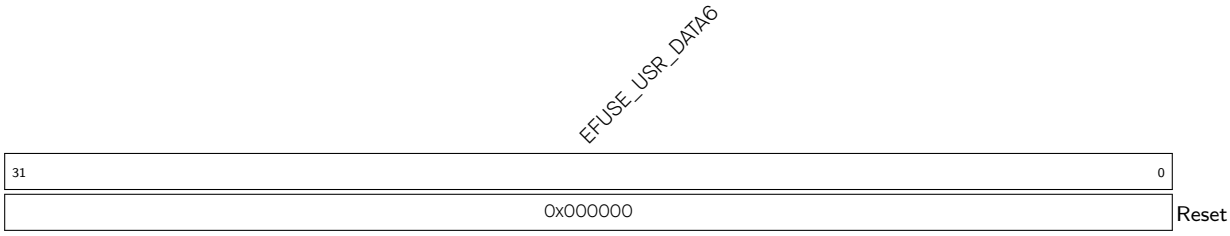

**EFUSE_USR_DATA4** Stores the fourth 32 bits of BLOCK3 (user). (RO)

**Register 4.37. EFUSE_RD_USR_DATA5_REG (0x0090)**



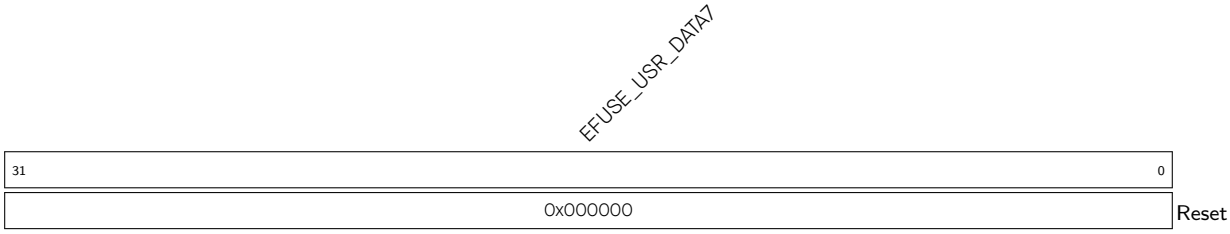

**EFUSE_USR_DATA5** Stores the fifth 32 bits of BLOCK3 (user). (RO)

**Register 4.38. EFUSE_RD_USR_DATA6_REG (0x0094)**




**EFUSE_USR_DATA6** Stores the sixth 32 bits of BLOCK3 (user). (RO)

**Register 4.39. EFUSE_RD_USR_DATA7_REG (0x0098)**




**EFUSE_USR_DATA7** Stores the seventh 32 bits of BLOCK3 (user). (RO)

**Register 4.40. EFUSE_RD_KEY0_DATA0_REG (0x009C)**

| EFUSE_KEY0_DATA0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY0_DATA0** Stores the zeroth 32 bits of KEY0. (RO)

**Register 4.41. EFUSE_RD_KEY0_DATA1_REG (0x00A0)**

| EFUSE_KEY0_DATA1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY0_DATA1** Stores the first 32 bits of KEY0. (RO)

**Register 4.42. EFUSE_RD_KEY0_DATA2_REG (0x00A4)**

| EFUSE_KEY0_DATA2 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

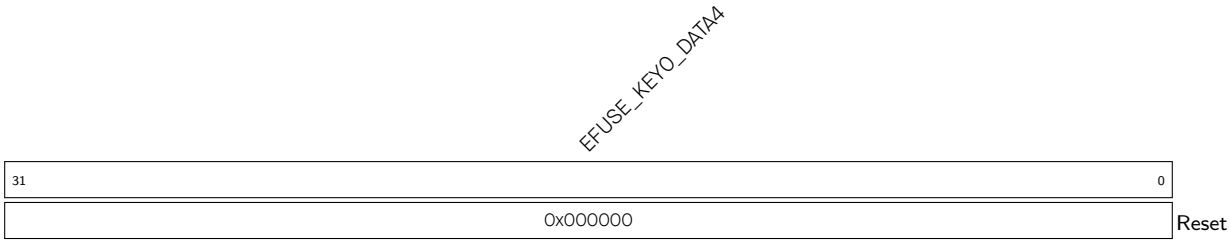
**EFUSE_KEY0_DATA2** Stores the second 32 bits of KEY0. (RO)

**Register 4.43. EFUSE_RD_KEY0_DATA3_REG (0x00A8)**

| EFUSE_KEY0_DATA3 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

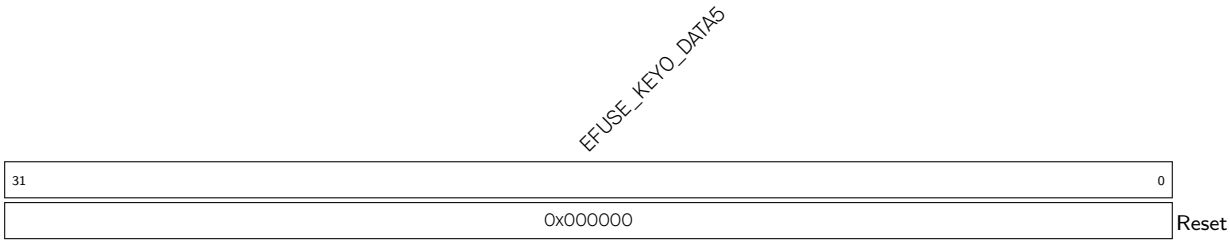
**EFUSE_KEY0_DATA3** Stores the third 32 bits of KEY0. (RO)

**Register 4.44. EFUSE_RD_KEY0_DATA4_REG (0x00AC)**



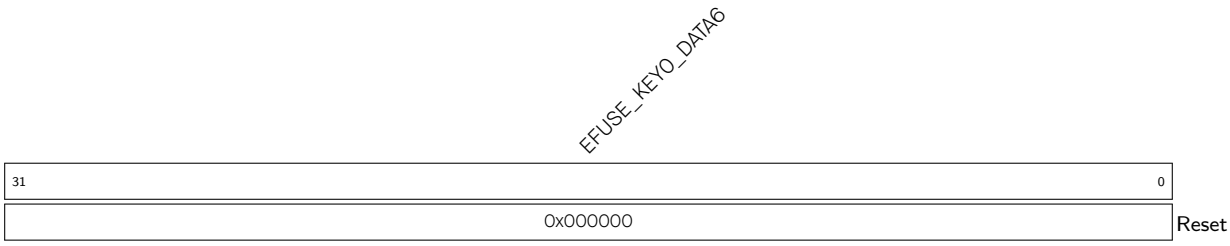

**EFUSE_KEY0_DATA4** Stores the fourth 32 bits of KEY0. (RO)

**Register 4.45. EFUSE_RD_KEY0_DATA5_REG (0x00B0)**



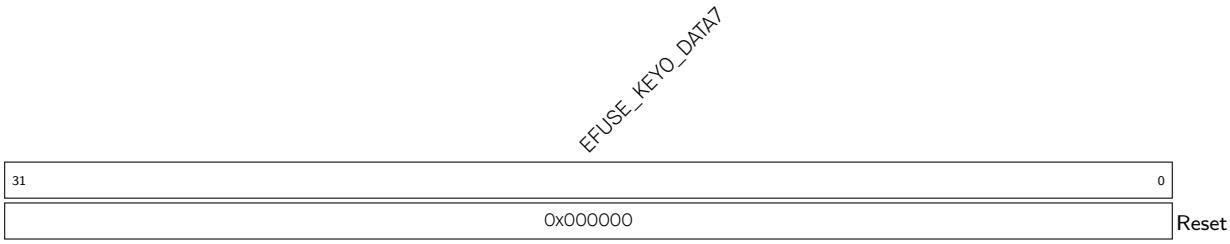

**EFUSE_KEY0_DATA5** Stores the fifth 32 bits of KEY0. (RO)

**Register 4.46. EFUSE_RD_KEY0_DATA6_REG (0x00B4)**




**EFUSE_KEY0_DATA6** Stores the sixth 32 bits of KEY0. (RO)

**Register 4.47. EFUSE_RD_KEY0_DATA7_REG (0x00B8)**




**EFUSE_KEY0_DATA7** Stores the seventh 32 bits of KEY0. (RO)

### Register 4.48. EFUSE_RD_KEY1_DATA0_REG (0x00BC)

| EFUSE_KEY1_DATA0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY1_DATA0** Stores the zeroth 32 bits of KEY1. (RO)

### Register 4.49. EFUSE_RD_KEY1_DATA1_REG (0x00C0)

| EFUSE_KEY1_DATA1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY1_DATA1** Stores the first 32 bits of KEY1. (RO)

### Register 4.50. EFUSE_RD_KEY1_DATA2_REG (0x00C4)

| EFUSE_KEY1_DATA2 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY1_DATA2** Stores the second 32 bits of KEY1. (RO)

### Register 4.51. EFUSE_RD_KEY1_DATA3_REG (0x00C8)

| EFUSE_KEY1_DATA3 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY1_DATA3** Stores the third 32 bits of KEY1. (RO)

### Register 4.52. EFUSE_RD_KEY1_DATA4_REG (0x00CC)

| EFUSE_KEY1_DATA4 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY1_DATA4** Stores the fourth 32 bits of KEY1. (RO)

### Register 4.53. EFUSE_RD_KEY1_DATA5_REG (0x00D0)

| EFUSE_KEY1_DATA5 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY1_DATA5** Stores the fifth 32 bits of KEY1. (RO)

### Register 4.54. EFUSE_RD_KEY1_DATA6_REG (0x00D4)

| EFUSE_KEY1_DATA6 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

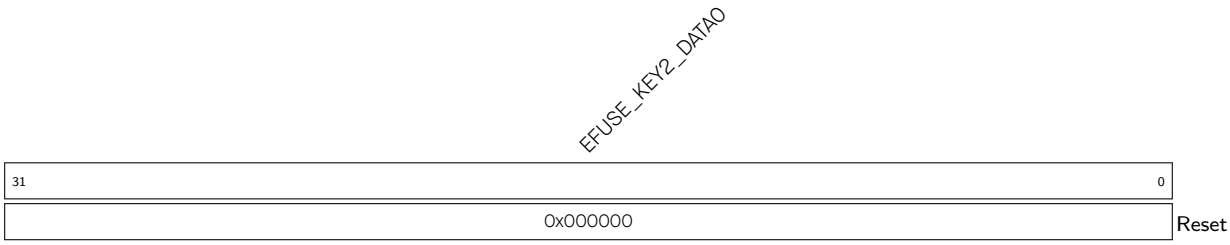
**EFUSE_KEY1_DATA6** Stores the sixth 32 bits of KEY1. (RO)

### Register 4.55. EFUSE_RD_KEY1_DATA7_REG (0x00D8)

| EFUSE_KEY1_DATA7 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

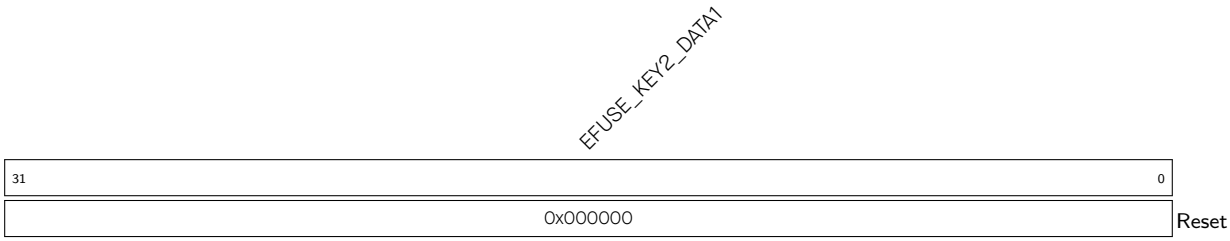
**EFUSE_KEY1_DATA7** Stores the seventh 32 bits of KEY1. (RO)

**Register 4.56. EFUSE_RD_KEY2_DATA0_REG (0x00DC)**



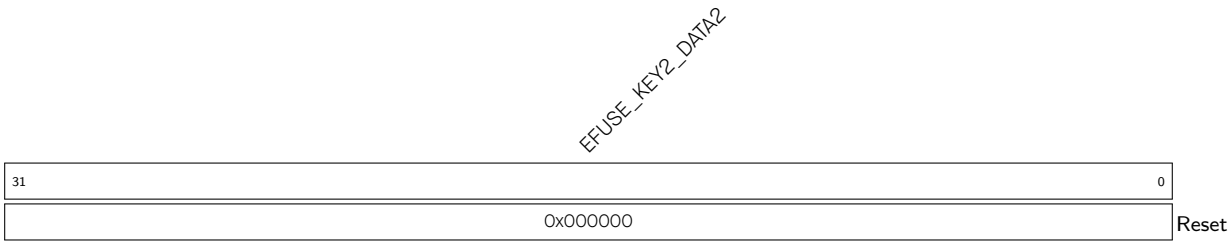

**EFUSE_KEY2_DATA0** Stores the zeroth 32 bits of KEY2. (RO)

**Register 4.57. EFUSE_RD_KEY2_DATA1_REG (0x00E0)**



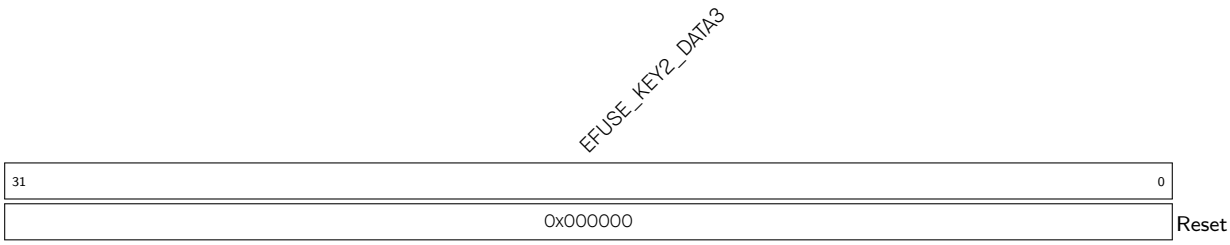

**EFUSE_KEY2_DATA1** Stores the first 32 bits of KEY2. (RO)

**Register 4.58. EFUSE_RD_KEY2_DATA2_REG (0x00E4)**



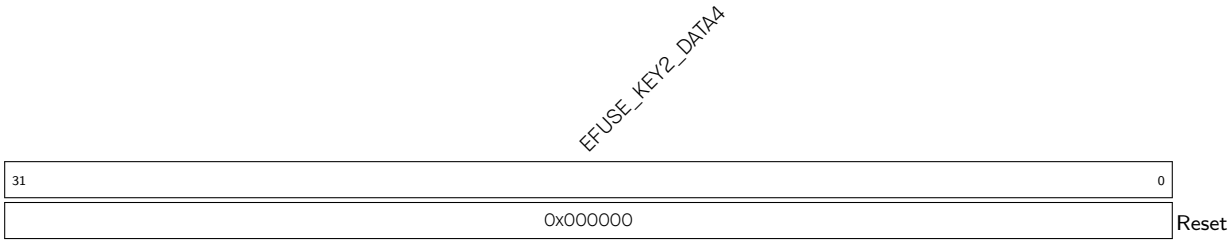

**EFUSE_KEY2_DATA2** Stores the second 32 bits of KEY2. (RO)

**Register 4.59. EFUSE_RD_KEY2_DATA3_REG (0x00E8)**



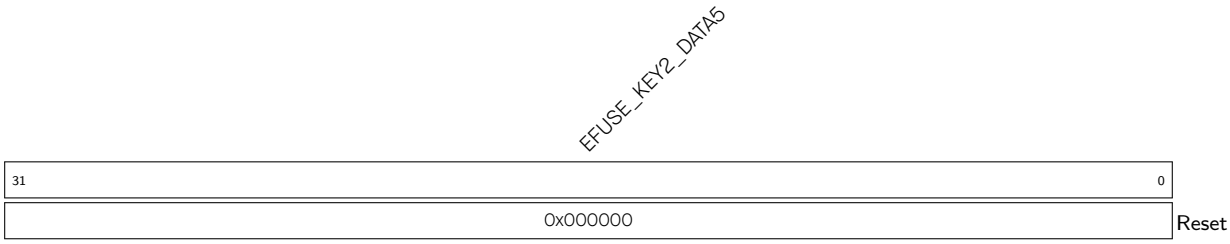

**EFUSE_KEY2_DATA3** Stores the third 32 bits of KEY2. (RO)

**Register 4.60. EFUSE_RD_KEY2_DATA4_REG (0x00EC)**



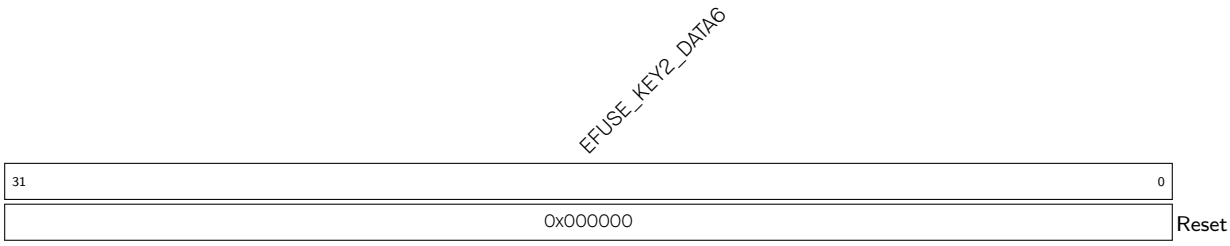

**EFUSE_KEY2_DATA4** Stores the fourth 32 bits of KEY2. (RO)

**Register 4.61. EFUSE_RD_KEY2_DATA5_REG (0x00F0)**



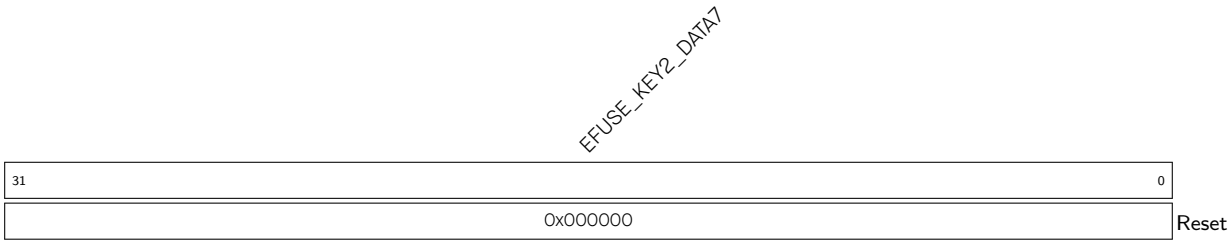

**EFUSE_KEY2_DATA5** Stores the fifth 32 bits of KEY2. (RO)

**Register 4.62. EFUSE_RD_KEY2_DATA6_REG (0x00F4)**



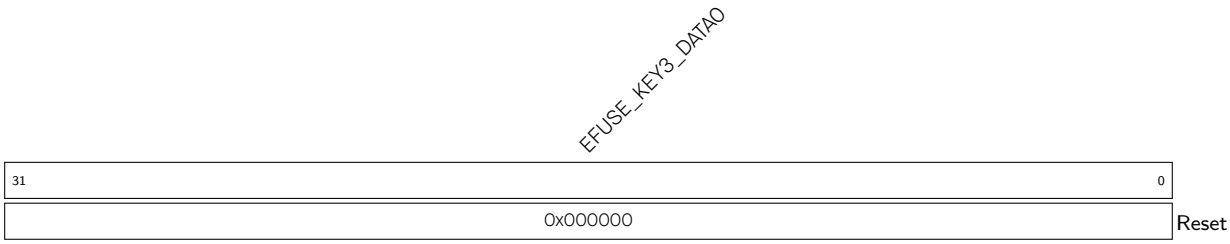

**EFUSE_KEY2_DATA6** Stores the sixth 32 bits of KEY2. (RO)

**Register 4.63. EFUSE_RD_KEY2_DATA7_REG (0x00F8)**



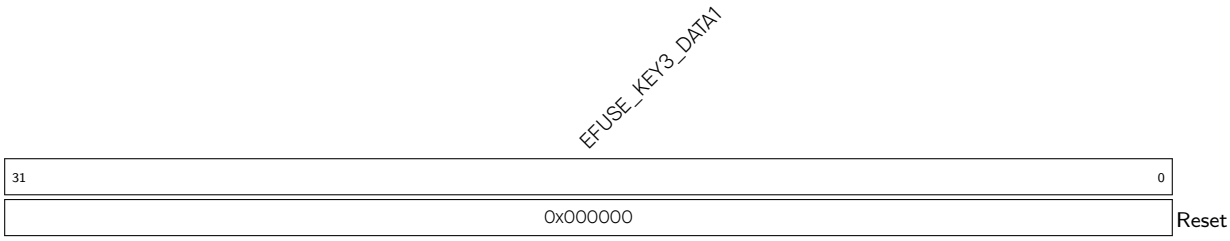

**EFUSE_KEY2_DATA7** Stores the seventh 32 bits of KEY2. (RO)

**Register 4.64. EFUSE_RD_KEY3_DATA0_REG (0x00FC)**



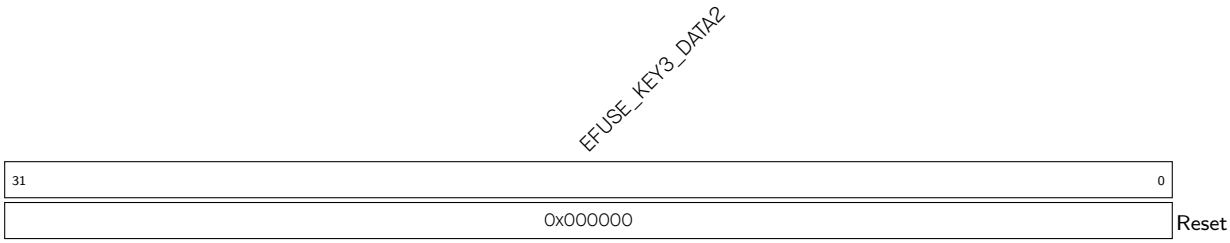

**EFUSE_KEY3_DATA0** Stores the zeroth 32 bits of KEY3. (RO)

**Register 4.65. EFUSE_RD_KEY3_DATA1_REG (0x0100)**



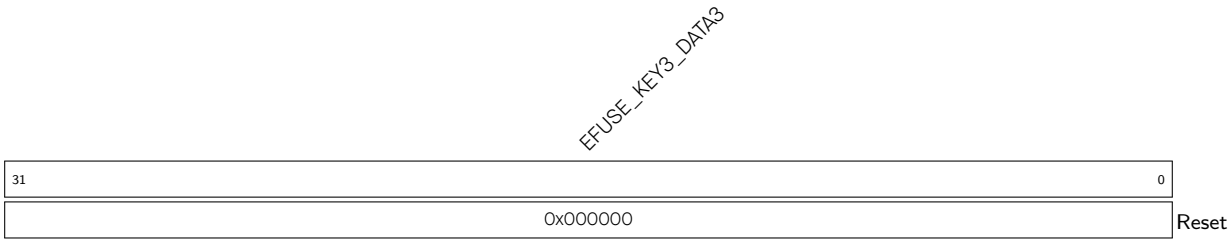

**EFUSE_KEY3_DATA1** Stores the first 32 bits of KEY3. (RO)

**Register 4.66. EFUSE_RD_KEY3_DATA2_REG (0x0104)**



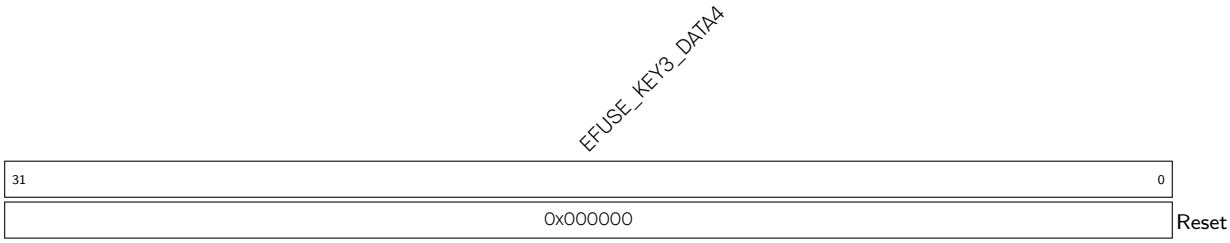

**EFUSE_KEY3_DATA2** Stores the second 32 bits of KEY3. (RO)

**Register 4.67. EFUSE_RD_KEY3_DATA3_REG (0x0108)**



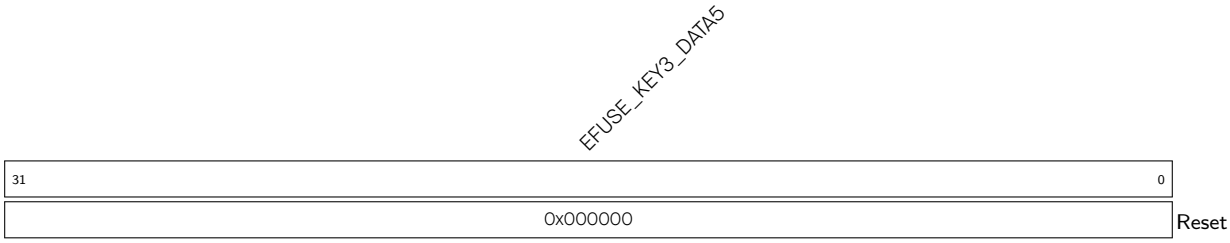

**EFUSE_KEY3_DATA3** Stores the third 32 bits of KEY3. (RO)

### Register 4.68. EFUSE_RD_KEY3_DATA4_REG (0x010C)



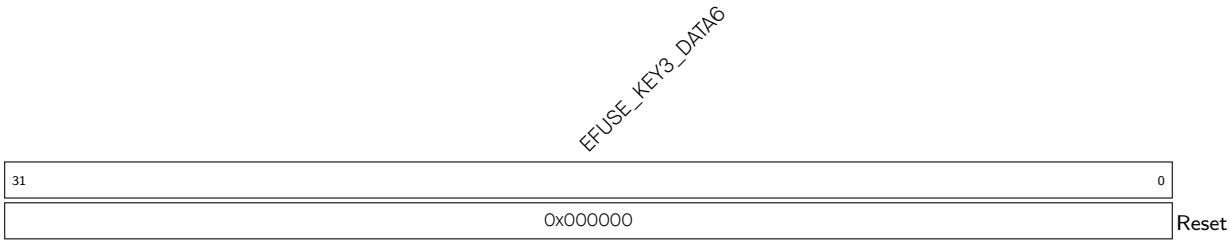

**EFUSE_KEY3_DATA4** Stores the fourth 32 bits of KEY3. (RO)

### Register 4.69. EFUSE_RD_KEY3_DATA5_REG (0x0110)



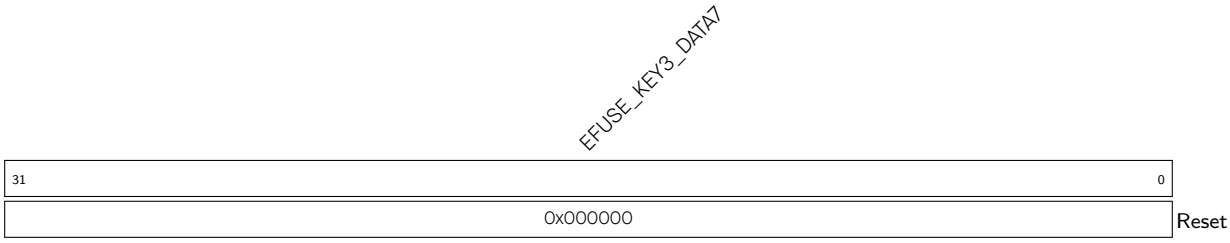

**EFUSE_KEY3_DATA5** Stores the fifth 32 bits of KEY3. (RO)

### Register 4.70. EFUSE_RD_KEY3_DATA6_REG (0x0114)



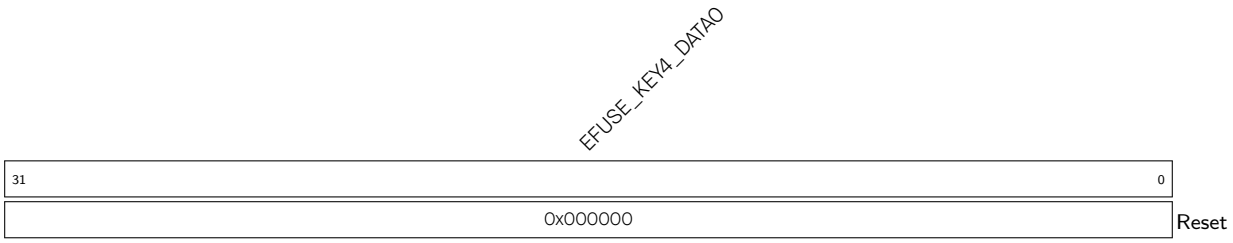

**EFUSE_KEY3_DATA6** Stores the sixth 32 bits of KEY3. (RO)

### Register 4.71. EFUSE_RD_KEY3_DATA7_REG (0x0118)



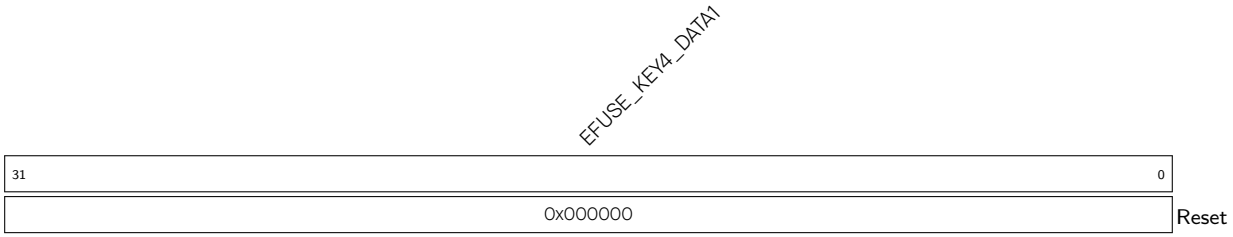

**EFUSE_KEY3_DATA7** Stores the seventh 32 bits of KEY3. (RO)

**Register 4.72. EFUSE_RD_KEY4_DATA0_REG (0x011C)**



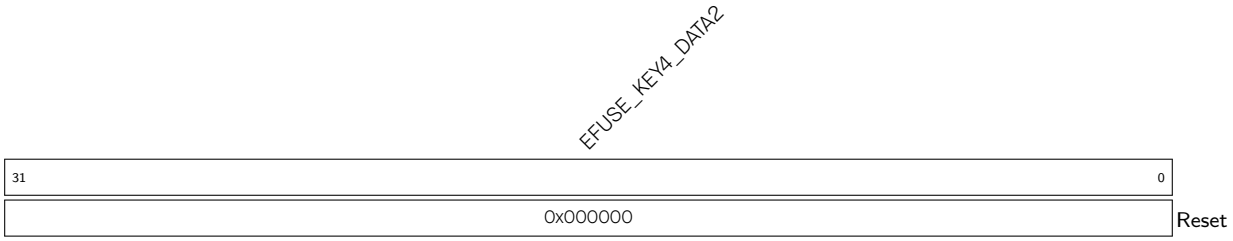

**EFUSE_KEY4_DATA0** Stores the zeroth 32 bits of KEY4. (RO)

**Register 4.73. EFUSE_RD_KEY4_DATA1_REG (0x0120)**



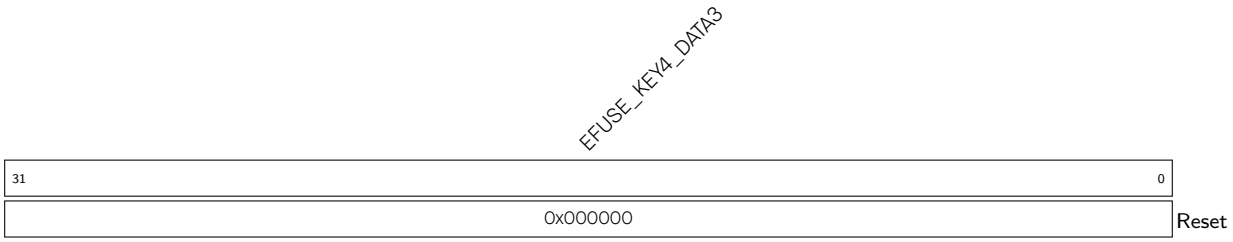

**EFUSE_KEY4_DATA1** Stores the first 32 bits of KEY4. (RO)

**Register 4.74. EFUSE_RD_KEY4_DATA2_REG (0x0124)**



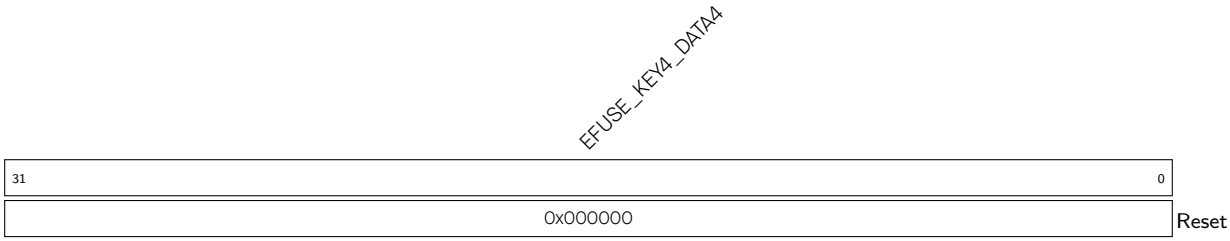

**EFUSE_KEY4_DATA2** Stores the second 32 bits of KEY4. (RO)

**Register 4.75. EFUSE_RD_KEY4_DATA3_REG (0x0128)**



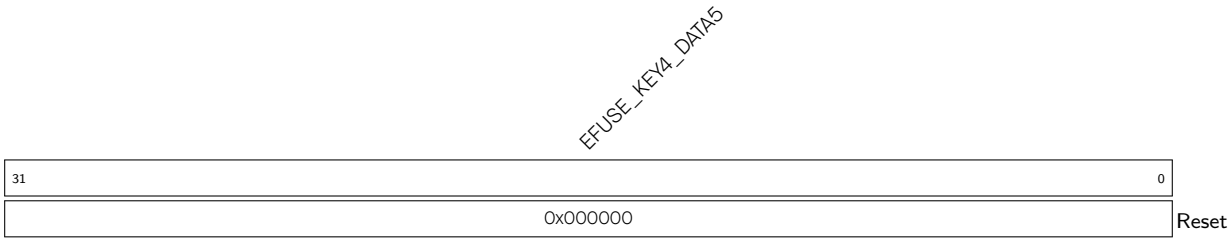

**EFUSE_KEY4_DATA3** Stores the third 32 bits of KEY4. (RO)

### Register 4.76. EFUSE_RD_KEY4_DATA4_REG (0x012C)



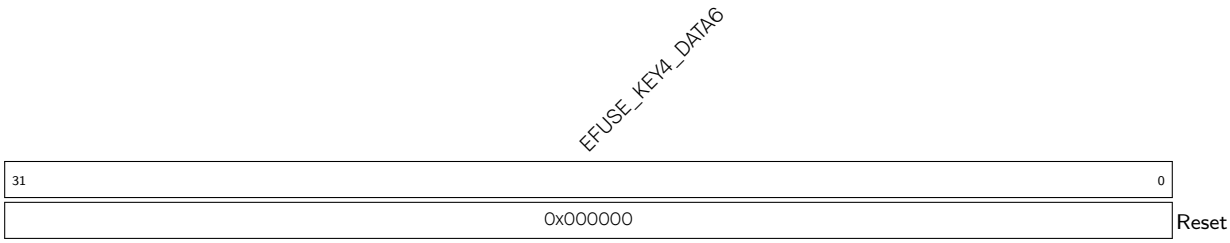

**EFUSE_KEY4_DATA4** Stores the fourth 32 bits of KEY4. (RO)

### Register 4.77. EFUSE_RD_KEY4_DATA5_REG (0x0130)



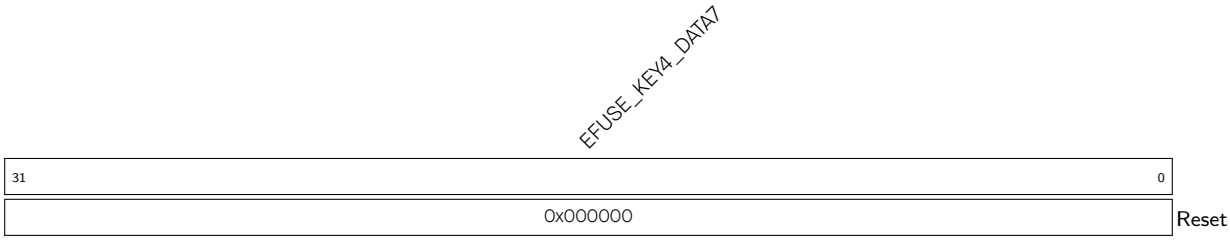

**EFUSE_KEY4_DATA5** Stores the fifth 32 bits of KEY4. (RO)

### Register 4.78. EFUSE_RD_KEY4_DATA6_REG (0x0134)




**EFUSE_KEY4_DATA6** Stores the sixth 32 bits of KEY4. (RO)

### Register 4.79. EFUSE_RD_KEY4_DATA7_REG (0x0138)




**EFUSE_KEY4_DATA7** Stores the seventh 32 bits of KEY4. (RO)

**Register 4.80. EFUSE_RD_KEY5_DATA0_REG (0x013C)**

| EFUSE_KEY5_DATA0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY5_DATA0** Stores the zeroth 32 bits of KEY5. (RO)

**Register 4.81. EFUSE_RD_KEY5_DATA1_REG (0x0140)**

| EFUSE_KEY5_DATA1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**EFUSE_KEY5_DATA1** Stores the first 32 bits of KEY5. (RO)

**Register 4.82. EFUSE_RD_KEY5_DATA2_REG (0x0144)**

| EFUSE_KEY5_DATA2 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

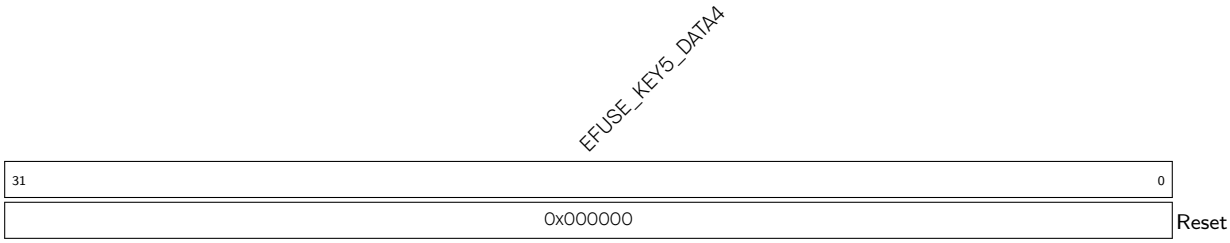
**EFUSE_KEY5_DATA2** Stores the second 32 bits of KEY5. (RO)

**Register 4.83. EFUSE_RD_KEY5_DATA3_REG (0x0148)**

| EFUSE_KEY5_DATA3 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

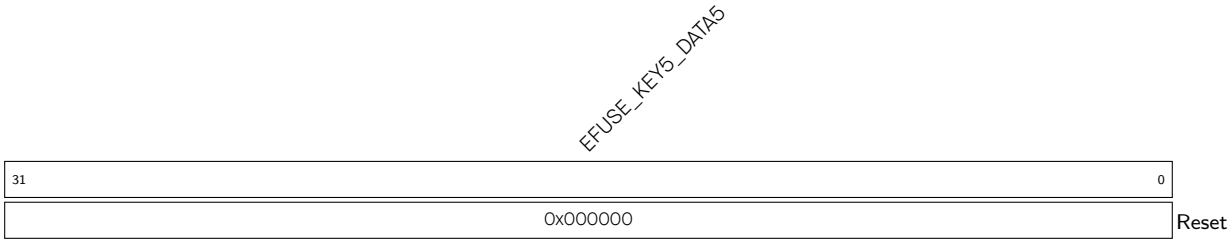
**EFUSE_KEY5_DATA3** Stores the third 32 bits of KEY5. (RO)

**Register 4.84. EFUSE_RD_KEY5_DATA4_REG (0x014C)**



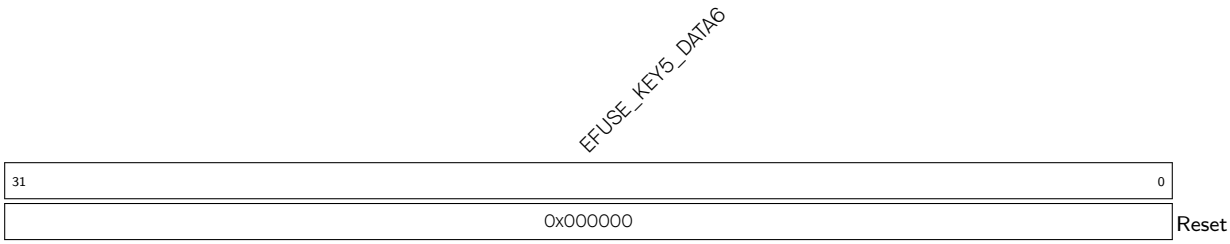

**EFUSE_KEY5_DATA4** Stores the fourth 32 bits of KEY5. (RO)

**Register 4.85. EFUSE_RD_KEY5_DATA5_REG (0x0150)**



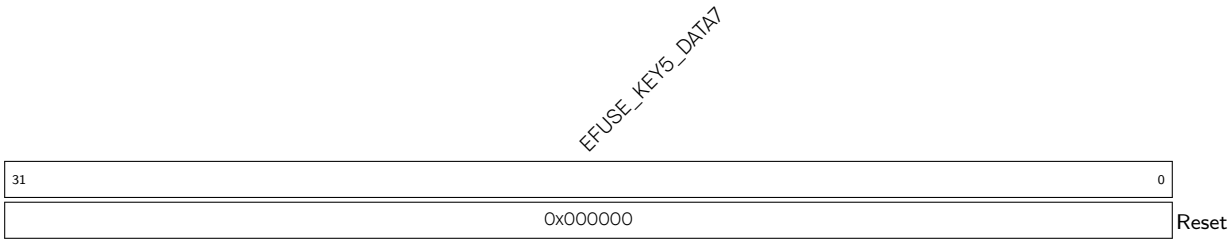

**EFUSE_KEY5_DATA5** Stores the fifth 32 bits of KEY5. (RO)

**Register 4.86. EFUSE_RD_KEY5_DATA6_REG (0x0154)**



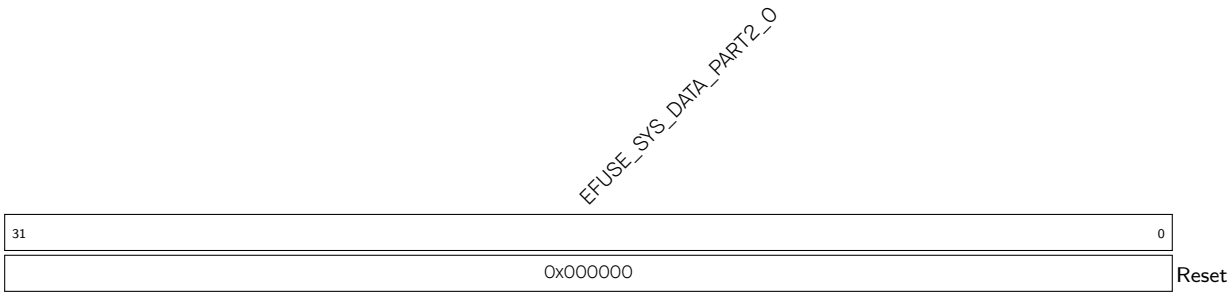

**EFUSE_KEY5_DATA6** Stores the sixth 32 bits of KEY5. (RO)

**Register 4.87. EFUSE_RD_KEY5_DATA7_REG (0x0158)**



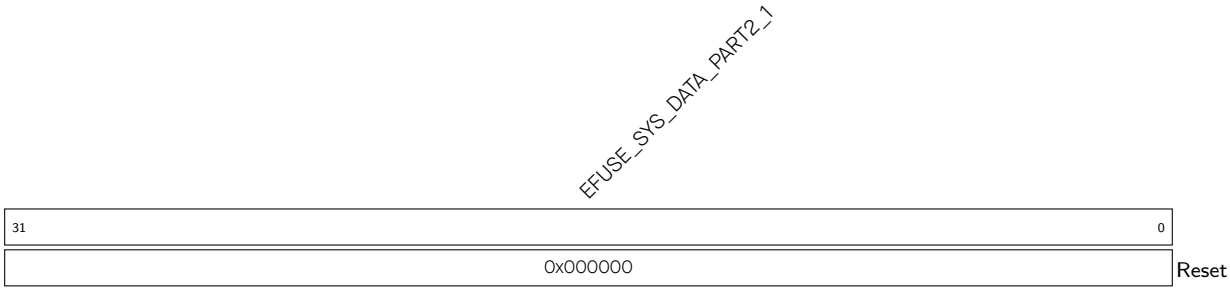

**EFUSE_KEY5_DATA7** Stores the seventh 32 bits of KEY5. (RO)

**Register 4.88. EFUSE_RD_SYS_PART2_DATA0_REG (0x015C)**



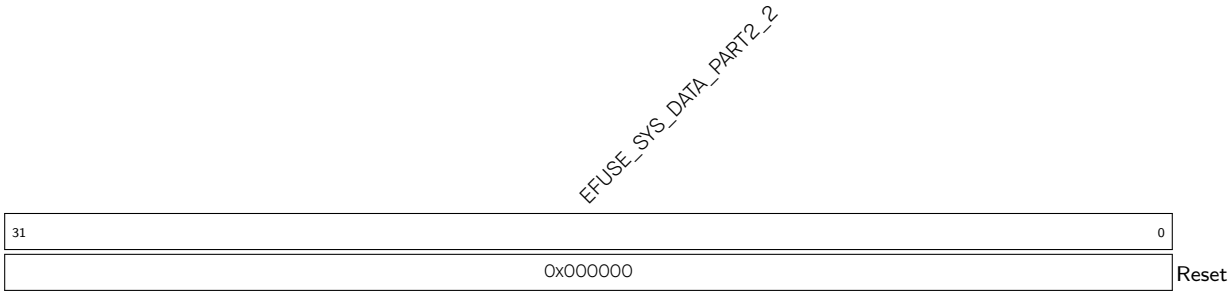

**EFUSE_SYS_DATA_PART2_0** Stores the 0th 32 bits of the 2nd part of system data. (RO)

**Register 4.89. EFUSE_RD_SYS_PART2_DATA1_REG (0x0160)**



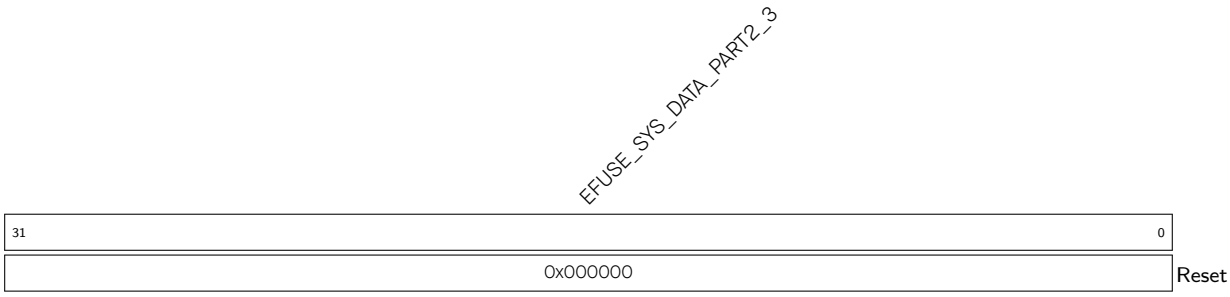

**EFUSE_SYS_DATA_PART2_1** Stores the 1st 32 bits of the 2nd part of system data. (RO)

**Register 4.90. EFUSE_RD_SYS_PART2_DATA2_REG (0x0164)**



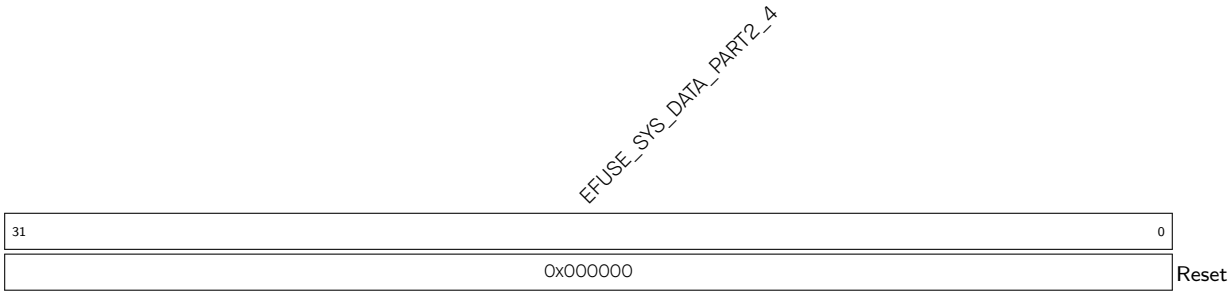

**EFUSE_SYS_DATA_PART2_2** Stores the 2nd 32 bits of the 2nd part of system data. (RO)

**Register 4.91. EFUSE_RD_SYS_PART2_DATA3_REG (0x0168)**



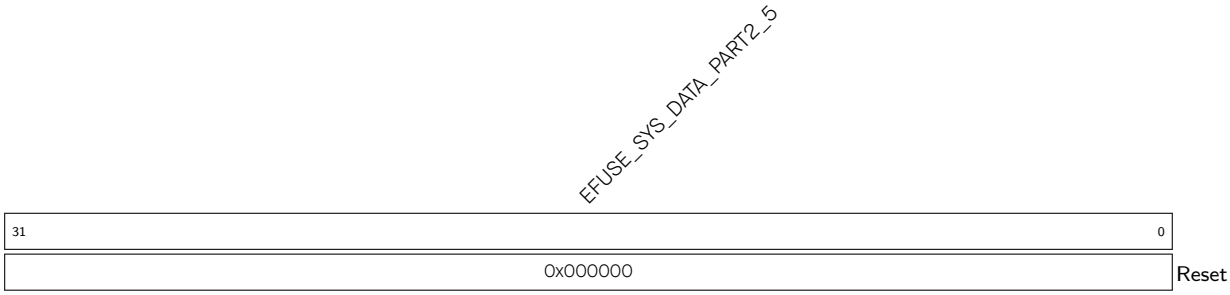

**EFUSE_SYS_DATA_PART2_3** Stores the 3rd 32 bits of the 2nd part of system data. (RO)

**Register 4.92. EFUSE_RD_SYS_PART2_DATA4_REG (0x016C)**




**EFUSE_SYS_DATA_PART2_4** Stores the 4th 32 bits of the 2nd part of system data. (RO)

**Register 4.93. EFUSE_RD_SYS_PART2_DATA5_REG (0x0170)**

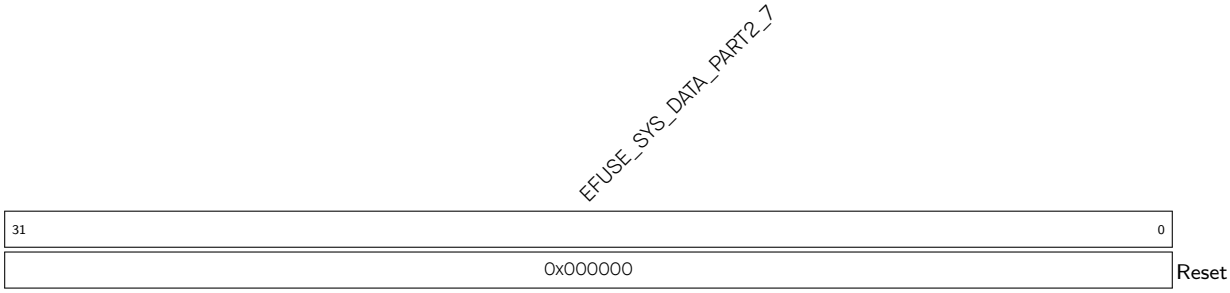



**EFUSE_SYS_DATA_PART2_5** Stores the 5th 32 bits of the 2nd part of system data. (RO)

**Register 4.94. EFUSE_RD_SYS_PART2_DATA6_REG (0x0174)**

**EFUSE_SYS_DATA_PART2_6** Stores the 6th 32 bits of the 2nd part of system data. (RO)

**Register 4.95. EFUSE_RD_SYS_PART2_DATA7_REG (0x0178)**




**EFUSE_SYS_DATA_PART2_7** Stores the 7th 32 bits of the 2nd part of system data. (RO)

**Register 4.96. EFUSE_RD_REPEAT_ERR0_REG (0x017C)**

| (reserved) | EFUSE_VDD_SPI_AS_GPIO_ERR | EFUSE_USB_EXCHG_PINS_ERR | (reserved) | EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT_ERR | EFUSE_DIS_PAD_JTAG_ERR | EFUSE_SOFT_DIS_JTAG_ERR | EFUSE_JTAG_SEL_ENABLE_ERR | EFUSE_DIS_TWAI_ERR | EFUSE_RPT4_RESERVED6_ERR | EFUSE_DIS_FORCE_DOWNLOAD_ERR | EFUSE_DIS_USB_SERIAL_JTAG_ERR | EFUSE_DIS_DOWNLOAD_ICACHE_ERR | EFUSE_DIS_USB_JTAG_ERR | EFUSE_DIS_ICACHE_ERR | EFUSE_DIS_RTC_RAM_BOOT_ERR | EFUSE_RD_DIS_ERR | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 27 | 26 | 25 | 24 21 | 20 | 19 | 18 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 0 | | |
| 0 0 0 0 0 | 0 | 0 | 0 0 0 0 | 0 | 0 | 0x0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x0 | Reset | |

**EFUSE_RD_DIS_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_RD_DIS. (RO)

**EFUSE_DIS_RTC_RAM_BOOT_ERR** Reserved. (RO)

**EFUSE_DIS_ICACHE_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_ICACHE. (RO)

**EFUSE_DIS_USB_JTAG_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_USB_JTAG. (RO)

**EFUSE_DIS_DOWNLOAD_ICACHE_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_DOWNLOAD_ICACHE. (RO)

**EFUSE_DIS_USB_SERIAL_JTAG_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_USB_SERIAL_JTAG. (RO)

**EFUSE_DIS_FORCE_DOWNLOAD_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_FORCE_DOWNLOAD. (RO)

**EFUSE_RPT4_RESERVED6_ERR** Reserved. (RO)

**EFUSE_DIS_TWAI_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_TWAI. (RO)

**EFUSE_JTAG_SEL_ENABLE_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_JTAG_SEL_ENABLE. (RO)

**EFUSE_SOFT_DIS_JTAG_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_SOFT_DIS_JTAG. (RO)

**EFUSE_DIS_PAD_JTAG_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_PAD_JTAG. (RO)

**EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT. (RO)

**EFUSE_USB_EXCHG_PINS_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_USB_EXCHG_PINS. (RO)

**EFUSE_VDD_SPI_AS_GPIO_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_VDD_SPI_AS_GPIO. (RO)

**Register 4.97. EFUSE_RD_REPEAT_ERR1_REG (0x0180)**

| EFUSE_KEY_PURPOSE_1_ERR | EFUSE_KEY_PURPOSE_0_ERR | EFUSE_SECURE_BOOT_KEY_REVOKE2_ERR | EFUSE_SECURE_BOOT_KEY_REVOKE1_ERR | EFUSE_SECURE_BOOT_KEY_REVOKE0_ERR | EFUSE_SPI_BOOT_CRYPT_CNT_ERR | EFUSE_WDT_DELAY_SEL_ERR | EFUSE_RPT4_RESERVED2_ERR | |
|---|---|---|---|---|---|---|---|---|
| 31 28 | 27 24 | 23 | 22 | 21 | 20 18 | 17 16 | 15 0 | |
| 0x0 | 0x0 | 0 | 0 | 0 | 0x0 | 0x0 | 0x00 | Reset |

**EFUSE_RPT4_RESERVED2_ERR** Reserved. (RO)

**EFUSE_WDT_DELAY_SEL_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_WDT_DELAY_SEL. (RO)

**EFUSE_SPI_BOOT_CRYPT_CNT_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_SPI_BOOT_CRYPT_CNT. (RO)

**EFUSE_SECURE_BOOT_KEY_REVOKE0_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_SECURE_BOOT_KEY_REVOKE0. (RO)

**EFUSE_SECURE_BOOT_KEY_REVOKE1_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_SECURE_BOOT_KEY_REVOKE1. (RO)

**EFUSE_SECURE_BOOT_KEY_REVOKE2_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_SECURE_BOOT_KEY_REVOKE2. (RO)

**EFUSE_KEY_PURPOSE_0_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_KEY_PURPOSE_0. (RO)

**EFUSE_KEY_PURPOSE_1_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_KEY_PURPOSE_1. (RO)

**Register 4.98. EFUSE_RD_REPEAT_ERR2_REG (0x0184)**

| EFUSE_FLASH_TPUW_ERR | EFUSE_RPT4_RESERVED0_ERR | EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE_ERR | EFUSE_SECURE_BOOT_EN_ERR | EFUSE_RPT4_RESERVED3_ERR | EFUSE_KEY_PURPOSE_5_ERR | EFUSE_KEY_PURPOSE_4_ERR | EFUSE_KEY_PURPOSE_3_ERR | EFUSE_KEY_PURPOSE_2_ERR | |
|---|---|---|---|---|---|---|---|---|---|
| 31 28 | 27 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 4 | 3 0 | |
| 0x0 | 0x0 | 0 | 0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | Reset |

**EFUSE_KEY_PURPOSE_2_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_KEY_PURPOSE_2. (RO)

**EFUSE_KEY_PURPOSE_3_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_KEY_PURPOSE_3. (RO)

**EFUSE_KEY_PURPOSE_4_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_KEY_PURPOSE_4. (RO)

**EFUSE_KEY_PURPOSE_5_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_KEY_PURPOSE_5. (RO)

**EFUSE_RPT4_RESERVED3_ERR** Reserved. (RO)

**EFUSE_SECURE_BOOT_EN_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_SECURE_BOOT_EN. (RO)

**EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_SECURE_BOOT_AGGRESSIVE_REVOKE. (RO)

**EFUSE_RPT4_RESERVED0_ERR** Reserved. (RO)

**EFUSE_FLASH_TPUW_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_FLASH_TPUW. (RO)

**Register 4.99. EFUSE_RD_REPEAT_ERR3_REG (0x0188)**

| 31 | 30 | 29 | 14 | 13 | 12 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EFUSE_ERR_RST_ENABLE_ERR | EFUSE_RPT4_RESERVED1_ERR | EFUSE_SECURE_VERSION_ERR | | EFUSE_FORCE_SEND_RESUME_ERR | EFUSE_RPT4_RESERVED5_ERR | | EFUSE_UART_PRINT_CONTROL_ERR | | EFUSE_ENABLE_SECURITY_DOWNLOAD_ERR | EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE_ERR | EFUSE_RPT4_RESERVED7_ERR | EFUSE_USB_PRINT_CHANNEL_ERR | EFUSE_RPT4_RESERVED8_ERR | EFUSE_DIS_DOWNLOAD_MODE_ERR | |
| 0 | 0 | 0x00 | | 0 | 0 | | 0x0 | | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**EFUSE_DIS_DOWNLOAD_MODE_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_DOWNLOAD_MODE. (RO)

**EFUSE_RPT4_RESERVED8_ERR** Reserved. (RO)

**EFUSE_USB_PRINT_CHANNEL_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_DOWNLOAD_MODE. (RO)

**EFUSE_RPT4_RESERVED7_ERR** Reserved. (RO)

**EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_DIS_USB_SERIAL_JTAG_DOWNLOAD_MODE. (RO)

**EFUSE_ENABLE_SECURITY_DOWNLOAD_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_ENABLE_SECURITY_DOWNLOAD. (RO)

**EFUSE_UART_PRINT_CONTROL_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_UART_PRINT_CONTROL. (RO)

**EFUSE_RPT4_RESERVED5_ERR** Reserved. (RO)

**EFUSE_FORCE_SEND_RESUME_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_FORCE_SEND_RESUME (RO)

**EFUSE_SECURE_VERSION_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_SECURE_VERSION. (RO)

**EFUSE_RPT4_RESERVED1_ERR** Reserved. (RO)

**EFUSE_ERR_RST_ENABLE_ERR** Any bit in this filed set to 1 indicates that an error occurs in programming EFUSE_ERR_RST_ENABLE. (RO)

**Register 4.100. EFUSE_RD_REPEAT_ERR4_REG (0x018C)**

| (reserved) | EFUSE_RPT4_RESERVED4_ERR | |
|---|---|---|
| 31 … 24 | 23 … 0 | |
| 0 0 0 0 0 0 0 0 | 0x0000 | Reset |

**EFUSE_RPT4_RESERVED4_ERR** Reserved. (RO)

**Register 4.101. EFUSE_RD_RS_ERR0_REG (0x01C0)**

| EFUSE_KEY3_FAIL | EFUSE_KEY4_ERR_NUM | EFUSE_KEY2_FAIL | EFUSE_KEY3_ERR_NUM | EFUSE_KEY1_FAIL | EFUSE_KEY2_ERR_NUM | EFUSE_KEY0_FAIL | EFUSE_KEY1_ERR_NUM | EFUSE_USR_DATA_FAIL | EFUSE_KEY0_ERR_NUM | EFUSE_SYS_PART1_FAIL | EFUSE_USR_DATA_ERR_NUM | EFUSE_MAC_SPI_8M_FAIL | EFUSE_SYS_PART1_NUM | (reserved) | EFUSE_MAC_SPI_8M_ERR_NUM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 | 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. (RO)

**EFUSE_SYS_PART1_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_MAC_SPI_8M_FAIL** 0: Means no failure and that the data of MAC_SPI_8M is reliable 1: Means that programming data of MAC_SPI_8M failed and the number of error bytes is over 6. (RO)

**EFUSE_USR_DATA_ERR_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_SYS_PART1_FAIL** 0: Means no failure and that the data of system part1 is reliable 1: Means that programming data of system part1 failed and the number of error bytes is over 6. (RO)

**EFUSE_KEY0_ERR_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_USR_DATA_FAIL** 0: Means no failure and that the user data is reliable 1: Means that programming user data failed and the number of error bytes is over 6. (RO)

**EFUSE_KEY1_ERR_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_KEY0_FAIL** 0: Means no failure and that the data of key0 is reliable 1: Means that programming key0 failed and the number of error bytes is over 6. (RO)

**EFUSE_KEY2_ERR_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_KEY1_FAIL** 0: Means no failure and that the data of key1 is reliable 1: Means that programming key1 failed and the number of error bytes is over 6. (RO)

**EFUSE_KEY3_ERR_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_KEY2_FAIL** 0: Means no failure and that the data of key2 is reliable 1: Means that programming key2 failed and the number of error bytes is over 6. (RO)

**EFUSE_KEY4_ERR_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_KEY3_FAIL** 0: Means no failure and that the data of key3 is reliable 1: Means that programming key3 failed and the number of error bytes is over 6. (RO)

**Register 4.102. EFUSE_RD_RS_ERR1_REG (0x01C4)**

| (reserved) | EFUSE_KEY5_FAIL | EFUSE_SYS_PART2_ERR_NUM | EFUSE_KEY4_FAIL | EFUSE_KEY5_ERR_NUM | |
|---|---|---|---|---|---|
| 31 … 8 | 7 | 6 … 4 | 3 | 2 … 0 | |
| 0 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 | 0 | 0x0 | Reset |

**EFUSE_KEY5_ERR_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_KEY4_FAIL** 0: Means no failure and that the data of KEY4 is reliable 1: Means that programming KEY4 data failed and the number of error bytes is over 6. (RO)

**EFUSE_SYS_PART2_ERR_NUM** The value of this signal means the number of error bytes. (RO)

**EFUSE_KEY5_FAIL** 0: Means no failure and that the data of KEY5 is reliable 1: Means that programming KEY5 data failed and the number of error bytes is over 6. (RO)

**Register 4.103. EFUSE_CLK_REG (0x01C8)**

| (reserved) | EFUSE_CLK_EN | (reserved) | EFUSE_EFUSE_MEM_FORCE_PU | EFUSE_MEM_CLK_FORCE_ON | EFUSE_EFUSE_MEM_FORCE_PD | |
|---|---|---|---|---|---|---|
| 31 … 17 | 16 | 15 … 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 | 1 | 0 | Reset |

**EFUSE_EFUSE_MEM_FORCE_PD** Set this bit to force eFuse SRAM into power-saving mode. (R/W)

**EFUSE_MEM_CLK_FORCE_ON** Set this bit and force to activate clock signal of eFuse SRAM. (R/W)

**EFUSE_EFUSE_MEM_FORCE_PU** Set this bit to force eFuse SRAM into working mode. (R/W)

**EFUSE_CLK_EN** Set this bit and force to enable clock signal of eFuse memory. (R/W)

**Register 4.104. EFUSE_CONF_REG (0x01CC)**

| (reserved) | EFUSE_OP_CODE | |
|---|---|---|
| 31 … 16 | 15 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x00 | Reset |

**EFUSE_OP_CODE** 0x5A5A: Operate programming command 0x5AA5: Operate read command. (R/W)

**Register 4.105. EFUSE_CMD_REG (0x01D4)**

| (reserved) | EFUSE_BLK_NUM | EFUSE_PGM_CMD | EFUSE_READ_CMD | |
|---|---|---|---|---|
| 31 … 6 | 5 … 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 | 0x0 | 0 | 0 | Reset |

**EFUSE_READ_CMD** Set this bit to send read command. (R/WS/SC)

**EFUSE_PGM_CMD** Set this bit to send programming command. (R/WS/SC)

**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 4.106. EFUSE_DAC_CONF_REG (0x01E8)**

| (reserved) | EFUSE_OE_CLR | EFUSE_DAC_NUM | EFUSE_DAC_CLK_PAD_SEL | EFUSE_DAC_CLK_DIV | |
|---|---|---|---|---|---|
| 31 … 18 | 17 | 16 … 9 | 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 255 | 0 | 28 | Reset |

**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 4.107. EFUSE_RD_TIM_CONF_REG (0x01EC)

| EFUSE_READ_INIT_NUM | (reserved) | |
|---|---|---|
| 31 … 24 | 23 … 0 | |
| 0x12 | 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 |

**EFUSE_READ_INIT_NUM** Configures the initial read time of eFuse. (R/W)

Register 4.108. EFUSE_WR_TIM_CONF1_REG (0x01F0)

| (reserved) | EFUSE_PWR_ON_NUM | (reserved) | |
|---|---|---|---|
| 31 … 24 | 23 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 | 0x2880 | 0 0 0 0 0 0 0 0 | Reset |

**EFUSE_PWR_ON_NUM** Configures the power up time for VDDQ. (R/W)

Register 4.109. EFUSE_WR_TIM_CONF2_REG (0x01F4)

| (reserved) | EFUSE_PWR_OFF_NUM | |
|---|---|---|
| 31 … 16 | 15 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x190 | Reset |

**EFUSE_PWR_OFF_NUM** Configures the power outage time for VDDQ. (R/W)

**Register 4.110. EFUSE_STATUS_REG (0x01D0)**

| (reserved) | EFUSE_REPEAT_ERR_CNT | (reserved) | EFUSE_STATE | |
|---|---|---|---|---|
| 31 … 18 | 17 … 10 | 9 … 4 | 3 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | 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 4.111. EFUSE_INT_RAW_REG (0x01D8)**

| (reserved) | EFUSE_PGM_DONE_INT_RAW | EFUSE_READ_DONE_INT_RAW | |
|---|---|---|---|
| 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 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. (R/WC/SS)

**EFUSE_PGM_DONE_INT_RAW** The raw bit signal for pgm_done interrupt. (R/WC/SS)

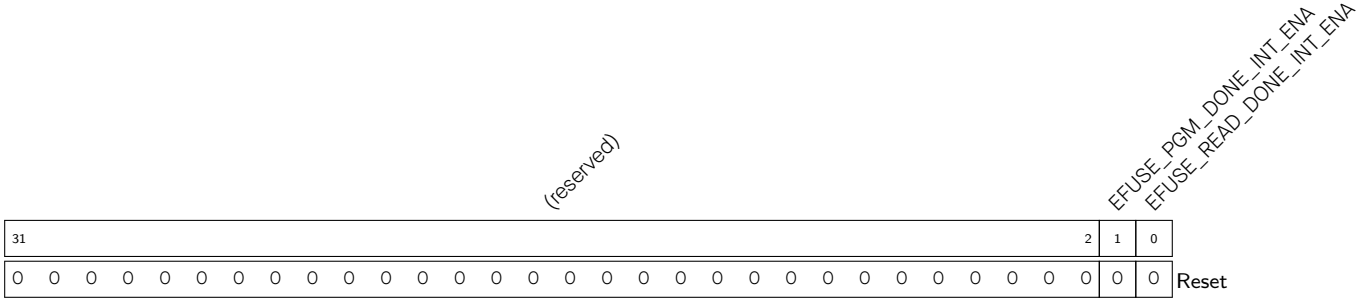
**Register 4.112. EFUSE_INT_ST_REG (0x01DC)**

| (reserved) | EFUSE_PGM_DONE_INT_ST | EFUSE_READ_DONE_INT_ST | |
|---|---|---|---|
| 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 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)

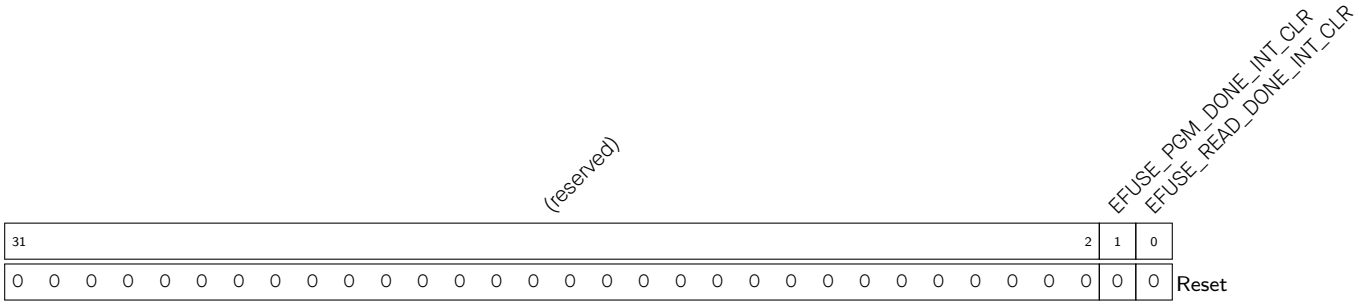
### Register 4.113. EFUSE_INT_ENA_REG (0x01E0)




**EFUSE_READ_DONE_INT_ENA** The enable signal for read_done interrupt. (R/W)

**EFUSE_PGM_DONE_INT_ENA** The enable signal for pgm_done interrupt. (R/W)

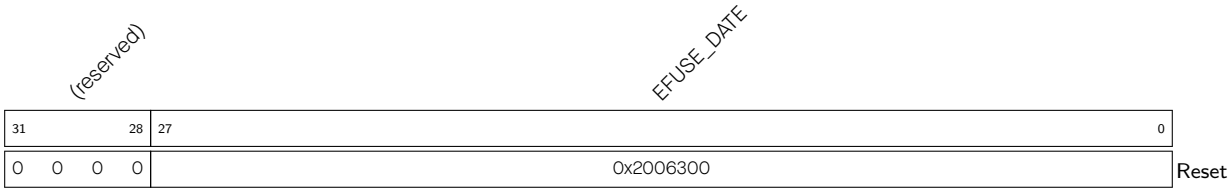
### Register 4.114. EFUSE_INT_CLR_REG (0x01E4)




**EFUSE_READ_DONE_INT_CLR** The clear signal for read_done interrupt. (WO)

**EFUSE_PGM_DONE_INT_CLR** The clear signal for pgm_done interrupt. (WO)

### Register 4.115. EFUSE_DATE_REG (0x01FC)




**EFUSE_DATE** Stores eFuse version. (R/W)

# Part III

# System Component

Encompassing a range of system-level functionalities, this part describes components related to system boot, clocks, GPIO, timers, watchdogs, debug assistance, low-power management, and various system registers.

# Chapter 5

# IO MUX and GPIO Matrix (GPIO, IO MUX)

## 5.1 Overview

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

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

## 5.2 Features

**GPIO Matrix Features**

- A full-switching matrix between the peripheral input/output signals and the pins.
- 42 peripheral input signals can be sourced from the input of any GPIO pins.
- The output of any GPIO pins can be from any of the 78 peripheral output signals.
- Supports signal synchronization for peripheral inputs based on APB clock bus.
- Provides input signal filter.
- Supports sigma delta modulated output.
- Supports GPIO simple input and output.

**IO MUX Features**

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

## 5.3 Architectural Overview

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

- Figure 5.3-1 shows the general work flow of IO MUX and GPIO matrix.

- Figure 5.3-2 shows in details how IO MUX and GPIO matrix route signals from pins to peripherals, and from peripherals to pins.
- Figure 5.3-3 shows the interface logic for a GPIO pin.

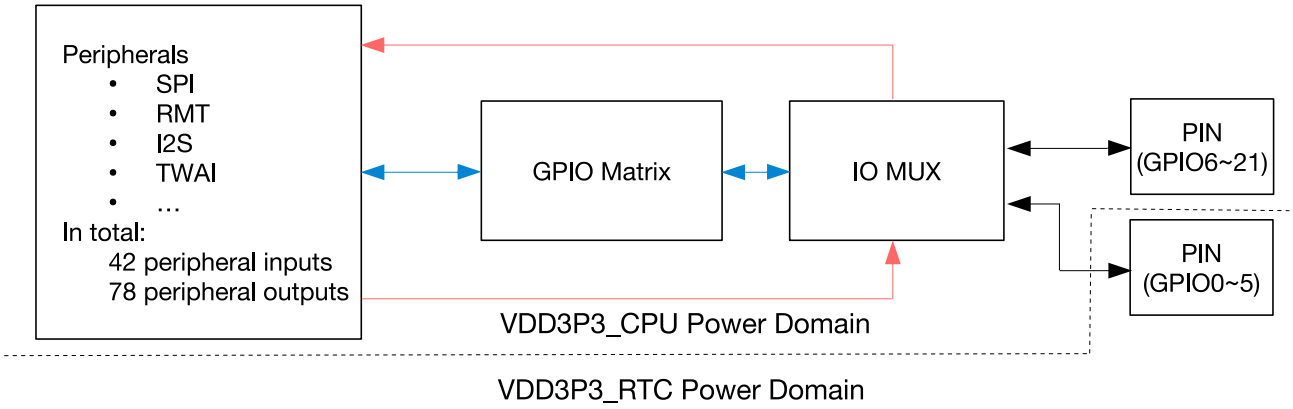



Figure 5.3-1. Diagram of IO MUX and GPIO Matrix

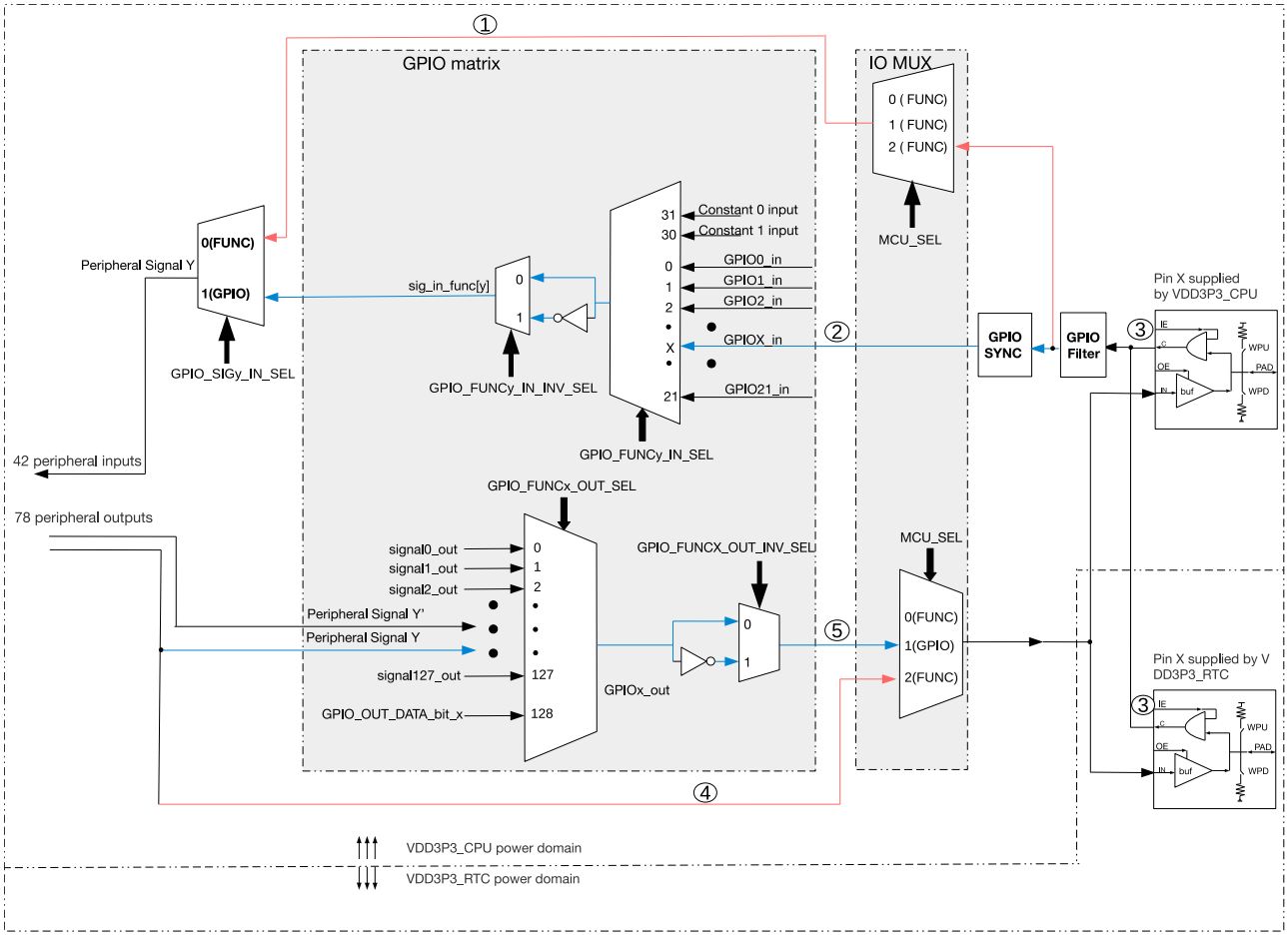



Figure 5.3-2. Architecture of IO MUX and GPIO Matrix

① Only part of peripheral input signals (marked "yes" in column "Direct input through IO MUX" in Table 5.11-1) can bypass GPIO matrix. The other input signals can only be routed to peripherals via GPIO matrix.

② There are only 22 inputs from GPIO SYNC to GPIO matrix, since ESP32-C3 provides 22 GPIO pins in

total.

③ The pins supplied by VDD3P3_CPU or by VDD3P3_RTC are controlled by the signals: IE, OE, WPU, and WPD.

④ Only part of peripheral outputs (marked "yes" in column "Direct output through IO MUX" in Table 5.11-1) can be routed to pins bypassing GPIO matrix. See Table 5.11-1.

⑤ There are only 22 outputs (GPIO pin X: 0 ~ 21) from GPIO matrix to IO MUX.

Figure 5.3-3 shows the internal structure of a pad, which is an electrical interface between the chip logic and the GPIO pin. The structure is applicable to all 22 GPIO pins and can be controlled using IE, OE, WPU, and WPD signals.

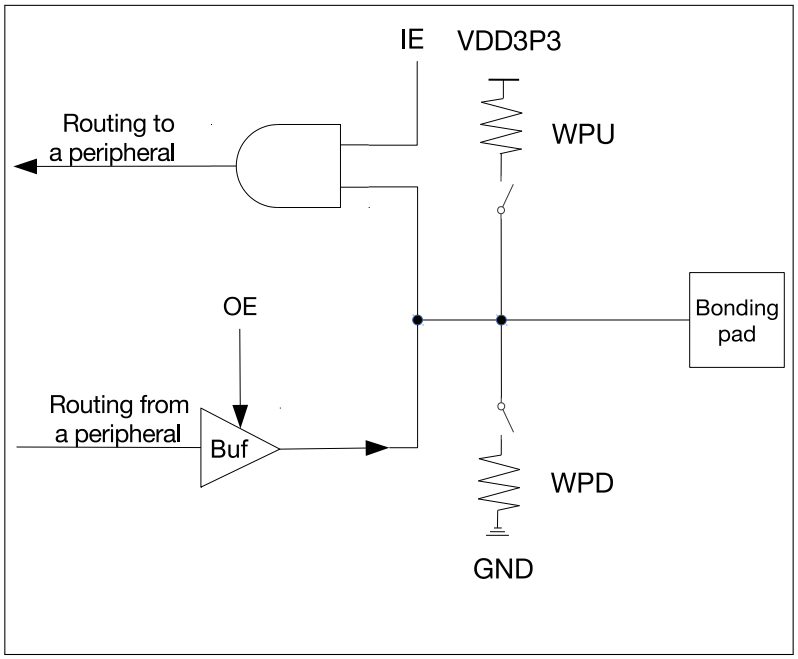



Figure 5.3-3. Internal Structure of a Pad

**Note:**

- IE: input enable
- OE: output enable
- WPU: internal weak pull-up
- WPD: internal weak pull-down
- Bonding pad: a terminal point of the chip logic used to make a physical connection from the chip die to GPIO pin in the chip package.

## 5.4 Peripheral Input via GPIO Matrix

### 5.4.1 Overview

To receive a peripheral input signal via GPIO matrix, the matrix is configured to source the peripheral input signal from one of the 22 GPIOs (0 ~ 21), see Table 5.11-1. Meanwhile, register corresponding to the peripheral signal should be set to receive input signal via GPIO matrix.

## 5.4.2 Signal Synchronization

When signals are directed from pins using GPIO matrix, the signals will be synchronized to the APB bus clock by GPIO SYNC hardware, then go to GPIO matrix. This synchronization applies to all GPIO matrix signals but does not apply when using the IO MUX, see Figure 5.3-2.

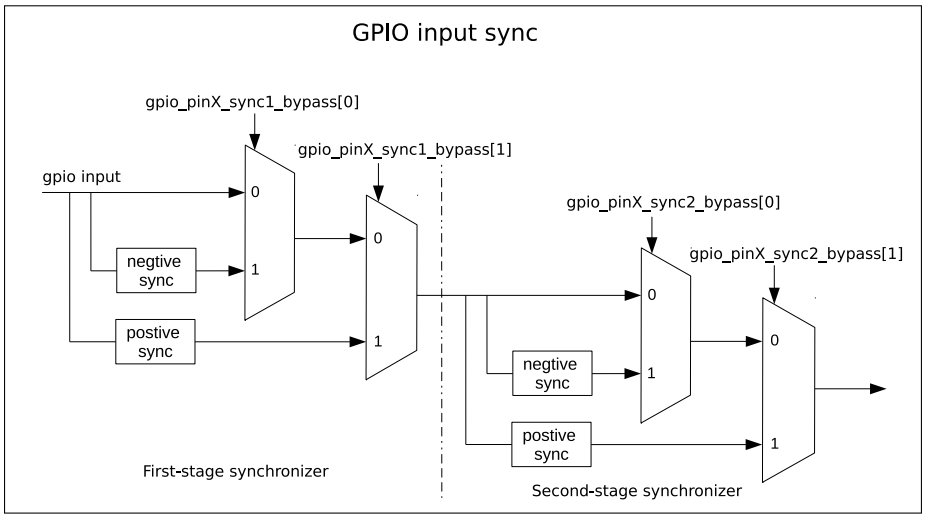



Figure 5.4-1. GPIO Input Synchronized on APB Clock Rising Edge or on Falling Edge

Figure 5.4-1 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.

## 5.4.3 Functional Description

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

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

   **Note that** some peripheral signals have no valid GPIO_SIG*y*_IN_SEL bit, namely, these peripherals can only receive input signals via GPIO matrix.
2. Optionally enable the filter for pin input signals by setting the register IO_MUX_GPIO*n*_FILTER_EN. Only the signals with a valid width of more than two clock cycles can be sampled, see Figure 5.4-2.

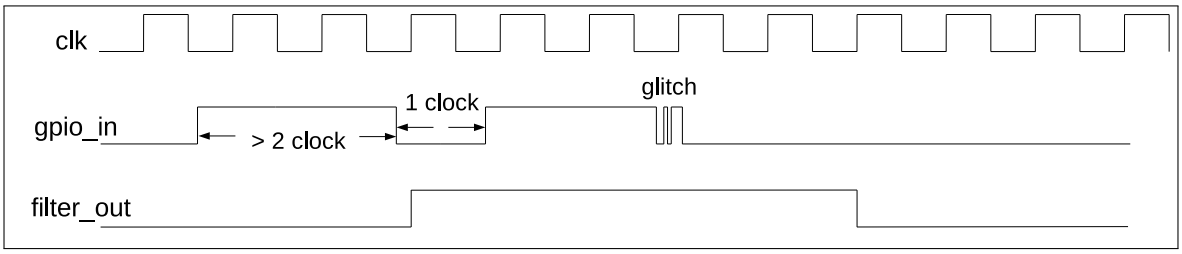



Figure 5.4-2. Filter Timing of GPIO Input Signals

3. Synchronize GPIO input. To do so, please set GPIO_PIN*x*_REG corresponding to GPIO pin *X* as follows:

- Set GPIO_PINx_SYNC1_BYPASS to enable input signal synchronized on rising edge or on falling edge in the first clock, see Figure 5.4-1.
- Set GPIO_PINx_SYNC2_BYPASS to enable input signal synchronized on rising edge or on falling edge in the second clock, see Figure 5.4-1.

4. Configure IO MUX register to enable pin input. For this end, please set IO_MUX_GPIOx_REG corresponding to GPIO pin *X* as follows:
   - Set IO_MUX_GPIOx_FUN_IE to enable input[2].
   - Set or clear IO_MUX_GPIOx_FUN_WPU and IO_MUX_GPIOx_FUN_WPD, as desired, to enable or disable pull-up and pull-down resistors.

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

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

**Note:**

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

### 5.4.4 Simple GPIO Input

GPIO_IN_REG holds the input values of each GPIO pin. The input value of any GPIO pin can be read at any time without configuring GPIO matrix for a particular peripheral signal. However, it is necessary to enable the input via IO MUX by setting IO_MUX_GPIOx_FUN_IE bit in register IO_MUX_GPIOx_REG corresponding to pin *X*, as mentioned in Section 5.4.2.

## 5.5 Peripheral Output via GPIO Matrix

### 5.5.1 Overview

To output a signal from a peripheral via GPIO matrix, the matrix is configured to route peripheral output signals (only signals with a name assigned in the column "Output signal" in Table 5.11-1) to one of the 22 GPIOs (0 ~ 21). See Table 5.11-1.

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

the pin.

**Note:**
There is a range of peripheral output signals (97 ~ 100) which are not connected to any peripheral, but to the input signals (97 ~ 100 in Table 5.11-1) directly. These can be used to input a signal from one GPIO pin and output directly to another GPIO pin.

### 5.5.2 Functional Description

Some of the 78 output signals (signals with a name assigned in the column "Output signal" in Table 5.11-1) can be set to go through GPIO matrix into IO MUX and then to a pin. Figure 5.3-2 illustrates the configuration.

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

1. Configure register GPIO_FUNC*x*_OUT_SEL_CFG_REG and GPIO_ENABLE_REG[*x*] corresponding to GPIO pin *X* in GPIO matrix. Recommended operation: use corresponding W1TS (write 1 to set) and W1TC (write 1 to clear) registers to set or clear GPIO_ENABLE_REG.
   - Set the GPIO_FUNC*x*_OUT_SEL field in register GPIO_FUNC*x*_OUT_SEL_CFG_REG to the index of the desired peripheral output signal *Y*.
   - If the signal should always be enabled as an output, set the GPIO_FUNC*x*_OEN_SEL bit in register GPIO_FUNC*x*_OUT_SEL_CFG_REG and the bit in register GPIO_ENABLE_W1TS_REG, corresponding to GPIO pin *X*. To have the output enable signal decided by internal logic (for example, the SPIQ_oe in column "Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0" in Table 5.11-1), clear GPIO_FUNC*x*
     _OEN_SEL bit instead.
   - Set the corresponding bit in register GPIO_ENABLE_W1TC_REG to disable the output from the GPIO pin.
2. For an open drain output, set the GPIO_PIN*x*_PAD_DRIVER bit in register GPIO_PIN*x*_REG corresponding to GPIO pin *X*.
3. Configure IO MUX register to enable output via GPIO matrix. Set the IO_MUX_GPIO*x*_REG corresponding to GPIO pin *X* as follows:
   - Set the field IO_MUX_GPIO*x*_MCU_SEL to desired IO MUX function corresponding to GPIO pin*X*. This is Function 1 (GPIO function), numeric value 1, for all pins.
   - Set the IO_MUX_GPIO*x*_FUN_DRV field to the desired value for output strength (0 ~ 3).
   - If using open drain mode, set/clear the IO_MUX_GPIO*x*_FUN_WPU and IO_MUX_GPIO*x*_FUN_WPD bits to enable/disable the internal pull-up/pull-down resistors.

**Note:**

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

### 5.5.3 Simple GPIO Output

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

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

**Note:**

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

### 5.5.4 Sigma Delta Modulated Output (SDM)

#### 5.5.4.1 Functional Description

Four out of the 125 peripheral outputs (output index: 55 ~ 58 in Table 5.11-1) support 1-bit second-order sigma delta modulation. By default output is enabled for these four channels. This modulator can also output PDM (pulse density modulation) signal with configurable duty cycle. The transfer function of this second-order SDM modulator is:

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

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_SD*n*_PRESCALE (*n* is 0 ~ 3 for four channels).

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

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

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

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

$$Duty_Cycle = \frac{GPIOSD_SDn_IN + 128}{256}$$

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

#### 5.5.4.2 SDM Configuration

The configuration of SDM is shown below:

- Route one of SDM outputs to a pin via GPIO matrix, see Section 5.5.2.
- Enable the modulator clock by setting the register GPIOSD_FUNCTION_CLK_EN.
- Configure the divider value by setting the register GPIOSD_SD*n*_PRESCALE.
- Configure the duty cycle of SDM output signal by setting the register GPIOSD_SD*n*_IN.

## 5.6 Direct Input and Output via IO MUX

### 5.6.1 Overview

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

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

### 5.6.2 Functional Description

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

1. IO_MUX_GPIO*n*_MCU_SEL for the GPIO pin must be set to the required pin function. For the list of pin functions, please refer to Section 5.12.
2. Clear GPIO_SIG*n*_IN_SEL to route the input directly to the peripheral.

To bypass GPIO matrix for peripheral output signals, IO_MUX_GPIO*n*_MCU_SEL for the GPIO pin must be set to the required pin function. For the list of pin functions, please refer to Section 5.12.

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

## 5.7 Analog Functions of GPIO Pins

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

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

See Table 5.13-1 for analog functions of ESP32-C3 pins.

## 5.8 Pin Functions in Light-sleep

Pins may provide different functions when ESP32-C3 is in Light-sleep mode. If IO_MUX_SLP_SEL in register IO_MUX_*n*_REG for a GPIO pin is set to 1, a different set of bits will be used to control the pin when the chip is in Light-sleep mode.

**Table 5.8-1. Bits Used to Control IO MUX Functions in Light-sleep Mode**

| IO MUX Functions | Normal Execution<br>OR IO_MUX_SLP_SEL = 0 | Light-sleep Mode<br>AND IO_MUX_SLP_SEL = 1 |
|---|---|---|
| Output Drive Strength | IO_MUX_FUN_DRV | IO_MUX_MCU_DRV |
| Pull-up Resistor | IO_MUX_FUN_WPU | IO_MUX_MCU_WPU |
| Pull-down Resistor | IO_MUX_FUN_WPD | IO_MUX_MCU_WPD |
| Output Enable | OEN_SEL from GPIO matrix * | IO_MUX_MCU_OE |

**Note:**
If IO_MUX_SLP_SEL is set to 0, pin functions remain the same in both normal execution and Light-sleep mode. Please refer to Section 5.5.2 for how to enable output in normal execution.

## 5.9 Pin Hold Feature

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

**Note:**

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

## 5.10 Power Supplies and Management of GPIO Pins

### 5.10.1 Power Supplies of GPIO Pins

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

### 5.10.2 Power Supply Management

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

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

## 5.11 Peripheral Signal List

Table 5.11-1 shows the peripheral input/output signals via GPIO matrix.

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

- GPIO_FUNC*n*_OEN_SEL = 1: the output enable is controlled by the corresponding bit *n* of GPIO_ENABLE_REG:
  - GPIO_ENABLE_REG = 0: output is disabled;
  - GPIO_ENABLE_REG = 1: output is enabled;
- GPIO_FUNC*n*_OEN_SEL = 0: use the output enable signal from peripheral, for example SPIQ_oe in the column "Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0" of Table 5.11-1. Note that the signals such as SPIQ_oe can be 1 (1'd1) or 0 (1'd0), depending on the configuration of corresponding peripherals. If it's 1'd1 in the "Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0", it indicates that once the register GPIO_FUNC*n*_OEN_SEL is cleared, the output signal is always enabled by default.

**Note:**

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

- Only the signals with a name assigned in the column "Input signal" in Table 5.11-1 are valid input signals.
- Only the signals with a name assigned in the column "Output signal" in Table 5.11-1 are valid output signals.

Table 5.11-1. Peripheral Signals via GPIO Matrix

| Signal No. | Input Signal | Default value | Direct Input via IO MUX | Output Signal | Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0 | Direct Output via IO MUX |
|---|---|---|---|---|---|---|
| 0 | SPIQ_in | 0 | yes | SPIQ_out | SPIQ_oe | yes |
| 1 | SPID_in | 0 | yes | SPID_out | SPID_oe | yes |
| 2 | SPIHD_in | 0 | yes | SPIHD_out | SPIHD_oe | yes |
| 3 | SPIWP_in | 0 | yes | SPIWP_out | SPIWP_oe | yes |
| 4 | - | - | - | SPICLK_out_mux | SPICLK_oe | yes |
| 5 | - | - | - | SPICS0_out | SPICS0_oe | yes |
| 6 | U0RXD_in | 0 | yes | U0TXD_out | 1'd1 | yes |
| 7 | U0CTS_in | 0 | no | U0RTS_out | 1'd1 | no |
| 8 | U0DSR_in | 0 | no | U0DTR_out | 1'd1 | no |
| 9 | U1RXD_in | 0 | no | U1TXD_out | 1'd1 | no |
| 10 | U1CTS_in | 0 | no | U1RTS_out | 1'd1 | no |
| 11 | U1DSR_in | 0 | no | U1DTR_out | 1'd1 | no |
| 12 | I2S_MCLK_in | 0 | no | I2S_MCLK_out | 1'd1 | no |
| 13 | I2S0_BCK_in | 0 | no | I2S0_BCK_out | 1'd1 | no |
| 14 | I2S0_WS_in | 0 | no | I2S0_WS_out | 1'd1 | no |
| 15 | I2S1_SD_in | 0 | no | I2S0_SD_out | 1'd1 | no |
| 16 | I2S1_BCK_in | 0 | no | I2S1_BCK_out | 1'd1 | no |
| 17 | I2S1_WS_in | 0 | no | I2S1_WS_out | 1'd1 | no |
| 18 | gpio_bt_priority | 0 | no | gpio_wlan_prio | 1'd1 | no |
| 19 | gpio_bt_active | 0 | no | gpio_wlan_active | 1'd1 | no |
| 20 | - | - | - | - | 1'd1 | no |
| 21 | - | - | - | - | 1'd1 | no |
| 22 | - | - | - | - | 1'd1 | no |
| 23 | - | - | - | - | 1'd1 | no |
| 24 | - | - | - | - | 1'd1 | no |

| Signal No. | Input Signal | Default value | Direct Input via IO MUX | Output Signal | Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0 | Direct Output via IO MUX |
|---|---|---|---|---|---|---|
| 25 | - | - | - | - | 1'd1 | no |
| 26 | - | - | - | - | 1'd1 | no |
| 27 | - | - | - | - | 1'd1 | no |
| 28 | cpu_gpio_in0 | 0 | no | cpu_gpio_out0 | cpu_gpio_out_oen0 | no |
| 29 | cpu_gpio_in1 | 0 | no | cpu_gpio_out1 | cpu_gpio_out_oen1 | no |
| 30 | cpu_gpio_in2 | 0 | no | cpu_gpio_out2 | cpu_gpio_out_oen2 | no |
| 31 | cpu_gpio_in3 | 0 | no | cpu_gpio_out3 | cpu_gpio_out_oen3 | no |
| 32 | cpu_gpio_in4 | 0 | no | cpu_gpio_out4 | cpu_gpio_out_oen4 | no |
| 33 | cpu_gpio_in5 | 0 | no | cpu_gpio_out5 | cpu_gpio_out_oen5 | no |
| 34 | cpu_gpio_in6 | 0 | no | cpu_gpio_out6 | cpu_gpio_out_oen6 | no |
| 35 | cpu_gpio_in7 | 0 | no | cpu_gpio_out7 | cpu_gpio_out_oen7 | no |
| 36 | - | - | - | usb_jtag_tck | 1'd1 | no |
| 37 | - | - | - | usb_jtag_tms | 1'd1 | no |
| 38 | - | - | - | usb_jtag_tdi | 1'd1 | no |
| 39 | - | - | - | usb_jtag_tdo | 1'd1 | no |
| 40 | - | - | - | - | 1'd1 | no |
| 41 | - | - | - | - | 1'd1 | no |
| 42 | - | - | - | - | 1'd1 | no |
| 43 | - | - | - | - | 1'd1 | no |
| 44 | - | - | - | - | 1'd1 | no |
| 45 | ext_adc_start | 0 | no | ledc_ls_sig_out0 | 1'd1 | no |
| 46 | - | - | - | ledc_ls_sig_out1 | 1'd1 | no |
| 47 | - | - | - | ledc_ls_sig_out2 | 1'd1 | no |
| 48 | - | - | - | ledc_ls_sig_out3 | 1'd1 | no |
| 49 | - | - | - | ledc_ls_sig_out4 | 1'd1 | no |
| 50 | - | - | - | ledc_ls_sig_out5 | 1'd1 | no |
| 51 | rmt_sig_in0 | 0 | no | rmt_sig_out0 | 1'd1 | no |

| Signal No. | Input Signal | Default value | Direct Input via IO MUX | Output Signal | Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0 | Direct Output via IO MUX |
|---|---|---|---|---|---|---|
| 52 | rmt_sig_in1 | 0 | no | rmt_sig_out1 | 1'd1 | no |
| 53 | I2CEXT0_SCL_in | 1 | no | I2CEXT0_SCL_out | I2CEXT0_SCL_oe | no |
| 54 | I2CEXT0_SDA_in | 1 | no | I2CEXT0_SDA_out | I2CEXT0_SDA_oe | no |
| 55 | - | - | - | gpio_sd0_out | 1'd1 | no |
| 56 | - | - | - | gpio_sd1_out | 1'd1 | no |
| 57 | - | - | - | gpio_sd2_out | 1'd1 | no |
| 58 | - | - | - | gpio_sd3_out | 1'd1 | no |
| 59 | - | - | - | I2S0_SD1_out | 1'd1 | no |
| 60 | - | - | - | - | 1'd1 | no |
| 61 | - | - | - | - | 1'd1 | no |
| 62 | - | - | - | - | 1'd1 | no |
| 63 | FSPICLK_in | 0 | yes | FSPICLK_out_mux | FSPICLK_oe | yes |
| 64 | FSPIQ_in | 0 | yes | FSPIQ_out | FSPIQ_oe | yes |
| 65 | FSPID_in | 0 | yes | FSPID_out | FSPID_oe | yes |
| 66 | FSPIHD_in | 0 | yes | FSPIHD_out | FSPIHD_oe | yes |
| 67 | FSPIWP_in | 0 | yes | FSPIWP_out | FSPIWP_oe | yes |
| 68 | FSPICS0_in | 0 | yes | FSPICS0_out | FSPICS0_oe | yes |
| 69 | - | - | - | FSPICS1_out | FSPICS1_oe | no |
| 70 | - | - | - | FSPICS2_out | FSPICS2_oe | no |
| 71 | - | - | - | FSPICS3_out | FSPICS3_oe | no |
| 72 | - | - | - | FSPICS4_out | FSPICS4_oe | no |
| 73 | - | - | - | FSPICS5_out | FSPICS5_oe | no |
| 74 | twai_rx | 1 | no | twai_tx | 1'd1 | no |
| 75 | - | - | - | twai_bus_off_on | 1'd1 | no |
| 76 | - | - | - | twai_clkout | 1'd1 | no |
| 77 | - | - | - | - | 1'd1 | no |
| 78 | - | - | - | - | 1'd1 | no |

| Signal No. | Input Signal | Default value | Direct Input via IO MUX | Output Signal | Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0 | Direct Output via IO MUX |
|---|---|---|---|---|---|---|
| 79 | - | - | - | - | 1'd1 | no |
| 80 | - | - | - | - | 1'd1 | no |
| 81 | - | - | - | - | 1'd1 | no |
| 82 | - | - | - | - | 1'd1 | no |
| 83 | - | - | - | - | 1'd1 | no |
| 84 | - | - | - | - | 1'd1 | no |
| 85 | - | - | - | - | 1'd1 | no |
| 86 | - | - | - | - | 1'd1 | no |
| 87 | - | - | - | - | 1'd1 | no |
| 88 | - | - | - | - | 1'd1 | no |
| 89 | - | - | - | ant_sel0 | 1'd1 | no |
| 90 | - | - | - | ant_sel1 | 1'd1 | no |
| 91 | - | - | - | ant_sel2 | 1'd1 | no |
| 92 | - | - | - | ant_sel3 | 1'd1 | no |
| 93 | - | - | - | ant_sel4 | 1'd1 | no |
| 94 | - | - | - | ant_sel5 | 1'd1 | no |
| 95 | - | - | - | ant_sel6 | 1'd1 | no |
| 96 | - | - | - | ant_sel7 | 1'd1 | no |
| 97 | sig_in_func_97 | 0 | no | sig_in_func97 | 1'd1 | no |
| 98 | sig_in_func_98 | 0 | no | sig_in_func98 | 1'd1 | no |
| 99 | sig_in_func_99 | 0 | no | sig_in_func99 | 1'd1 | no |
| 100 | sig_in_func_100 | 0 | no | sig_in_func100 | 1'd1 | no |
| 101 | - | - | - | - | 1'd1 | no |
| 102 | - | - | - | - | 1'd1 | no |
| 103 | - | - | - | - | 1'd1 | no |
| 104 | - | - | - | - | 1'd1 | no |
| 105 | - | - | - | - | 1'd1 | no |

| Signal No. | Input Signal | Default value | Direct Input via IO MUX | Output Signal | Output enable signal when GPIO_FUNC*n*_OEN_SEL = 0 | Direct Output via IO MUX |
|---|---|---|---|---|---|---|
| 106 | - | - | - | - | 1'd1 | no |
| 107 | - | - | - | - | 1'd1 | no |
| 108 | - | - | - | - | 1'd1 | no |
| 109 | - | - | - | - | 1'd1 | no |
| 110 | - | - | - | - | 1'd1 | no |
| 111 | - | - | - | - | 1'd1 | no |
| 112 | - | - | - | - | 1'd1 | no |
| 113 | - | - | - | - | 1'd1 | no |
| 114 | - | - | - | - | 1'd1 | no |
| 115 | - | - | - | - | 1'd1 | no |
| 116 | - | - | - | - | 1'd1 | no |
| 117 | - | - | - | - | 1'd1 | no |
| 118 | - | - | - | - | 1'd1 | no |
| 119 | - | - | - | - | 1'd1 | no |
| 120 | - | - | - | - | 1'd1 | no |
| 121 | - | - | - | - | 1'd1 | no |
| 122 | - | - | - | - | 1'd1 | no |
| 123 | - | - | - | CLK_OUT_out1 | 1'd1 | no |
| 124 | - | - | - | CLK_OUT_out2 | 1'd1 | no |
| 125 | - | - | - | CLK_OUT_out3 | 1'd1 | no |
| 126 | - | - | - | SPICS1_out | 1'd1 | no |
| 127 | - | - | - | usb_jtag_trst | 1'd1 | no |

## 5.12 IO MUX Functions List

Table 5.12-1 shows the IO MUX functions of each pin.

Table 5.12-1. IO MUX Pin Functions

| Pin No. | Pin Name | Function 0 | Function 1 | Function 2 | Function 3 | DRV | Reset | Notes |
|---|---|---|---|---|---|---|---|---|
| 4 | XTAL_32K_P | GPIO0 | GPIO0 | - | - | 2 | 0 | R |
| 5 | XTAL_32K_N | GPIO1 | GPIO1 | - | - | 2 | 0 | R |
| 6 | GPIO2 | GPIO2 | GPIO2 | FSPIQ | - | 2 | 1 | R |
| 8 | GPIO3 | GPIO3 | GPIO3 | - | - | 2 | 1 | R |
| 9 | MTMS | MTMS | GPIO4 | FSPIHD | - | 2 | 1 | R |
| 10 | MTDI | MTDI | GPIO5 | FSPIWP | - | 2 | 1 | R |
| 12 | MTCK | MTCK | GPIO6 | FSPICLK | - | 2 | 1* | G |
| 13 | MTDO | MTDO | GPIO7 | FSPID | - | 2 | 1 | G |
| 14 | GPIO8 | GPIO8 | GPIO8 | - | - | 2 | 1 | - |
| 15 | GPIO9 | GPIO9 | GPIO9 | - | - | 2 | 3 | - |
| 16 | GPIO10 | GPIO10 | GPIO10 | FSPICS0 | - | 2 | 1 | G |
| 18 | VDD_SPI | GPIO11 | GPIO11 | - | - | 2 | 0 | - |
| 19 | SPIHD | SPIHD | GPIO12 | - | - | 2 | 3 | - |
| 20 | SPIWP | SPIWP | GPIO13 | - | - | 2 | 3 | - |
| 21 | SPICS0 | SPICS0 | GPIO14 | - | - | 2 | 3 | - |
| 22 | SPICLK | SPICLK | GPIO15 | - | - | 2 | 3 | - |
| 23 | SPID | SPID | GPIO16 | - | - | 2 | 3 | - |
| 24 | SPIQ | SPIQ | GPIO17 | - | - | 2 | 3 | - |
| 25 | GPIO18 | GPIO18 | GPIO18 | - | - | 3 | 0 | USB, G |
| 26 | GPIO19 | GPIO19 | GPIO19 | - | - | 3 | 0* | USB |
| 27 | U0RXD | U0RXD | GPIO20 | - | - | 2 | 3 | G |
| 28 | U0TXD | U0TXD | GPIO21 | - | - | 2 | 4 | - |

### Drive Strength

"DRV" column shows the drive strength of each pin after reset:

- GPIO2, GPIO3, GPIO4, GPIO5, GPIO18, GPIO19
  - **0** - Drive current = ~5 mA
  - **1** - Drive current = ~20 mA
  - **2** - Drive current = ~10 mA
  - **3** - Drive current = ~40 mA
- Other GPIOs
  - **0** - Drive current = ~5 mA
  - **1** - Drive current = ~10 mA

- **2** - Drive current = ~20 mA
- **3** - Drive current = ~40 mA

**Reset Configurations**

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

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

**Note:**

- **R** - Pins in VDD3P3_RTC domain, and part of them have analog functions, see Table 5.13-1.
- **USB** - GPIO18 and GPIO19 are USB pins. The pull-up value of the two pins are controlled by the pins' pull-up value together with USB pull-up value. If any one of the pull-up value is 1, the pin's pull-up resistor will be enabled. The pull-up resistors of USB pins are controlled by USB_SERIAL_JTAG_DP_PULLUP.
- **G** - These pins have glitches during power-up. See details in Table 5.12-2.

**Table 5.12-2. Power-Up Glitches on Pins**

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

## 5.13 Analog Functions List

Table 5.13-1 shows the IO MUX pins with analog functions.

**Table 5.13-1. Analog Functions of IO MUX Pins**

| GPIO Num | Pin Name | Analog Function 0 | Analog Function 1 |
|---|---|---|---|
| 0 | XTAL_32K_P | XTAL_32K_P | ADC1_CH0 |
| 1 | XTAL_32K_N | XTAL_32K_N | ADC1_CH1 |
| 2 | GPIO2 | - | ADC1_CH2 |

| GPIO Num | Pin Name | Analog Function 0 | Analog Function 1 |
|---|---|---|---|
| 3 | GPIO3 | - | ADC1_CH3 |
| 4 | MTMS | - | ADC1_CH4 |

## 5.14 Register Summary

### 5.14.1 GPIO Matrix Register Summary

The addresses in this section are relative to the GPIO base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Registers** | | | |
| GPIO_BT_SELECT_REG | GPIO bit select register | 0x0000 | R/W |
| GPIO_OUT_REG | GPIO output register | 0x0004 | R/W/SS |
| GPIO_OUT_W1TS_REG | GPIO output set register | 0x0008 | WT |
| GPIO_OUT_W1TC_REG | GPIO output clear register | 0x000C | WT |
| GPIO_ENABLE_REG | GPIO output enable register | 0x0020 | R/W/SS |
| GPIO_ENABLE_W1TS_REG | GPIO output enable set register | 0x0024 | WT |
| GPIO_ENABLE_W1TC_REG | GPIO output enable clear register | 0x0028 | WT |
| GPIO_STRAP_REG | pin strapping register | 0x0038 | RO |
| GPIO_IN_REG | GPIO input register | 0x003C | RO |
| GPIO_STATUS_REG | GPIO interrupt status register | 0x0044 | R/W/SS |
| GPIO_STATUS_W1TS_REG | GPIO interrupt status set register | 0x0048 | WT |
| GPIO_STATUS_W1TC_REG | GPIO interrupt status clear register | 0x004C | WT |
| GPIO_PCPU_INT_REG | GPIO PRO_CPU interrupt status register | 0x005C | RO |
| GPIO_STATUS_NEXT_REG | GPIO interrupt source register | 0x014C | RO |
| **Pin Configuration Registers** | | | |
| GPIO_PIN0_REG | GPIO pin0 configuration register | 0x0074 | R/W |
| GPIO_PIN1_REG | GPIO pin1 configuration register | 0x0078 | R/W |
| GPIO_PIN2_REG | GPIO pin2 configuration register | 0x007C | R/W |
| GPIO_PIN3_REG | GPIO pin3 configuration register | 0x0080 | R/W |
| GPIO_PIN4_REG | GPIO pin4 configuration register | 0x0084 | R/W |
| GPIO_PIN5_REG | GPIO pin5 configuration register | 0x0088 | R/W |
| GPIO_PIN6_REG | GPIO pin6 configuration register | 0x008C | R/W |
| GPIO_PIN7_REG | GPIO pin7 configuration register | 0x0090 | R/W |
| GPIO_PIN8_REG | GPIO pin8 configuration register | 0x0094 | R/W |
| GPIO_PIN9_REG | GPIO pin9 configuration register | 0x0098 | R/W |
| GPIO_PIN10_REG | GPIO pin10 configuration register | 0x009C | R/W |
| GPIO_PIN11_REG | GPIO pin11 configuration register | 0x00A0 | R/W |
| GPIO_PIN12_REG | GPIO pin12 configuration register | 0x00A4 | R/W |
| GPIO_PIN13_REG | GPIO pin13 configuration register | 0x00A8 | R/W |
| GPIO_PIN14_REG | GPIO pin14 configuration register | 0x00AC | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| GPIO_PIN15_REG | GPIO pin15 configuration register | 0x00B0 | R/W |
| GPIO_PIN16_REG | GPIO pin16 configuration register | 0x00B4 | R/W |
| GPIO_PIN17_REG | GPIO pin17 configuration register | 0x00B8 | R/W |
| GPIO_PIN18_REG | GPIO pin18 configuration register | 0x00BC | R/W |
| GPIO_PIN19_REG | GPIO pin19 configuration register | 0x00C0 | R/W |
| GPIO_PIN20_REG | GPIO pin20 configuration register | 0x00C4 | R/W |
| GPIO_PIN21_REG | GPIO pin21 configuration register | 0x00C8 | R/W |
| **Input Function Configuration Registers** | | | |
| GPIO_FUNC0_IN_SEL_CFG_REG | Configuration register for input signal 0 | 0x0154 | R/W |
| GPIO_FUNC1_IN_SEL_CFG_REG | Configuration register for input signal 1 | 0x0158 | R/W |
| ... | ... | ... | ... |
| GPIO_FUNC126_IN_SEL_CFG_REG | Configuration register for input signal 126 | 0x034C | R/W |
| GPIO_FUNC127_IN_SEL_CFG_REG | Configuration register for input signal 127 | 0x0350 | R/W |
| **Output Function Configuration Registers** | | | |
| GPIO_FUNC0_OUT_SEL_CFG_REG | Configuration register for GPIO0 output | 0x0554 | R/W |
| GPIO_FUNC1_OUT_SEL_CFG_REG | Configuration register for GPIO1 output | 0x0558 | R/W |
| GPIO_FUNC2_OUT_SEL_CFG_REG | Configuration register for GPIO2 output | 0x055C | R/W |
| GPIO_FUNC3_OUT_SEL_CFG_REG | Configuration register for GPIO3 output | 0x0560 | R/W |
| GPIO_FUNC4_OUT_SEL_CFG_REG | Configuration register for GPIO4 output | 0x0564 | R/W |
| GPIO_FUNC5_OUT_SEL_CFG_REG | Configuration register for GPIO5 output | 0x0568 | R/W |
| GPIO_FUNC6_OUT_SEL_CFG_REG | Configuration register for GPIO6 output | 0x056C | R/W |
| GPIO_FUNC7_OUT_SEL_CFG_REG | Configuration register for GPIO7 output | 0x0570 | R/W |
| GPIO_FUNC8_OUT_SEL_CFG_REG | Configuration register for GPIO8 output | 0x0574 | R/W |
| GPIO_FUNC9_OUT_SEL_CFG_REG | Configuration register for GPIO9 output | 0x0578 | R/W |
| GPIO_FUNC10_OUT_SEL_CFG_REG | Configuration register for GPIO10 output | 0x057C | R/W |
| GPIO_FUNC11_OUT_SEL_CFG_REG | Configuration register for GPIO11 output | 0x0580 | R/W |
| GPIO_FUNC12_OUT_SEL_CFG_REG | Configuration register for GPIO12 output | 0x0584 | R/W |
| GPIO_FUNC13_OUT_SEL_CFG_REG | Configuration register for GPIO13 output | 0x0588 | R/W |
| GPIO_FUNC14_OUT_SEL_CFG_REG | Configuration register for GPIO14 output | 0x058C | R/W |
| GPIO_FUNC15_OUT_SEL_CFG_REG | Configuration register for GPIO15 output | 0x0590 | R/W |
| GPIO_FUNC16_OUT_SEL_CFG_REG | Configuration register for GPIO16 output | 0x0594 | R/W |
| GPIO_FUNC17_OUT_SEL_CFG_REG | Configuration register for GPIO17 output | 0x0598 | R/W |
| GPIO_FUNC18_OUT_SEL_CFG_REG | Configuration register for GPIO18 output | 0x059C | R/W |
| GPIO_FUNC19_OUT_SEL_CFG_REG | Configuration register for GPIO19 output | 0x05A0 | R/W |
| GPIO_FUNC20_OUT_SEL_CFG_REG | Configuration register for GPIO20 output | 0x05A4 | R/W |
| GPIO_FUNC21_OUT_SEL_CFG_REG | Configuration register for GPIO21 output | 0x05A8 | R/W |
| **Version Register** | | | |
| GPIO_DATE_REG | GPIO version register | 0x06FC | R/W |
| **Clock Gate Register** | | | |
| GPIO_CLOCK_GATE_REG | GPIO clock gate register | 0x062C | R/W |

### 5.14.2 IO MUX Register Summary

The addresses in this section are relative to the IO MUX base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Registers** | | | |
| IO_MUX_PIN_CTRL_REG | Clock output configuration Register | 0x0000 | R/W |
| IO_MUX_GPIO0_REG | IO MUX configuration register for pin XTAL_32K_P | 0x0004 | R/W |
| IO_MUX_GPIO1_REG | IO MUX configuration register for pin XTAL_32K_N | 0x0008 | R/W |
| IO_MUX_GPIO2_REG | IO MUX configuration register for pin GPIO2 | 0x000C | R/W |
| IO_MUX_GPIO3_REG | IO MUX configuration register for pin GPIO3 | 0x0010 | R/W |
| IO_MUX_GPIO4_REG | IO MUX configuration register for pin MTMS | 0x0014 | R/W |
| IO_MUX_GPIO5_REG | IO MUX configuration register for pin MTDI | 0x0018 | R/W |
| IO_MUX_GPIO6_REG | IO MUX configuration register for pin MTCK | 0x001C | R/W |
| IO_MUX_GPIO7_REG | IO MUX configuration register for pin MTDO | 0x0020 | R/W |
| IO_MUX_GPIO8_REG | IO MUX configuration register for pin GPIO8 | 0x0024 | R/W |
| IO_MUX_GPIO9_REG | IO MUX configuration register for pin GPIO9 | 0x0028 | R/W |
| IO_MUX_GPIO10_REG | IO MUX configuration register for pin GPIO10 | 0x002C | R/W |
| IO_MUX_GPIO11_REG | IO MUX configuration register for pin VDD_SPI | 0x0030 | R/W |
| IO_MUX_GPIO12_REG | IO MUX configuration register for pin SPIHD | 0x0034 | R/W |
| IO_MUX_GPIO13_REG | IO MUX configuration register for pin SPIWP | 0x0038 | R/W |
| IO_MUX_GPIO14_REG | IO MUX configuration register for pin SPICS0 | 0x003C | R/W |
| IO_MUX_GPIO15_REG | IO MUX configuration register for pin SPICLK | 0x0040 | R/W |
| IO_MUX_GPIO16_REG | IO MUX configuration register for pin SPID | 0x0044 | R/W |
| IO_MUX_GPIO17_REG | IO MUX configuration register for pin SPIQ | 0x0048 | R/W |
| IO_MUX_GPIO18_REG | IO MUX configuration register for pin GPIO18 | 0x004C | R/W |
| IO_MUX_GPIO19_REG | IO MUX configuration register for pin GPIO19 | 0x0050 | R/W |
| IO_MUX_GPIO20_REG | IO MUX configuration register for pin U0RXD | 0x0054 | R/W |
| IO_MUX_GPIO21_REG | IO MUX configuration register for pin U0TXD | 0x0058 | R/W |
| **Version Register** | | | |
| IO_MUX_DATE_REG | IO MUX Version Control Register | 0x00FC | R/W |

### 5.14.3 SDM Register Summary

The addresses in this section are relative to (GPIO base address provided in Table 3.3-3 in Chapter 3 *System and Memory* + 0x0F00).

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration registers** | | | |

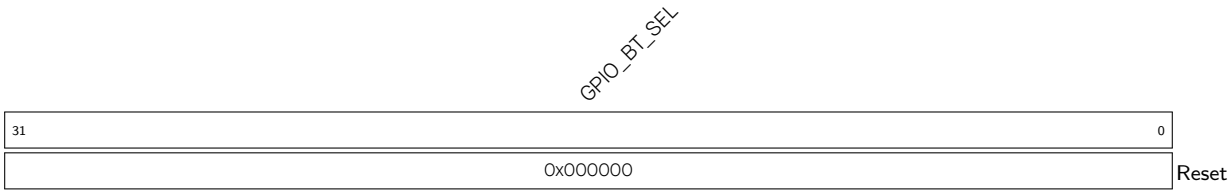
| Name | Description | Address | Access |
|---|---|---|---|
| GPIOSD_SIGMADELTA0_REG | Duty Cycle Configuration Register of SDM0 | 0x0000 | R/W |
| GPIOSD_SIGMADELTA1_REG | Duty Cycle Configuration Register of SDM1 | 0x0004 | R/W |
| GPIOSD_SIGMADELTA2_REG | Duty Cycle Configuration Register of SDM2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

## 5.15 Registers

### 5.15.1 GPIO Matrix Registers

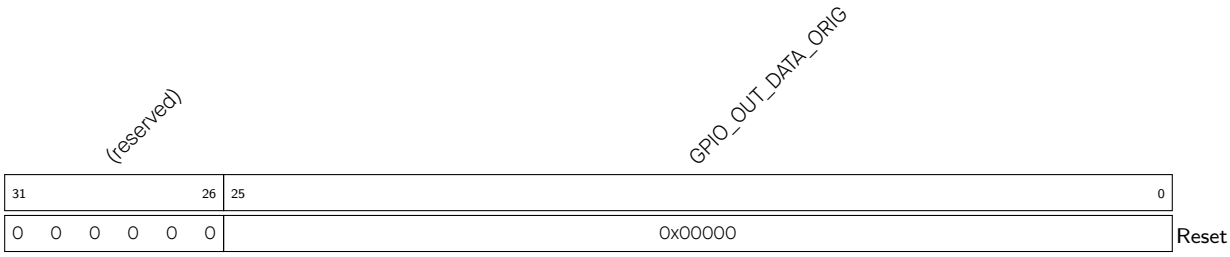
The addresses in this section are relative to the GPIO base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

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

**Register 5.3. GPIO_OUT_W1TS_REG (0x0008)**

| (reserved) | GPIO_OUT_W1TS | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.4. GPIO_OUT_W1TC_REG (0x000C)**

| (reserved) | GPIO_OUT_W1TC | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.5. GPIO_ENABLE_REG (0x0020)**

| (reserved) | GPIO_ENABLE_DATA | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.6. GPIO_ENABLE_W1TS_REG (0x0024)**

| (reserved) | GPIO_ENABLE_W1TS | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.7. GPIO_ENABLE_W1TC_REG (0x0028)**

| (reserved) | GPIO_ENABLE_W1TC | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.8. GPIO_STRAP_REG (0x0038)**

| (reserved) | GPIO_STRAPPING | |
|---|---|---|
| 31 16 | 15 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x00 | Reset |

**GPIO_STRAPPING** GPIO strapping values. (RO)

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

Register 5.9. GPIO_IN_REG (0x003C)

| (reserved) | GPIO_IN_DATA_NEXT | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

Register 5.10. GPIO_STATUS_REG (0x0044)

| (reserved) | GPIO_STATUS_INTERRUPT | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

Register 5.11. GPIO_STATUS_W1TS_REG (0x0048)

| (reserved) | GPIO_STATUS_W1TS | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.12. GPIO_STATUS_W1TC_REG (0x004C)**

| (reserved) | GPIO_STATUS_W1TC | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.13. GPIO_PCPU_INT_REG (0x005C)**

| (reserved) | GPIO_PROCPU_INT | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.14. GPIO_PIN*n*_REG (*n*: 0-21) (0x0074+4**n*)**

| (reserved) | GPIO_PIN*n*_INT_ENA | GPIO_PIN*n*_CONFIG | GPIO_PIN*n*_WAKEUP_ENABLE | GPIO_PIN*n*_INT_TYPE | (reserved) | GPIO_PIN*n*_SYNC1_BYPASS | GPIO_PIN*n*_PAD_DRIVER | GPIO_PIN*n*_SYNC2_BYPASS | |
|---|---|---|---|---|---|---|---|---|---|
| 31 18 | 17 13 | 12 11 | 10 | 9 7 | 6 5 | 4 3 | 2 | 1 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | 0x0 | 0 | 0x0 | 0 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** pin drive selection. 0: normal output; 1: open drain output. (R/W)

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

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

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

**GPIO_PIN*n*_CONFIG** reserved (R/W)

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

**Register 5.15. GPIO_STATUS_NEXT_REG (0x014C)**

| (reserved) | GPIO_STATUS_INTERRUPT_NEXT | |
|---|---|---|
| 31 26 | 25 0 | |
| 0 0 0 0 0 0 | 0x00000 | Reset |

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

**Register 5.16. GPIO_FUNC*n*_IN_SEL_CFG_REG (*n*: 0-127) (0x0154+4**n*)**

| (reserved) | GPIO_SIG*n*_IN_SEL | GPIO_FUNC*n*_IN_INV_SEL | GPIO_FUNC*n*_IN_SEL | |
|---|---|---|---|---|
| 31 … 7 | 6 | 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 0 0 | 0 | 0 | 0x0 | Reset |

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

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

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

**Register 5.17. GPIO_FUNC*n*_OUT_SEL_CFG_REG (*n*: 0-21) (0x0554+4**n*)**

| (reserved) | GPIO_FUNC*n*_OEN_INV_SEL | GPIO_FUNC*n*_OEN_SEL | GPIO_FUNC*n*_OUT_INV_SEL | GPIO_FUNC*n*_OUT_SEL | |
|---|---|---|---|---|---|
| 31 … 11 | 10 | 9 | 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0x80 | Reset |

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

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

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

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

**Register 5.18. GPIO_CLOCK_GATE_REG (0x062C)**

| (reserved) | GPIO_CLK_EN | |
|---|---|---|
| 31 … 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 | 1 | Reset |

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

**Register 5.19. GPIO_DATE_REG (0x06FC)**

| (reserved) | GPIO_DATE_REG | |
|---|---|---|
| 31 … 28 | 27 … 0 | |
| 0 0 0 0 | 0x2006130 | Reset |

**GPIO_DATE_REG** Version control register (R/W)

## 5.15.2 IO MUX Registers

The addresses in this section are relative to the IO MUX base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 5.20. IO_MUX_PIN_CTRL_REG (0x0000)**

| (reserved) | IO_MUX_CLK_OUT3 | IO_MUX_CLK_OUT2 | IO_MUX_CLK_OUT1 | |
|---|---|---|---|---|
| 31 … 12 | 11 … 8 | 7 … 4 | 3 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x7 | 0xf | 0xf | Reset |

**IO_MUX_CLK_OUT*x*** If you want to output clock for I2S to CLK_OUT_out*x*, set IO_MUX_CLK_OUT*x* to 0x0. CLK_OUT_out*x* can be found in Table 5.11-1. (R/W)

**Register 5.21. IO_MUX_GPIO*n*_REG (*n*: 0-21) (0x0004+4**n*)**

| 31–16 | 15 | 14–12 | 11–10 | 9 | 8 | 7 | 6–5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | IO_MUX_GPIO*n*_FILTER_EN | IO_MUX_GPIO*n*_MCU_SEL | IO_MUX_GPIO*n*_FUN_DRV | IO_MUX_GPIO*n*_FUN_IE | IO_MUX_GPIO*n*_FUN_WPU | IO_MUX_GPIO*n*_FUN_WPD | IO_MUX_GPIO*n*_MCU_DRV | IO_MUX_GPIO*n*_MCU_IE | IO_MUX_GPIO*n*_MCU_WPU | IO_MUX_GPIO*n*_MCU_WPD | IO_MUX_GPIO*n*_SLP_SEL | IO_MUX_GPIO*n*_MCU_OE | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0x0 | 0x2 | 1 | 1 | 0 | 0 0 | 0 | 0 | 0 | 0 | 0 | Reset |

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

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

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

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

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

**IO_MUX_GPIO*n*_MCU_DRV** Configures the drive strength of GPIO*n* during sleep mode.

- GPIO2, GPIO3, GPIO5, GPIO18, GPIO18, GPIO19
  0: ~5 mA
  1: **~20 mA**
  2: **~10 mA**
  3: ~40 mA
- Other GPIOs
  0: ~5 mA
  1: **~10 mA**
  2: **~20 mA**
  3: ~40 mA

(R/W)

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

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

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

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

**Register 5.21. IO_MUX_GPIO*n*_REG (*n*: 0-21) (0x0004+4**n*)**

**Continued from the previous page...**

**IO_MUX_GPIO*n*_FUN_DRV** Select the drive strength of the pin.

- GPIO2, GPIO3, GPIO5, GPIO18, GPIO18, GPIO19
  0: ~5 mA
  1: **~20 mA**
  2: **~10 mA**
  3: ~40 mA
- Other GPIOs
  0: ~5 mA
  1: **~10 mA**
  2: **~20 mA**
  3: ~40 mA

(R/W)

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

**IO_MUX_GPIO*n*_FILTER_EN** Enable filter for pin input signals. 1: Filter enabled; 0: Filter disabled. (R/W)

**Register 5.22. IO_MUX_DATE_REG (0x00FC)**

| (reserved) | IO_MUX_DATE_REG | |
|---|---|---|
| 31 28 | 27 0 | |
| 0 0 0 0 | 0x2006050 | Reset |

**IO_MUX_DATE_REG** Version control register (R/W)

## 5.15.3 SDM Output Registers

The addresses in this section are relative to (GPIO base address provided in Table 3.3-3 in Chapter 3 *System and Memory* + 0x0F00).

**Register 5.23. GPIOSD_SIGMADELTA*n*_REG (*n*: 0-3) (0x0000+4**n*)**

| (reserved) | GPIOSD_SD*n*_PRESCALE | GPIOSD_SD*n*_IN | |
|---|---|---|---|
| 31 … 16 | 15 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 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 5.24. GPIOSD_SIGMADELTA_CG_REG (0x0020)**

| GPIOSD_CLK_EN | (reserved) | |
|---|---|---|
| 31 | 30 … 0 | |
| 0 | 0 0 0 0 0 0 0 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 |

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

**Register 5.25. GPIOSD_SIGMADELTA_MISC_REG (0x0024)**

| GPIOSD_SPI_SWAP | GPIOSD_FUNCTION_CLK_EN | (reserved) | |
|---|---|---|---|
| 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 | Reset |

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

**GPIOSD_SPI_SWAP** Reserved. (R/W)

**Register 5.26. GPIOSD_SIGMADELTA_VERSION_REG (0x0028)**

| (reserved) | GPIOSD_DATE | |
|---|---|---|
| 31 28 | 27 0 | |
| 0 0 0 0 | 0x2006230 | Reset |

**GPIOSD_DATE** Version Control Register. (R/W)

# Chapter 6

# Reset and Clock

## 6.1 Reset

### 6.1.1 Overview

ESP32-C3 provides four types of reset that occur at different levels, namely CPU Reset, Core Reset, System Reset, and Chip Reset. All reset types mentioned above (except Chip Reset) maintain the data stored in internal memory. Figure 6.1-1 shows the scope of affected subsystems by each type of reset.

### 6.1.2 Architectural Overview

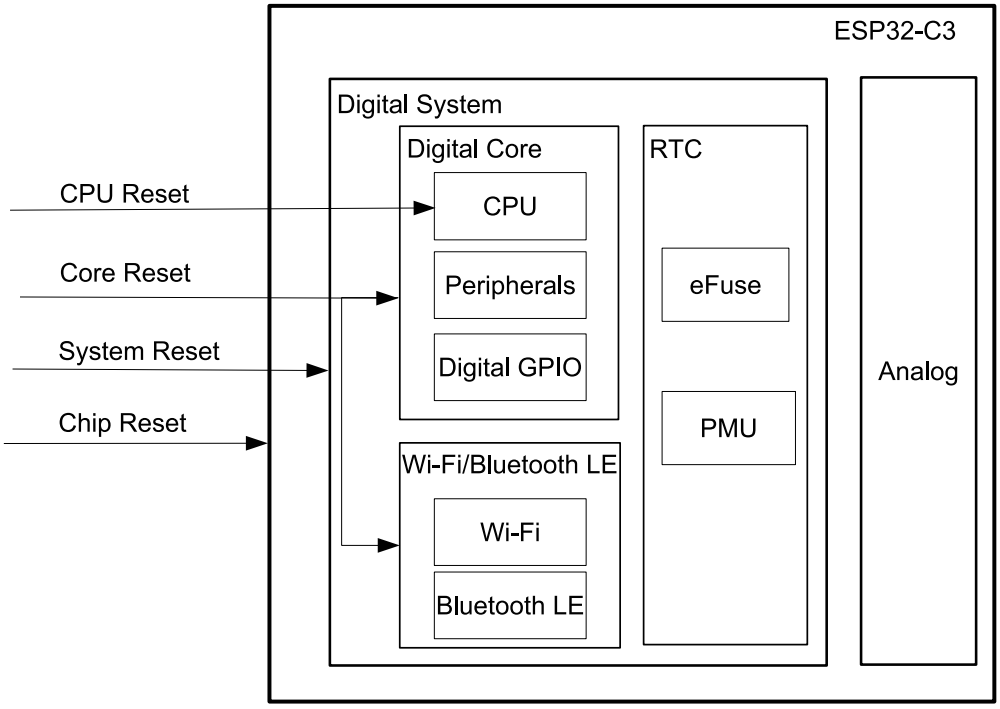



Figure 6.1-1. Reset Types

### 6.1.3 Features

- Support four reset levels:
  - CPU Reset: Only resets CPU core. Once such reset is released, the instructions from the CPU reset vector will be executed.

  - Core Reset: Resets the whole digital system except RTC, including CPU, peripherals, Wi-Fi, Bluetooth® LE, and digital GPIOs.
  - System Reset: Resets the whole digital system, including RTC.
  - Chip Reset: Resets the whole chip.
- Support software reset and hardware reset:
  - Software Reset: the CPU can trigger a software reset by configuring the corresponding registers, see Chapter 9 *Low-power Management*.
  - Hardware Reset: Hardware reset is directly triggered by the circuit.

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

### 6.1.4 Functional Description

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

Table 6.1-1 lists possible reset sources and the types of reset they trigger.

Table 6.1-1. Reset Sources

| Code | Source | Reset Type | Comments |
|---|---|---|---|
| 0x01 | Chip reset[1] | Chip Reset | - |
| 0x0F | Brown-out system reset | Chip Reset or System Reset | Triggered by brown-out detector[2] |
| 0x10 | RWDT system reset | System Reset | See Chapter 12 *Watchdog Timers (WDT)* |
| 0x12 | Super Watchdog reset | System Reset | See Chapter 12 *Watchdog Timers (WDT)* |
| 0x13 | CLK GLITCH reset | System Reset | See Chapter 25 *Clock Glitch Detection* |
| 0x03 | Software system reset | Core Reset | Triggered by configuring RTC_CNTL_SW_SYS_RST |
| 0x05 | Deep-sleep reset | Core Reset | See Chapter 9 *Low-power Management* |
| 0x07 | MWDT0 core reset | Core Reset | See Chapter 12 *Watchdog Timers (WDT)* |
| 0x08 | MWDT1 core reset | Core Reset | See Chapter 12 *Watchdog Timers (WDT)* |
| 0x09 | RWDT core reset | Core Reset | See Chapter 12 *Watchdog Timers (WDT)* |
| 0x14 | eFuse reset | Core Reset | Triggered by eFuse CRC error |
| 0x15 | USB (UART) reset | Core Reset | Triggered when external USB host sends a specific command to the Serial interface of USB-Serial-JTAG. See 30 *USB Serial/JTAG Controller (USB_SERIAL_JTAG)* |
| 0x16 | USB (JTAG) reset | Core Reset | Triggered when external USB host sends a specific command to the JTAG interface of USB-Serial-JTAG. See 30 *USB Serial/JTAG Controller (USB_SERIAL_JTAG)* |
| 0x17 | Power glitch reset | Core Reset | Triggered by power glitch |
| 0x0B | MWDT0 CPU reset | CPU Reset | See Chapter 12 *Watchdog Timers (WDT)* |
| 0x0C | Software CPU reset | CPU Reset | Triggered by configuring RTC_CNTL_SW_PROCPU_RST |
| 0x0D | RWDT CPU reset | CPU Reset | See Chapter 12 *Watchdog Timers (WDT)* |
| 0x11 | MWDT1 CPU reset | CPU Reset | See Chapter 12 *Watchdog Timers (WDT)* |

[1] Chip Reset can be triggered by the following two sources:

- Triggered by chip power-on.
- Triggered by brown-out detector.

[2] Once brown-out status is detected, the detector will trigger System Reset or Chip Reset, depending on register configuration. See Chapter 9 *Low-power Management*.

## 6.2 Clock

### 6.2.1 Overview

ESP32-C3 clocks are mainly sourced from oscillator (OSC), RC, and PLL circuit, and then processed by the dividers or selectors, which allows most functional modules to select their working clock according to their power consumption and performance requirements. Figure 6.2-1 shows the system clock structure.

## 6.2.2 Architectural Overview

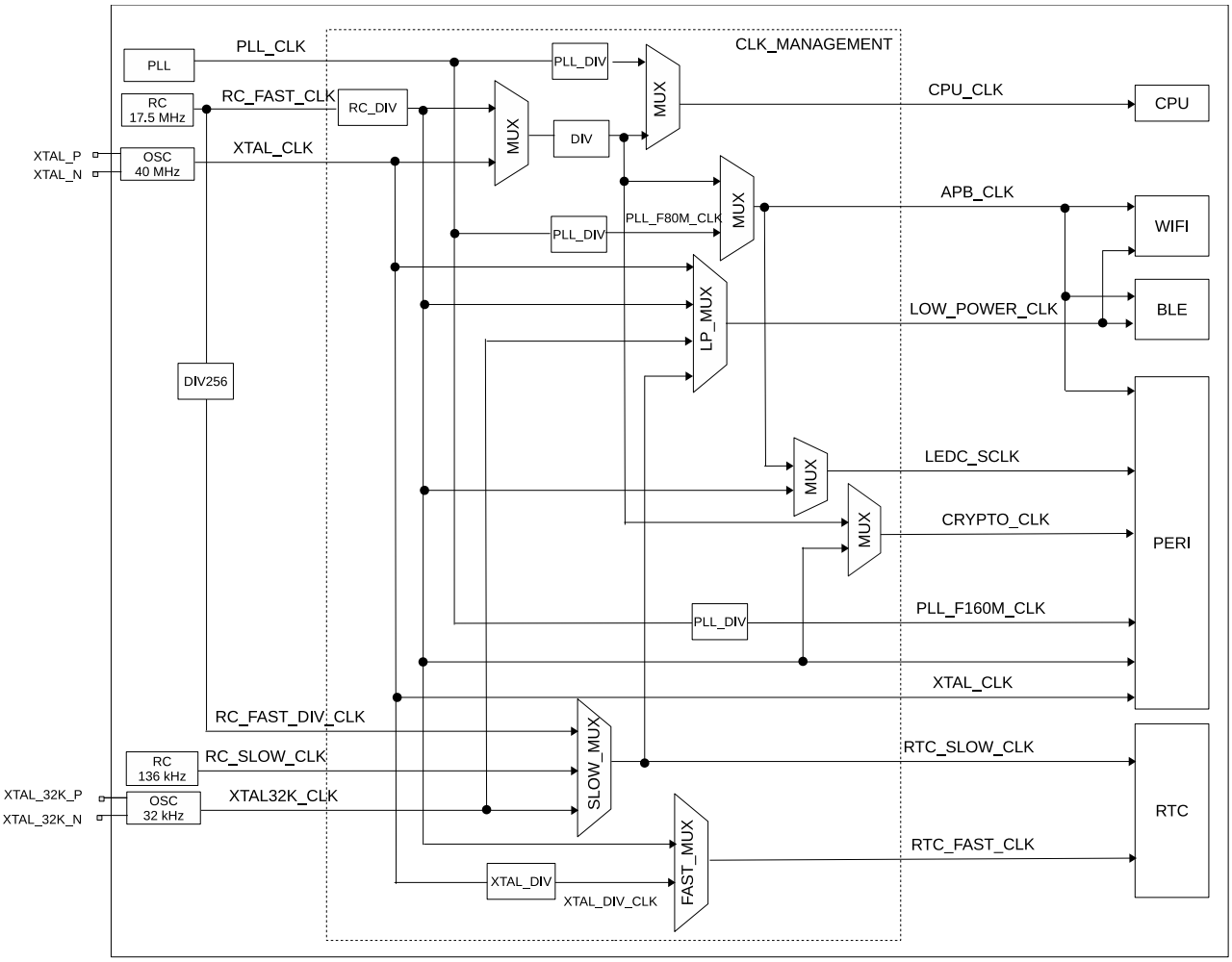



Figure 6.2-1. System Clock

## 6.2.3 Features

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

- High speed clocks for devices working at a higher frequency, such as CPU and digital peripherals
  - PLL_CLK (320 MHz or 480 MHz): internal PLL clock
  - XTAL_CLK (40 MHz): external crystal clock
- Slow speed clocks for low-power devices, such as RTC module and low-power peripherals
  - XTAL32K_CLK (32 kHz): external crystal clock
  - RC_FAST_CLK (17.5 MHz by default): internal fast RC oscillator with adjustable frequency
  - RC_FAST_DIV_CLK: internal fast RC oscillator derived from RC_FAST_CLK divided by 256
  - RC_SLOW_CLK (136 kHz by default): internal low RC oscillator with adjustable frequency

## 6.2.4 Functional Description

#### 6.2.4.1 CPU Clock

As Figure 6.2-1 shows, CPU_CLK is the master clock for CPU and it can be as high as 160 MHz when CPU works in high performance mode. Alternatively, CPU can run at lower frequencies, such as at 2 MHz, to lower power consumption. Users can set PLL_CLK, RC_FAST_CLK or XTAL_CLK as CPU_CLK clock source by configuring register SYSTEM_SOC_CLK_SEL, see Table 6.2-1 and Table 6.2-2. By default, the CPU clock is sourced from XTAL_CLK with a divider of 2, i.e. the CPU clock is 20 MHz.

Table 6.2-1. CPU Clock Source

| SYSTEM_SOC_CLK_SEL Value | CPU Clock Source |
|---|---|
| 0 | XTAL_CLK |
| 1 | PLL_CLK |
| 2 | RC_FAST_CLK |

Table 6.2-2. CPU Clock Frequency

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

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

#### 6.2.4.2 Peripheral Clock

Peripheral clocks include APB_CLK, CRYPTO_CLK, PLL_F160M_CLK, LEDC_SCLK, XTAL_CLK, and RC_FAST_CLK. Table 6.2-3 shows which clock can be used by each peripheral.

Table 6.2-3. Peripheral Clocks

| Peripheral | XTAL_CLK | APB_CLK | PLL_F160M_CLK | RTC_FAST_CLK | RC_FAST_CLK | CRYPTO_CLK | LEDC_CLK | PLL_D2_CLK |
|---|---|---|---|---|---|---|---|---|
| TIMG | Y | Y | | | | | | |
| I2S | Y | | Y | | | | | Y |
| UHCI | | Y | | | | | | |
| UART | Y | Y | | | Y | | | |
| RMT | Y | Y | | | Y | | | |
| I2C | Y | | | | Y | | | |
| SPI | Y | Y | | | | | | |
| eFuse Controller | Y | | | Y | | | | |
| SARADC | | Y | | | | | | |
| Temperature Sensor | Y | | | | Y | | | |
| USB | | Y | | | | | | |
| CRYPTO | | | | | | Y | | |
| TWAI Controller | | Y | | | | | | |
| LEDC | Y | Y | Y | | Y | | Y | |
| SYS_TIMER | Y | Y | | | | | | |

#### APB_CLK

The frequency of APB_CLK is determined by the clock source of CPU_CLK as shown in Table 6.2-4.

Table 6.2-4. APB_CLK Clock Frequency

| CPU_CLK Source | APB_CLK Frequency |
|---|---|
| PLL_CLK | 80 MHz |
| XTAL_CLK | CPU_CLK |
| RC_FAST_CLK | CPU_CLK |

#### CRYPTO_CLK

The frequency of CRYPTO_CLK is determined by the CPU_CLK source, as shown in Table 6.2-5.

Table 6.2-5. CRYPTO_CLK Frequency

| CPU_CLK Source | CRYPTO_CLK Frequency |
|---|---|
| PLL_CLK | 160 MHz |
| XTAL_CLK | CPU_CLK |
| RC_FAST_CLK | CPU_CLK |

#### PLL_F160M_CLK

PLL_F160M_CLK is divided from PLL_CLK according to current PLL frequency.

#### LEDC_SCLK

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

### 6.2.4.3 Wi-Fi and Bluetooth LE Clock

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

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

### 6.2.4.4 RTC Clock

The clock sources for RTC_SLOW_CLK and RTC_FAST_CLK are low-frequency clocks. RTC module can operate when most other clocks are stopped. RTC_SLOW_CLK derived from RC_SLOW_CLK, XTAL32K_CLK or RC_FAST_DIV_CLK is used to clock Power Management module. RTC_FAST_CLK is used to clock On-chip Sensor module. It can be sourced from a divided XTAL_CLK or from a divided RC_FAST_CLK.

# Chapter 7

# Chip Boot Control

## 7.1 Overview

ESP32-C3 has three strapping pins:

- GPIO2
- GPIO8
- GPIO9

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

- control chip boot mode
- ROM code printing

During power-on reset, RTC watchdog reset, brownout reset, analog super watchdog reset, and crystal clock glitch detection reset (see Chapter 6 *Reset and Clock*), hardware captures samples and stores the voltage level of strapping pins as strapping bit of "0" or "1" in latches, and holds these bits until the chip is powered down or shut down. Software can read the latch status (strapping value) from GPIO_STRAPPING.

By default, GPIO9 is connected to the chip's internal pull-up resistor. If GPIO9 is not connected or connected to an external high-impedance circuit, the internal weak pull-up determines the default input level of this strapping pin (see Table 7.1-1).

**Table 7.1-1. Default Configuration of Strapping Pins**

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

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

**Note:**
The following section provides description of the chip functions and the pattern of the strapping pins values to invoke each function. Only documented patterns should be used. If some pattern is not documented, it may trigger unexpected behavior.

## 7.2 Boot Mode Control

The values of GPIO2, GPIO3, GPIO8, and GPIO9 at reset determine the boot mode after the reset is released. Table 7.2-1 shows the strapping pin values of GPIO9, GPIO8, GPIO3, and GPIO2, and the associated boot modes.

**Table 7.2-1. Boot Mode Control**

| Boot Mode | GPIO9 | GPIO8 | GPIO3 | GPIO2 |
|---|---|---|---|---|
| SPI Boot mode | 1 | x[1] | x | x |
| Joint Download Boot mode[2] | 0 | 1 | x | x |
| SPI Download Boot mode[3] | 0 | 0 | 0 | 1 |
| Invalid Combination[4] | 0 | 0 | x | 0 |

[1] x: values that have no effect on the result and can therefore be ignored.

[2] Joint Download Boot mode: Joint Download Boot mode supports the following download methods:

- USB-Serial-JTAG Download Boot
- UART Download Boot

[3] SPI Download Boot mode: GPIO3 and GPIO2 need to be reserved only when using SPI Download Boot mode. GPIO3 and GPIO2 are floating by default and are in a high-impedance state at reset.

[4] Invalid Combination: This combination can trigger unexpected behavior and should be avoided.

In SPI Boot mode, the ROM bootloader loads and executes the program from SPI flash to boot the system. SPI Boot mode can be further classified as follows:

- Normal Flash Boot: supports Security Boot. The ROM bootloader loads the program from flash into SRAM and executes it. In most practical scenarios, this program is the 2nd stage bootloader, which later boots the target application.
- Direct Boot: does not support Security Boot and programs run directly from flash. To enable this mode, make sure that the first two words of the bin file downloaded to flash (address: 0x42000000) are 0xaedb041d.

In Joint Download Boot mode, users can download binary files into flash using UART0 or USB interface. It is also possible to download binary files into SRAM and execute it in this mode.

In SPI Download Boot mode, users can download binary files into flash using SPI interface. It is also possible to download binary files into SRAM and execute it from SRAM.

The following eFuses control boot mode behaviors:

- EFUSE_DIS_FORCE_DOWNLOAD
  - If this eFuse is 0 (default), software can force switch the chip from SPI Boot mode to Joint Download Boot mode by setting RTC_CNTL_FORCE_DOWNLOAD_BOOT and triggering a CPU reset. In this case, hardware overwrites GPIO_STRAPPING[3:2] from "1x" to "01".

  - If this eFuse is 1, RTC_CNTL_FORCE_DOWNLOAD_BOOT is disabled. GPIO_STRAPPING can not be overwritten.

- EFUSE_DIS_DOWNLOAD_MODE

  If this eFuse is 1, Joint Download Boot mode is disabled. GPIO_STRAPPING will not be overwritten by RTC_CNTL_FORCE_DOWNLOAD_BOOT.

- EFUSE_ENABLE_SECURITY_DOWNLOAD

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

- EFUSE_DIS_DIRECT_BOOT

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

USB Serial/JTAG Controller can also force the chip into Joint Download Boot mode from SPI Boot mode, as well as force the chip into SPI Boot mode from Joint Download Boot mode. For detailed information, please refer to Chapter *30 USB Serial/JTAG Controller (USB_SERIAL_JTAG)*.

## 7.3 ROM Messages Printing Control

During early SPI Boot process, the messages by the ROM code can be printed to:

- **(Default) UART0 and USB Serial/JTAG controller**
- UART0
- USB Serial/JTAG controller

EFUSE_UART_PRINT_CONTROL and GPIO8 control ROM messages printing to UART0 as shown in Table 7.3-1.

**Table 7.3-1. ROM Message Printing Control**

| eFuse[1] | GPIO8 | ROM Code Printing |
|---|---|---|
| 0 | x | ROM code is always printed to UART0 during boot. The value of GPIO8 is ignored. |
| 1 | 0 | Print is enabled during boot. |
| | 1 | Print is disabled during boot. |
| 2 | 0 | Print is disabled during boot. |
| | 1 | Print is enabled during boot. |
| 3 | x | Print is always disabled during boot. The value of GPIO8 is ignored. |

[1] eFuse: EFUSE_UART_PRINT_CONTROL

EFUSE_USB_PRINT_CHANNEL controls the printing to USB Serial/JTAG controller. When this bit is 1, printing to USB Serial/JTAG controller is disabled. When this bit is 0 and the USB Serial/JTAG controller is enabled via EFUSE_DIS_USB_SERIAL_JTAG, ROM messages can be printed to USB Serial/JTAG controller.

Note that if EFUSE_USB_PRINT_CHANNEL is set to 0 to print ROM messages to USB, but USB Serial/JTAG controller has been disabled, then ROM messages will not be printed to USB Serial/JTAG controller.

Please note that ROM message printing to UART0 and to the USB Serial/JTAG Controller is controlled independently.

# Chapter 8

# Interrupt Matrix (INTERRUPT)

## 8.1 Overview

The interrupt matrix embedded in ESP32-C3 independently routes peripheral interrupt sources to the ESP-RISC-V CPU's peripheral interrupts, to timely inform CPU to process the coming interrupts.

The ESP32-C3 has 62 peripheral interrupt sources. To map them to 31 CPU interrupts, this interrupt matrix is needed.

> **Note:**
> This chapter focuses on how to map peripheral interrupt sources to CPU interrupts. For more details about interrupt configuration, vector, and ISA suggested operations, please refer to Chapter 1 *ESP-RISC-V CPU*.

## 8.2 Features

- Accept 62 peripheral interrupt sources as input
- Generate 31 CPU peripheral interrupts to CPU as output
- Query current interrupt status of peripheral interrupt sources
- Configure priority, type, threshold, and enable signal of CPU interrupts

Figure 8.2-1 shows the structure of the interrupt matrix.

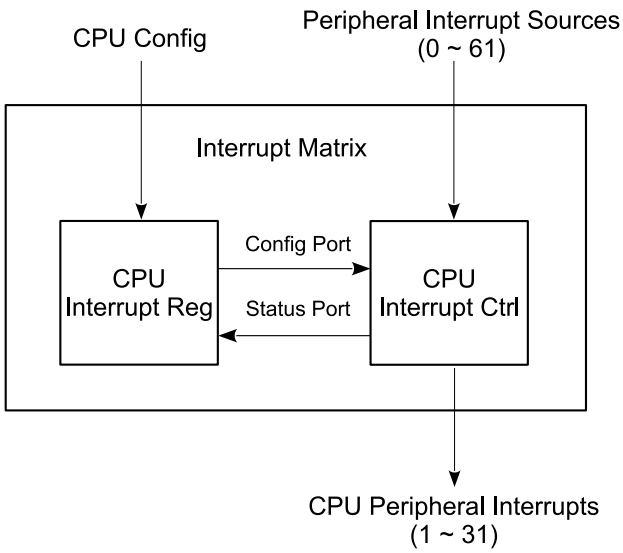



**Figure 8.2-1. Interrupt Matrix Structure**

## 8.3 Functional Description

### 8.3.1 Peripheral Interrupt Sources

The ESP32-C3 has 62 peripheral interrupt sources in total. Table 8.3-1 lists all these sources and their configuration/status registers.

- Column "No.": Peripheral interrupt source number, can be 0 ~ 61.
- Column "Chapter": in which chapter the interrupt source is described in detailed.
- Column "Source": Name of the peripheral interrupt source.
- Column "Configuration Register": Registers used for routing the peripheral interrupt sources to CPU peripheral interrupts
- Column "Status Register": Registers used for indicating the interrupt status of peripheral interrupt sources.
  - Column "Status Register - Bit": Bit position in status register, indicating the interrupt status.
  - Column "Status Register - Name": Name of status registers.

Table 8.3-1. CPU Peripheral Interrupt Configuration/Status Registers and Peripheral Interrupt Sources

| No. | Chapter | Source | Configuration Register | Status Register | |
|---|---|---|---|---|---|
| | | | | Bit | Name |
| 0 | reserved | reserved | reserved | 0 | INTERRUPT_COREO_INTR_STATUS_0_REG |
| 1 | reserved | reserved | reserved | 1 | |
| 2 | reserved | reserved | reserved | 2 | |
| 3 | reserved | reserved | reserved | 3 | |
| 4 | reserved | reserved | reserved | 4 | |
| 5 | reserved | reserved | reserved | 5 | |
| 6 | reserved | reserved | reserved | 6 | |
| 7 | reserved | reserved | reserved | 7 | |
| 8 | reserved | reserved | reserved | 8 | |
| 9 | reserved | reserved | reserved | 9 | |
| 10 | reserved | reserved | reserved | 10 | |
| 11 | reserved | reserved | reserved | 11 | |
| 12 | reserved | reserved | reserved | 12 | |
| 13 | reserved | reserved | reserved | 13 | |
| 14 | reserved | reserved | reserved | 14 | |
| 15 | *UART Controller (UART)* | UHCIO_INTR | INTERRUPT_COREO_UHCIO_INTR_MAP_REG | 15 | |
| 16 | *IO MUX and GPIO Matrix (GPIO, IO MUX)* | GPIO_PROCPU_INTR | INTERRUPT_COREO_GPIO_INTERRUPT_PRO_MAP_REG | 16 | |
| 17 | reserved | reserved | reserved | 17 | |
| 18 | reserved | reserved | reserved | 18 | |
| 19 | *SPI Controller (SPI)* | GPSPI2_INTR_2 | INTERRUPT_COREO_SPI_INTR_2_MAP_REG | 19 | |
| 20 | *I2S Controller (I2S)* | I2S_INTR | INTERRUPT_COREO_I2S_INT_MAP_REG | 20 | |
| 21 | *UART Controller (UART)* | UART_INTR | INTERRUPT_COREO_UART_INTR_MAP_REG | 21 | |
| 22 | *UART Controller (UART)* | UART1_INTR | INTERRUPT_COREO_UART1_INTR_MAP_REG | 22 | |
| 23 | *LED PWM Controller (LEDC)* | LEDC_INTR | INTERRUPT_COREO_LEDC_INT_MAP_REG | 23 | |
| 24 | *eFuse Controller (EFUSE)* | EFUSE_INTR | INTERRUPT_COREO_EFUSE_INT_MAP_REG | 24 | |
| 25 | *Two-wire Automotive Interface (TWAI)* | TWAI_INTR | INTERRUPT_COREO_TWAI_INT_MAP_REG | 25 | |
| 26 | *USB Serial/JTAG Controller (USB_SERIAL_JTAG)* | USB_SERIAL_JTAG_INTR | INTERRUPT_COREO_USB_INTR_MAP_REG | 26 | |
| 27 | *Low-power Management* | RTC_CNTL_INTR | INTERRUPT_COREO_RTC_CORE_INTR_MAP_REG | 27 | |
| 28 | *Remote Control Peripheral (RMT)* | RMT_INTR | INTERRUPT_COREO_RMT_INTR_MAP_REG | 28 | |
| 29 | *I2C Controller (I2C)* | I2C_EXTO_INTR | INTERRUPT_COREO_I2C_EXTO_INTR_MAP_REG | 29 | |
| 30 | reserved | reserved | reserved | 30 | |

| No. | Chapter | Source | Configuration Register | Status Register | |
|---|---|---|---|---|---|
| | | | | Bit | Name |
| 31 | reserved | reserved | reserved | 31 | |
| 32 | *Timer Group (TIMG)* | TG_T0_INTR | INTERRUPT_CORE0_TG_T0_INT_MAP_REG | 0 | INTERRUPT_CORE0_INTR_STATUS_1_REG |
| 33 | *Timer Group (TIMG)* | TG_WDT_INTR | INTERRUPT_CORE0_TG_WDT_INT_MAP_REG | 1 | |
| 34 | *Timer Group (TIMG)* | TG1_T0_INTR | INTERRUPT_CORE0_TG1_T0_INT_MAP_REG | 2 | |
| 35 | *Timer Group (TIMG)* | TG1_WDT_INTR | INTERRUPT_CORE0_TG1_WDT_INT_MAP_REG | 3 | |
| 36 | reserved | reserved | reserved | 36 | |
| 37 | *System Timer (SYSTIMER)* | SYSTIMER_TARGET0_INTR | INTERRUPT_CORE0_SYSTIMER_TARGET0_INT_MAP_REG | 5 | |
| 38 | *System Timer (SYSTIMER)* | SYSTIMER_TARGET1_INTR | INTERRUPT_CORE0_SYSTIMER_TARGET1_INT_MAP_REG | 6 | |
| 39 | *System Timer (SYSTIMER)* | SYSTIMER_TARGET2_INTR | INTERRUPT_CORE0_SYSTIMER_TARGET2_INT_MAP_REG | 7 | |
| 40 | reserved | reserved | reserved | 8 | |
| 41 | reserved | reserved | reserved | 9 | |
| 42 | reserved | reserved | reserved | 10 | |
| 43 | *On-Chip Sensor and Analog Signal Processing* | vDIGTAL_ADC_INTR | INTERRUPT_CORE0_APB_ADC_INT_MAP_REG | 11 | |
| 44 | *GDMA Controller (GDMA)* | GDMA_CH0_INTR | INTERRUPT_CORE0_DMA_CH0_INT_MAP_REG | 12 | |
| 45 | *GDMA Controller (GDMA)* | GDMA_CH1_INTR | INTERRUPT_CORE0_DMA_CH1_INT_MAP_REG | 13 | |
| 46 | *GDMA Controller (GDMA)* | GDMA_CH2_INTR | INTERRUPT_CORE0_DMA_CH2_INT_MAP_REG | 14 | |
| 47 | *RSA Accelerator (RSA)* | RSA_INTR | INTERRUPT_CORE0_RSA_INTR_MAP_REG | 15 | |
| 48 | *AES Accelerator (AES)* | AES_INTR | INTERRUPT_CORE0_AES_INTR_MAP_REG | 16 | |
| 49 | *SHA Accelerator (SHA)* | SHA_INTR | INTERRUPT_CORE0_SHA_INTR_MAP_REG | 17 | |
| 50 | *System Registers (SYSREG)* | SW_INTR_0 | INTERRUPT_CORE0_CPU_INTR_FROM_CPU_0_MAP_REG | 18 | |
| 51 | *System Registers (SYSREG)* | SW_INTR_1 | INTERRUPT_CORE0_CPU_INTR_FROM_CPU_1_MAP_REG | 19 | |
| 52 | *System Registers (SYSREG)* | SW_INTR_2 | INTERRUPT_CORE0_CPU_INTR_FROM_CPU_2_MAP_REG | 20 | |
| 53 | *System Registers (SYSREG)* | SW_INTR_3 | INTERRUPT_CORE0_CPU_INTR_FROM_CPU_3_MAP_REG | 21 | |
| 54 | *Debug Assistant (ASSIST_DEBUG)* | ASSIST_DEBUG_INTR | INTERRUPT_CORE0_ASSIST_DEBUG_INTR_MAP_REG | 22 | |
| 55 | *Permission Control (PMS)* | PMS_DMA_VIO_INTR | INTER-RUPT_CORE0_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR_MAP_REG | 23 | |
| 56 | *Permission Control (PMS)* | PMS_IBUS_VIO_INTR | INTER-RUPT_CORE0_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG | 24 | |
| 57 | *Permission Control (PMS)* | PMS_DBUS_VIO_INTR | INTER-RUPT_CORE0_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG | 25 | |
| 58 | *Permission Control (PMS)* | PMS_PERI_VIO_INTR | INTER-RUPT_CORE0_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG | 26 | |
| 59 | *Permission Control (PMS)* | PMS_PERI_VIO_SIZE_INTR | INTER-RUPT_CORE0_CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG | 27 | |
| 60 | reserved | reserved | reserved | 28 | |

| No. | Chapter | Source | Configuration Register | Status Register | |
|---|---|---|---|---|---|
| | | | | Bit | Name |
| 61 | reserved | reserved | reserved | 29 | |

### 8.3.2 CPU Interrupts

The ESP32-C3 implements its interrupt mechanism using an interrupt controller instead of RISC-V Privileged ISA specification. The ESP-RISC-V CPU has 31 interrupts, numbered from 1 ~ 31. Each CPU interrupt has the following properties.

- Priority levels from 1 (lowest) to 15 (highest).
- Configurable as level-triggered or edge-triggered.
- Lower-priority interrupts mask-able by setting interrupt threshold.

**Note:**
For detailed information about how to configure CPU interrupts, see Chapter 1 *ESP-RISC-V CPU*.

### 8.3.3 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 peripheral interrupt source, wherein *X* means the number of this interrupt source in Table 8.3-1.
- INTERRUPT_COREO_SOURCE_*X*_MAP_REG: stands for a configuration register for the peripheral interrupt source (Source_*X*).
- Num_P: the index of CPU interrupts, can be 1 ~ 31.
- Interrupt_P: stands for the CPU interrupt numbered as Num_P.

#### 8.3.3.1 Allocate one peripheral interrupt source (Source_*X*) to CPU

Setting the corresponding configuration register INTERRUPT_COREO_SOURCE_*X*_MAP_REG of Source_*X* to Num_P allocates this interrupt source to Interrupt_P.

#### 8.3.3.2 Allocate multiple peripheral interrupt sources (Source_X*n*) to CPU

Setting the corresponding configuration register INTERRUPT_COREO_SOURCE_X*n*_MAP_REG of each interrupt source to the same Num_P allocates multiple sources to the same Interrupt_P. Any of these sources can trigger CPU Interrupt_P. When an interrupt signal is generated, CPU should check the interrupt status registers to figure out which peripheral generated the interrupt. For more information, see Chapter 1 *ESP-RISC-V CPU*.

#### 8.3.3.3 Disable CPU peripheral interrupt source (Source_*X*)

Clearing the configuration register INTERRUPT_COREO_SOURCE_*X*_MAP_REG disables the corresponding interrupt source.

### 8.3.4 Query Current Interrupt Status of Peripheral Interrupt Source

Users can query current interrupt status of a peripheral interrupt source by reading the bit value in INTERRUPT_COREO

_INTR_STATUS_*n*_REG (read only). For the mapping between INTERRUPT_COREO_INTR_STATUS_*n*_REG and peripheral interrupt sources, please refer to Table 8.3-1.

## 8.4 Register Summary

The addresses in this section are relative to the interrupt matrix base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Interrupt Source Mapping Registers** | | | |
| INTERRUPT_CORE0_PWR_INTR_MAP_REG | PWR_INTR mapping register | 0x0008 | R/W |
| INTERRUPT_CORE0_I2C_MST_INT_MAP_REG | I2C_MST_INT mapping register | 0x002C | R/W |
| INTERRUPT_CORE0_SLC0_INTR_MAP_REG | SLC0_INTR mapping register | 0x0030 | R/W |
| INTERRUPT_CORE0_SLC1_INTR_MAP_REG | SLC1_INTR mapping register | 0x0034 | R/W |
| INTERRUPT_CORE0_APB_CTRL_INTR_MAP_REG | APB_CTRL_INTR mapping register | 0x0038 | R/W |
| INTERRUPT_CORE0_UHCI0_INTR_MAP_REG | UHCI0_INTR mapping register | 0x003C | R/W |
| INTERRUPT_CORE0_GPIO_INTERRUPT_PRO_MAP_REG | GPIO_INTERRUPT_PRO mapping register | 0x0040 | R/W |
| INTERRUPT_CORE0_SPI_INTR_1_MAP_REG | SPI_INTR_1 mapping register | 0x0048 | R/W |
| INTERRUPT_CORE0_SPI_INTR_2_MAP_REG | SPI_INTR_2 mapping register | 0x004C | R/W |
| INTERRUPT_CORE0_I2S_INT_MAP_REG | I2S_INT mapping register | 0x0050 | R/W |
| INTERRUPT_CORE0_UART_INTR_MAP_REG | UART_INTR mapping register | 0x0054 | R/W |
| INTERRUPT_CORE0_UART1_INTR_MAP_REG | UART1_INTR mapping register | 0x0058 | R/W |
| INTERRUPT_CORE0_LEDC_INT_MAP_REG | LEDC_INT mapping register | 0x005C | R/W |
| INTERRUPT_CORE0_EFUSE_INT_MAP_REG | EFUSE_INT mapping register | 0x0060 | R/W |
| INTERRUPT_CORE0_TWAI_INT_MAP_REG | TWAI_INT mapping register | 0x0064 | R/W |
| INTERRUPT_CORE0_USB_INTR_MAP_REG | USB_INTR mapping register | 0x0068 | R/W |
| INTERRUPT_CORE0_RTC_CORE_INTR_MAP_REG | RTC_CORE_INTR mapping register | 0x006C | R/W |
| INTERRUPT_CORE0_RMT_INTR_MAP_REG | RMT_INTR mapping register | 0x0070 | R/W |
| INTERRUPT_CORE0_I2C_EXT0_INTR_MAP_REG | I2C_EXT0 intr mapping register | 0x0074 | R/W |
| INTERRUPT_CORE0_TIMER_INT1_MAP_REG | TIMER_INT1 mapping register | 0x0078 | R/W |
| INTERRUPT_CORE0_TIMER_INT2_MAP_REG | TIMER_INT2 mapping register | 0x007C | R/W |
| INTERRUPT_CORE0_TG_T0_INT_MAP_REG | TG_T0_INT mapping register | 0x0080 | R/W |
| INTERRUPT_CORE0_TG_WDT_INT_MAP_REG | TG_WDT_INT mapping register | 0x0084 | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| INTERRUPT_CORE0_TG1_T0_INT_MAP_REG | TG1_T0_INT mapping register | 0x0088 | R/W |
| INTERRUPT_CORE0_TG1_WDT_INT_MAP_REG | TG1_WDT_INT mapping register | 0x008C | R/W |
| INTERRUPT_CORE0_CACHE_IA_INT_MAP_REG | CACHE_IA_INT mapping register | 0x0090 | R/W |
| INTERRUPT_CORE0_SYSTIMER_TARGET0_INT_MAP_REG | SYSTIMER_TARGET0_INT mapping register | 0x0094 | R/W |
| INTERRUPT_CORE0_SYSTIMER_TARGET1_INT_MAP_REG | SYSTIMER_TARGET1_INT mapping register | 0x0098 | R/W |
| INTERRUPT_CORE0_SYSTIMER_TARGET2_INT_MAP_REG | SYSTIMER_TARGET2_INT mapping register | 0x009C | R/W |
| INTERRUPT_CORE0_SPI_MEM_REJECT_INTR_MAP_REG | SPI_MEM_REJECT_INTR mapping register | 0x00A0 | R/W |
| INTERRUPT_CORE0_ICACHE_PRELOAD_INT_MAP_REG | ICACHE_PRELOAD_INT mapping register | 0x00A4 | R/W |
| INTERRUPT_CORE0_ICACHE_SYNC_INT_MAP_REG | ICACHE_SYNC_INT mapping register | 0x00A8 | R/W |
| INTERRUPT_CORE0_APB_ADC_INT_MAP_REG | APB_ADC_INT mapping register | 0x00AC | R/W |
| INTERRUPT_CORE0_DMA_CH0_INT_MAP_REG | DMA_CH0_INT mapping register | 0x00B0 | R/W |
| INTERRUPT_CORE0_DMA_CH1_INT_MAP_REG | DMA_CH1_INT mapping register | 0x00B4 | R/W |
| INTERRUPT_CORE0_DMA_CH2_INT_MAP_REG | DMA_CH2_INT mapping register | 0x00B8 | R/W |
| INTERRUPT_CORE0_RSA_INT_MAP_REG | RSA_INT mapping register | 0x00BC | R/W |
| INTERRUPT_CORE0_AES_INT_MAP_REG | AES_INT mapping register | 0x00C0 | R/W |
| INTERRUPT_CORE0_SHA_INT_MAP_REG | SHA_INT mapping register | 0x00C4 | R/W |
| INTERRUPT_CORE0_CPU_INTR_FROM_CPU_0_MAP_REG | CPU_INTR_FROM_CPU_0 mapping register | 0x00C8 | R/W |
| INTERRUPT_CORE0_CPU_INTR_FROM_CPU_1_MAP_REG | CPU_INTR_FROM_CPU_1 mapping register | 0x00CC | R/W |
| INTERRUPT_CORE0_CPU_INTR_FROM_CPU_2_MAP_REG | CPU_INTR_FROM_CPU_2 mapping register | 0x00D0 | R/W |
| INTERRUPT_CORE0_CPU_INTR_FROM_CPU_3_MAP_REG | CPU_INTR_FROM_CPU_3 intr mapping register | 0x00D4 | R/W |
| INTERRUPT_CORE0_ASSIST_DEBUG_INTR_MAP_REG | ASSIST_DEBUG_INTR mapping register | 0x00D8 | R/W |
| INTERRUPT_CORE0_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR_MAP_REG | DMA_APBPERI_PMS_MONITOR_VIOLATE mapping register | 0x00DC | R/W |
| INTERRUPT_CORE0_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG | IRAM0_PMS_MONITOR_VIOLATE mapping register | 0x00E0 | R/W |
| INTERRUPT_CORE0_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP_REG | DRAM0_PMS_MONITOR_VIOLATE mapping register | 0x00E4 | R/W |
| INTERRUPT_CORE0_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP_REG | PIF_PMS_MONITOR_VIOLATE mapping register | 0x00E8 | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| INTERRUPT_COREO_CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP_REG | PIF_PMS_MONITOR_VIOLATE_SIZE mapping register | 0x00EC | R/W |
| INTERRUPT_COREO_BACKUP_PMS_VIOLATE_INTR_MAP_REG | BACKUP_PMS_VIOLATE mapping register | 0x00F0 | R/W |
| INTERRUPT_COREO_CACHE_COREO_ACS_INT_MAP_REG | CACHE_COREO_ACS mapping register | 0x00F4 | R/W |
| **Interrupt Source Status Registers** | | | |
| INTERRUPT_COREO_INTR_STATUS_0_REG | Status register for interrupt sources 0 ~ 31 | 0x00F8 | RO |
| INTERRUPT_COREO_INTR_STATUS_1_REG | Status register for interrupt sources 32 ~ 61 | 0x00FC | RO |
| **Clock Register** | | | |
| INTERRUPT_COREO_CLOCK_GATE_REG | Clock register | 0x0100 | R/W |
| **CPU Interrupt Registers** | | | |
| INTERRUPT_COREO_CPU_INT_ENABLE_REG | Enable register for CPU interrupts | 0x0104 | R/W |
| INTERRUPT_COREO_CPU_INT_TYPE_REG | Type configuration register for CPU interrupts | 0x0108 | R/W |
| INTERRUPT_COREO_CPU_INT_CLEAR_REG | CPU interrupt clear register | 0x010C | R/W |
| INTERRUPT_COREO_CPU_INT_EIP_STATUS_REG | Pending status register for CPU interrupts | 0x0110 | RO |
| INTERRUPT_COREO_CPU_INT_PRI_1_REG | Priority configuration register for CPU interrupt 1 | 0x0118 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_2_REG | Priority configuration register for CPU interrupt 2 | 0x011C | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_3_REG | Priority configuration register for CPU interrupt 3 | 0x0120 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_4_REG | Priority configuration register for CPU interrupt 4 | 0x0124 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_5_REG | Priority configuration register for CPU interrupt 5 | 0x0128 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_6_REG | Priority configuration register for CPU interrupt 6 | 0x012C | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_7_REG | Priority configuration register for CPU interrupt 7 | 0x0130 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_8_REG | Priority configuration register for CPU interrupt 8 | 0x0134 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_9_REG | Priority configuration register for CPU interrupt 9 | 0x0138 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_10_REG | Priority configuration register for CPU interrupt 10 | 0x013C | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_11_REG | Priority configuration register for CPU interrupt 11 | 0x0140 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_12_REG | Priority configuration register for CPU interrupt 12 | 0x0144 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_13_REG | Priority configuration register for CPU interrupt 13 | 0x0148 | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_14_REG | Priority configuration register for CPU interrupt 14 | 0x014C | R/W |
| INTERRUPT_COREO_CPU_INT_PRI_15_REG | Priority configuration register for CPU interrupt 15 | 0x0150 | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| INTERRUPT_CORE0_CPU_INT_PRI_16_REG | Priority configuration register for CPU interrupt 16 | 0x0154 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_17_REG | Priority configuration register for CPU interrupt 17 | 0x0158 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_18_REG | Priority configuration register for CPU interrupt 18 | 0x015C | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_19_REG | Priority configuration register for CPU interrupt 19 | 0x0160 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_20_REG | Priority configuration register for CPU interrupt 20 | 0x0164 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_21_REG | Priority configuration register for CPU interrupt 21 | 0x0168 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_22_REG | Priority configuration register for CPU interrupt 22 | 0x016C | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_23_REG | Priority configuration register for CPU interrupt 23 | 0x0170 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_24_REG | Priority configuration register for CPU interrupt 24 | 0x0174 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_25_REG | Priority configuration register for CPU interrupt 25 | 0x0178 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_26_REG | Priority configuration register for CPU interrupt 26 | 0x017C | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_27_REG | Priority configuration register for CPU interrupt 27 | 0x0180 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_28_REG | Priority configuration register for CPU interrupt 28 | 0x0184 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_29_REG | Priority configuration register for CPU interrupt 29 | 0x0188 | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_30_REG | Priority configuration register for CPU interrupt 30 | 0x018C | R/W |
| INTERRUPT_CORE0_CPU_INT_PRI_31_REG | Priority configuration register for CPU interrupt 31 | 0x0190 | R/W |
| INTERRUPT_CORE0_CPU_INT_THRESH_REG | Threshold configuration register for CPU interrupts | 0x0194 | R/W |
| **Version Register** | | | |
| INTERRUPT_CORE0_INTERRUPT_DATE_REG | Version control register | 0x07FC | R/W |

## 8.5 Registers

The addresses in this section are relative to the interrupt matrix base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

Register 8.1. INTERRUPT_CORE0_*PWR_INTR_MAP*_REG (0x0008)

Register 8.2. INTERRUPT_CORE0_*I2C_MST_INT_MAP*_REG (0x002C)

Register 8.3. INTERRUPT_CORE0_*SLC0_INTR_MAP*_REG (0x0030)

Register 8.4. INTERRUPT_CORE0_*SLC1_INTR_MAP*_REG (0x0034)

Register 8.5. INTERRUPT_CORE0_*SYSCON_INTR_MAP*_REG (0x0038)

Register 8.6. INTERRUPT_CORE0_*UHCI0_INTR_MAP*_REG (0x003C)

Register 8.7. INTERRUPT_CORE0_*GPIO_INTERRUPT_PRO_MAP*_REG (0x0040)

Register 8.8. INTERRUPT_CORE0_*SPI_INTR_1_MAP*_REG (0x0048)

Register 8.9. INTERRUPT_CORE0_*SPI_INTR_2_MAP*_REG (0x004C)

Register 8.10. INTERRUPT_CORE0_*I2S_INT_MAP*_REG (0x0050)

Register 8.11. INTERRUPT_CORE0_*UART_INTR_MAP*_REG (0x0054)

Register 8.12. INTERRUPT_CORE0_*UART1_INTR_MAP*_REG (0x0058)

Register 8.13. INTERRUPT_CORE0_*LEDC_INT_MAP*_REG (0x005C)

Register 8.14. INTERRUPT_CORE0_*EFUSE_INT_MAP*_REG (0x0060)

Register 8.15. INTERRUPT_CORE0_*TWAI_INT_MAP*_REG (0x0064)

Register 8.16. INTERRUPT_CORE0_*USB_INTR_MAP*_REG (0x0068)

Register 8.17. INTERRUPT_CORE0_*RTC_CORE_INTR_MAP*_REG (0x006C)

Register 8.18. INTERRUPT_CORE0_*RMT_INTR_MAP*_REG (0x0070)

Register 8.19. INTERRUPT_CORE0_*I2C_EXT0_INTR_MAP*_REG (0x0074)

Register 8.20. INTERRUPT_CORE0_*TIMER_INT1_MAP*_REG (0x0078)

Register 8.21. INTERRUPT_CORE0_*TIMER_INT2_MAP*_REG (0x007C)

Register 8.22. INTERRUPT_CORE0_*TG_T0_INT_MAP*_REG (0x0080)

Register 8.23. INTERRUPT_CORE0_*TG_WDT_INT_MAP*_REG (0x0084)

Register 8.24. INTERRUPT_CORE0_*TG1_T0_INT_MAP*_REG (0x0088)

Register 8.25. INTERRUPT_CORE0_*TG1_WDT_INT_MAP*_REG (0x008C)

Register 8.26. INTERRUPT_CORE0_*CACHE_IA_INT_MAP*_REG (0x0090)

Register 8.27. INTERRUPT_CORE0_*SYSTIMER_TARGET0_INT_MAP*_REG (0x0094)

Register 8.28. INTERRUPT_CORE0_*SYSTIMER_TARGET1_INT_MAP*_REG (0x0098)

Register 8.29. INTERRUPT_CORE0_*SYSTIMER_TARGET2_INT_MAP*_REG (0x009C)

Register 8.30. INTERRUPT_CORE0_*SPI_MEM_REJECT_INTR_MAP*_REG (0x00A0)

Register 8.31. INTERRUPT_CORE0_*ICACHE_PRELOAD_INT_MAP*_REG (0x00A4)

Register 8.32. INTERRUPT_CORE0_*ICACHE_SYNC_INT_MAP*_REG (0x00A8)

Register 8.33. INTERRUPT_CORE0_*APB_ADC_INT_MAP*_REG (0x00AC)

Register 8.34. INTERRUPT_CORE0_*DMA_CH0_INT_MAP*_REG (0x00B0)

Register 8.35. INTERRUPT_CORE0_*DMA_CH1_INT_MAP*_REG (0x00B4)

Register 8.36. INTERRUPT_CORE0_*DMA_CH2_INT_MAP*_REG (0x00B8)

Register 8.37. INTERRUPT_CORE0_*RSA_INT_MAP*_REG (0x00BC)

Register 8.38. INTERRUPT_CORE0_*AES_INT_MAP*_REG (0x00C0)

Register 8.39. INTERRUPT_CORE0_*SHA_INT_MAP*_REG (0x00C4)

Register 8.40. INTERRUPT_CORE0_*CPU_INTR_FROM_CPU_0_MAP*_REG (0x00C8)

Register 8.41. INTERRUPT_CORE0_*CPU_INTR_FROM_CPU_1_MAP*_REG (0x00CC)

Register 8.42. INTERRUPT_CORE0_*CPU_INTR_FROM_CPU_2_MAP*_REG (0x00D0)

Register 8.43. INTERRUPT_CORE0_*CPU_INTR_FROM_CPU_3_MAP*_REG (0x00D4)

Register 8.44. INTERRUPT_CORE0_*ASSIST_DEBUG_INTR_MAP*_REG (0x00D8)

Register 8.45. INTERRUPT_CORE0_*DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR_MAP*_REG (0x00DC)

Register 8.46. INTERRUPT_CORE0_*CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR_MAP*_REG (0x00E0)

Register 8.47. INTERRUPT_CORE0_*CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR_MAP*_REG (0x00E4)

Register 8.48. INTERRUPT_CORE0_*CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR_MAP*_REG (0x00E8)

Register 8.49. INTERRUPT_CORE0_*CORE_0_PIF_PMS_MONITOR_VIOLATE_SIZE_INTR_MAP*_REG (0x00EC)

Register 8.50. INTERRUPT_CORE0_*BACKUP_PMS_VIOLATE_INTR_MAP*_REG (0x00F0)

Register 8.51. INTERRUPT_CORE0_*CACHE_CORE0_ACS_INT_MAP*_REG (0x00F4)

| (reserved) | INTERRUPT_CORE0_SOURCE_*X*_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 | Reset |

**INTERRUPT_CORE0_SOURCE_*X*_MAP** Map the interrupt source (SOURCE_*X*) into one CPU interrupt. For the information of SOURCE_*X*, see Table 8.3-1. (R/W)

**Register 8.52. INTERRUPT_CORE0_INTR_STATUS_0_REG (0x00F8)**

| INTERRUPT_CORE0_INTR_STATUS_0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**INTERRUPT_CORE0_INTR_STATUS_0** This register stores the status of the first 32 interrupt sources: 0 ~ 31. If the bit is 1 here, it means the corresponding source triggered an interrupt. (RO)

**Register 8.53. INTERRUPT_CORE0_INTR_STATUS_1_REG (0x00FC)**

| INTERRUPT_CORE0_INTR_STATUS_1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

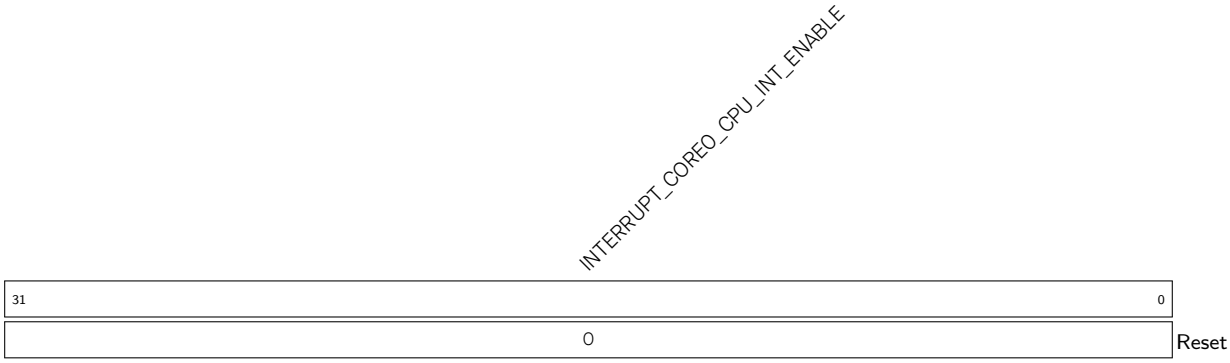
**INTERRUPT_CORE0_INTR_STATUS_1** This register stores the status of the first 32 interrupt sources: 32 ~ 61. If the bit is 1 here, it means the corresponding source triggered an interrupt. (RO)

**Register 8.54. INTERRUPT_CORE0_CLOCK_GATE_REG (0x0100)**

| (reserved) | INTERRUPT_CORE0_CLK_EN | |
|---|---|---|
| 31 … 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 | 1 | Reset |

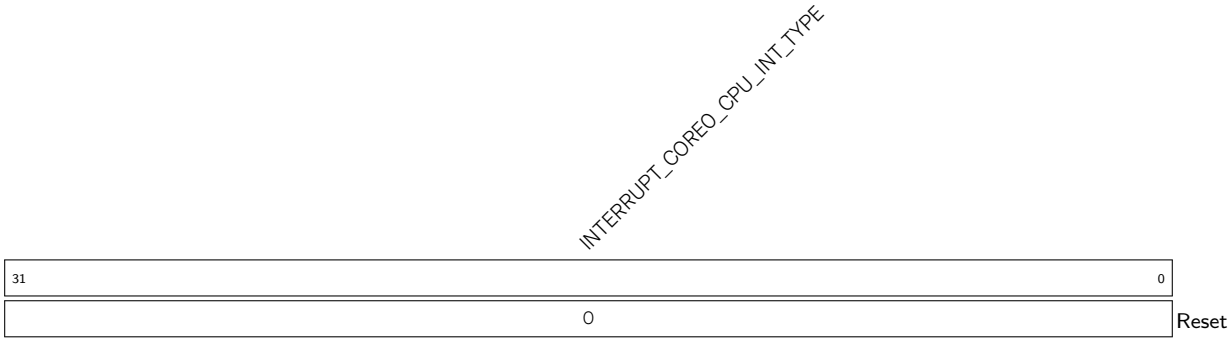
**INTERRUPT_CORE0_CLK_EN** Set 1 to force interrupt register clock-gate on. (R/W)

**Register 8.55. INTERRUPT_COREO_CPU_INT_ENABLE_REG (0x0104)**



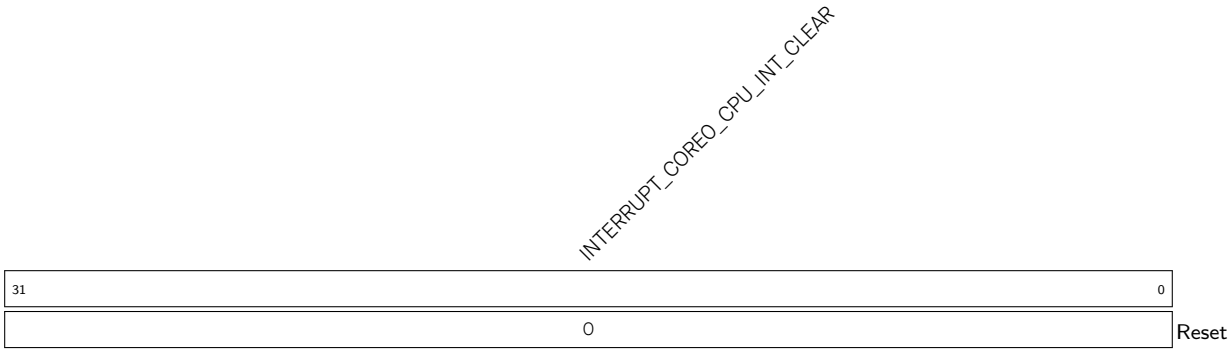

**INTERRUPT_COREO_CPU_INT_ENABLE** Writing 1 to the bit here enables its corresponding CPU interrupt. For more information about how to use this register, see Chapter 1 *ESP-RISC-V CPU*. (R/W)

**Register 8.56. INTERRUPT_COREO_CPU_INT_TYPE_REG (0x0108)**



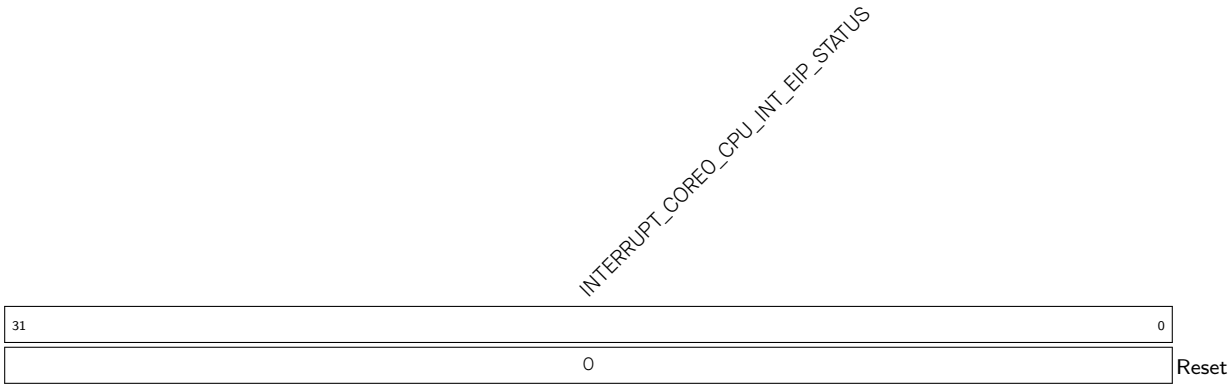

**INTERRUPT_COREO_CPU_INT_TYPE** Configure CPU interrupt type. 0: level-triggered; 1: edge-triggered. For more information about how to use this register, see Chapter 1 *ESP-RISC-V CPU*. (R/W)

**Register 8.57. INTERRUPT_COREO_CPU_INT_CLEAR_REG (0x010C)**



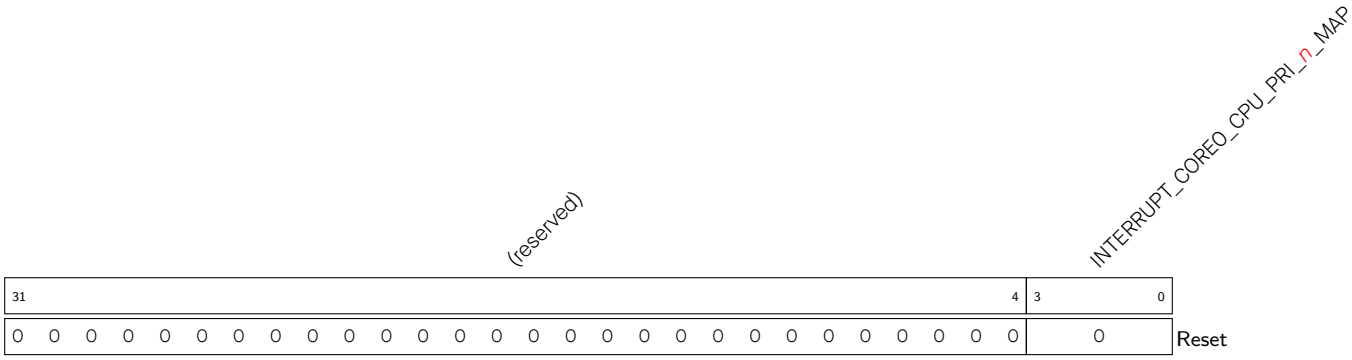

**INTERRUPT_COREO_CPU_INT_CLEAR** Writing 1 to the bit here clears its corresponding CPU interrupt. For more information about how to use this register, see Chapter 1 *ESP-RISC-V CPU*. (R/W)

**Register 8.58. INTERRUPT_COREO_CPU_INT_EIP_STATUS_REG (0x0110)**



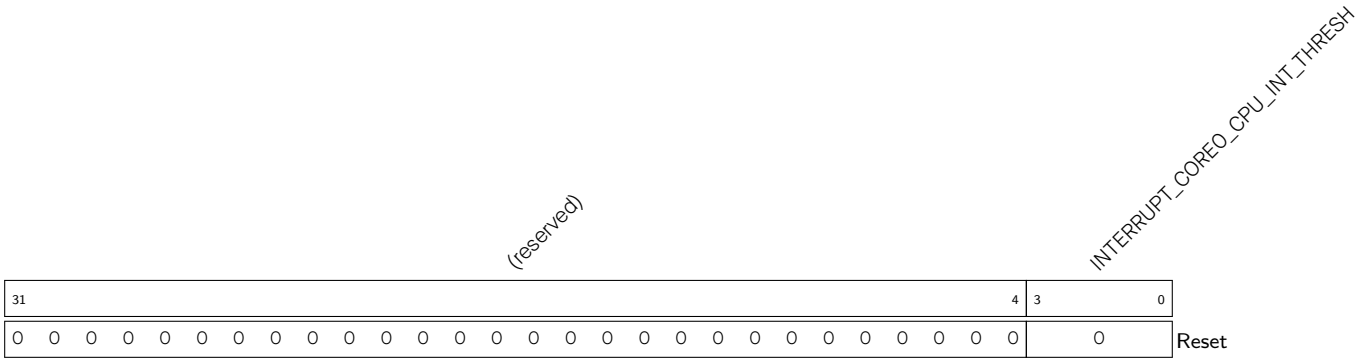

**INTERRUPT_COREO_CPU_INT_EIP_STATUS** Store the pending status of CPU interrupts. For more information about how to use this register, see Chapter 1 *ESP-RISC-V CPU*. (RO)

**Register 8.59. INTERRUPT_COREO_CPU_INT_PRI_*n*_REG (n: 1 - 31)(0x0118 + 0x4**n*)**



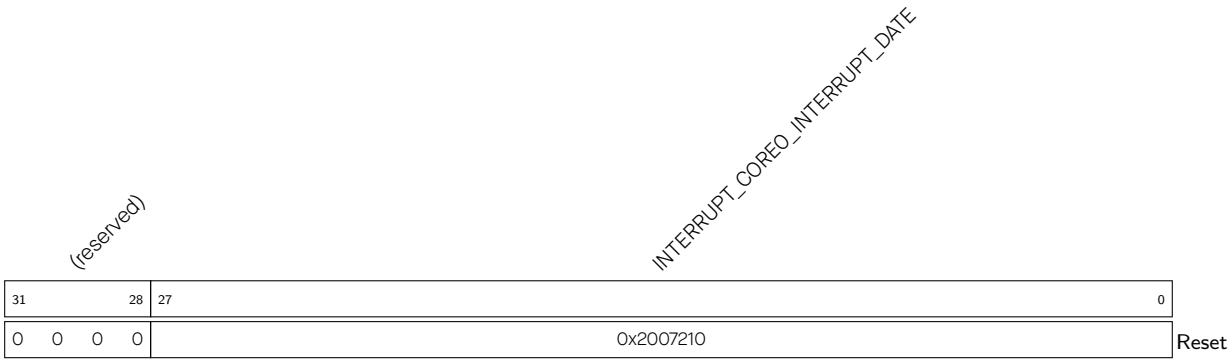

**INTERRUPT_COREO_CPU_PRI_*n*_MAP** Set the priority for CPU interrupt *n*. The priority here can be 1 (lowest) ~ 15 (highest). For more information about how to use this register, see Chapter 1 *ESP-RISC-V CPU*. (R/W)

**Register 8.60. INTERRUPT_COREO_CPU_INT_THRESH_REG (0x0194)**




**INTERRUPT_COREO_CPU_INT_THRESH** Set threshold for interrupt assertion to CPU. Only when the interrupt priority is equal to or higher than this threshold, CPU will respond to this interrupt. For more information about how to use this register, see Chapter 1 *ESP-RISC-V CPU*. (R/W)

**Register 8.61. INTERRUPT_COREO_INTERRUPT_DATE_REG (0x07FC)**




**INTERRUPT_COREO_INTERRUPT_DATE** Version control register. (R/W)

# Chapter 9

# Low-power Management

## 9.1 Introduction

ESP32-C3 has an advanced Power Management Unit (PMU), which can flexibly power up different power domains of the chip, to achieve the best balance among chip performance, power consumption, and wakeup latency. To simplify power management for typical scenarios, ESP32-C3 has predefined four power modes, which are preset configurations that power up different combinations of power domains. On top of that, the chip also allows the users to independently power up any particular power domain to meet more complex requirements.

## 9.2 Features

ESP32-C3's low-power management supports the following features:

- Four predefined power modes to simplify power management for typical scenarios
- Up to 8 KB of retention memory
- 8 x 32-bit retention registers
- RTC Boot supported for reduced wakeup latency

In this chapter, we first introduce the working process of ESP32-C3's low-power management, then introduce the predefined power modes of the chip, and at last, introduce the RTC boot of the chip.

## 9.3 Functional Description

ESP32-C3's low-power management involves the following components:

- Power management unit: controls the power supply to Analog, RTC and Digital power domains.
- Power isolation unit: isolates different power domains, so any powered down power domain does not affect the powered up ones.
- Low-power clocks: provide clocks to power domains working in low-power modes.
- RTC timer: logs the status of the RTC main state machine in dedicated registers.
- 8 x 32-bit "always-on" retention registers: These registers are always powered up and are not affected by any low-power modes, thus can be used for storing data that cannot be lost.
- 6 x "always-on" pins: These pins are always powered up and are not affected by any low-power modes, which makes them suitable for working as wakeup sources when the chip is working in the low-power

modes (for details, please refer to Section 9.4.3), or can be used as regular GPIOs (for details, please refer to Chapter 5 *IO MUX and GPIO Matrix (GPIO, IO MUX)*).

- RTC fast memory: 8 KB SRAM that works under CPU clock (CPU_CLK), which can be used as extended memory.
- Voltage regulators: regulate the power supply to different power domains.

The schematic diagram of ESP32-C3's low-power management is shown in Figure 9.3-1.

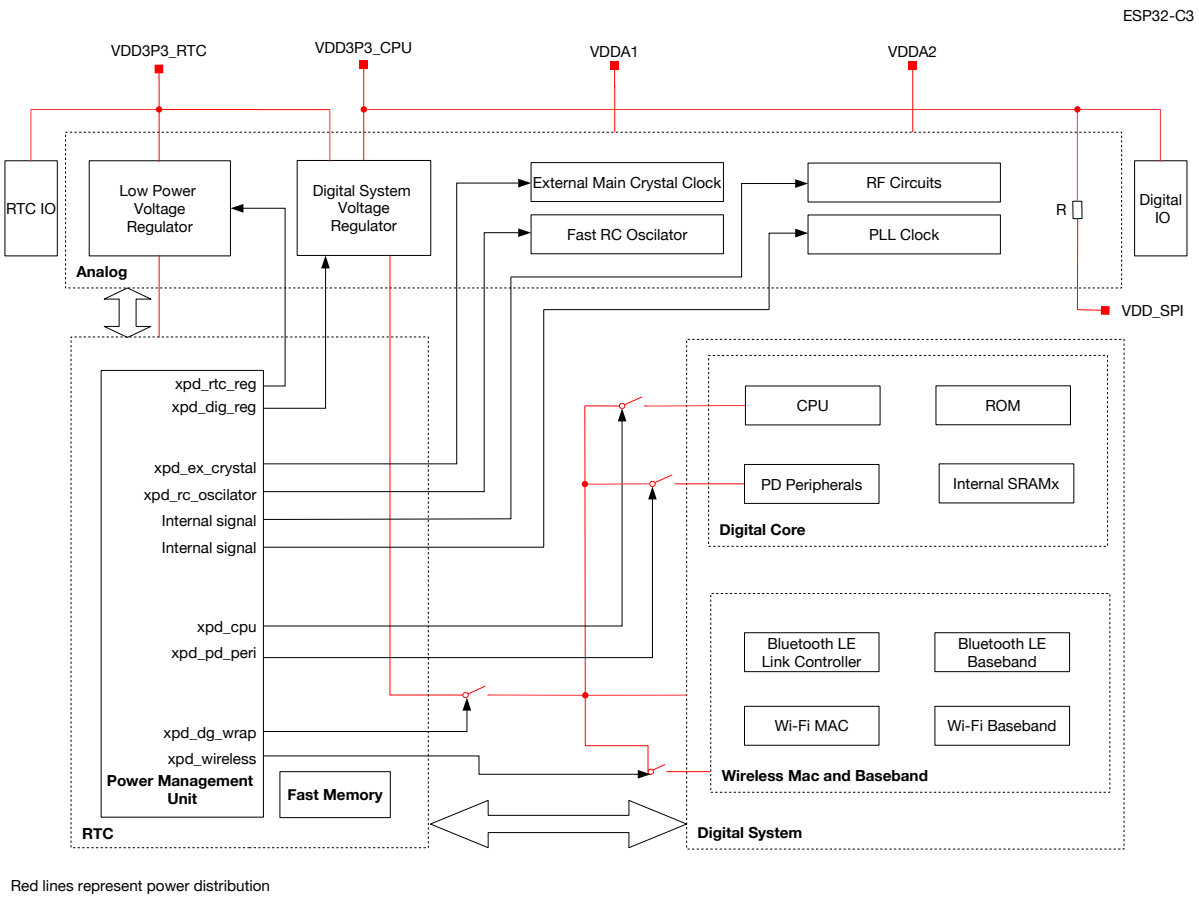



Figure 9.3-1. Low-power Management Schematics

**Note:**

- For more information about different power domains, please check Section 9.4.1.
- Switches in the above diagram can be controlled by Register RTC_CNTL_DIG_PWC_REG.
- Signals in the above diagram are described below:
  - xpd_rtc_reg:
    - When RTC_CNTL_REGULATOR_FORCE_PU is set to 1, low power voltage regulator is always-on;
    - Otherwise, the low power voltage regulator is off when chip enters Light-sleep and Deep-sleep modes. In this case, the RTC domain is powered by an ultra low-power internal power source.
  - xpd_dig_reg:
    - When RTC_CNTL_DG_WRAP_PD_EN is enabled, the digital system voltage regulator is off when the chip enters Light-sleep and Deep-sleep modes;
    - Otherwise, the digital system voltage regulator is always-on.

- xpd_ex_crystal:
  * When RTC_CNTL_XTL_FORCE_PU is set to 1, the external main crystal clock is always-on;
  * Otherwise, the external main crystal clock is off when chip enters Light-sleep and Deep-sleep modes.
- xpd_rc_oscilator:
  * when RTC_CNTL_FOSC_FORCE_PU is set to 1, the fast RC oscillator is always-on;
  * Otherwise, the fast RC oscillator is off when chip enters Light-sleep and Deep-sleep modes.

### 9.3.1 Power Management Unit (PMU)

ESP32-C3's power management unit controls the power supply to different power domains. The main components of the power management unit include:

- RTC main state machine: generates power gating, clock gating, and reset signals.
- Power controllers: power up and power down different power domains, according to the power gating signals from the main state machine.
- Sleep / wakeup controllers: send sleep or wakeup requests to the RTC main state machine.
- Clock controller: selects and powers up/down clock sources.
- Protection Timer: controls the transition interval between main state machine states.

In ESP32-C3's power management unit, the sleep / wakeup controllers send sleep or wakeup requests to the RTC main state machine, which then generates power gating, clock gating, and reset signals. Then, the power controller and clock controller power up and power down different power domains and clock sources, according to the signals generated by the RTC main state machine, so that the chip enters or exits the low-power modes. The main workflow is shown in Figure 9.3-2.

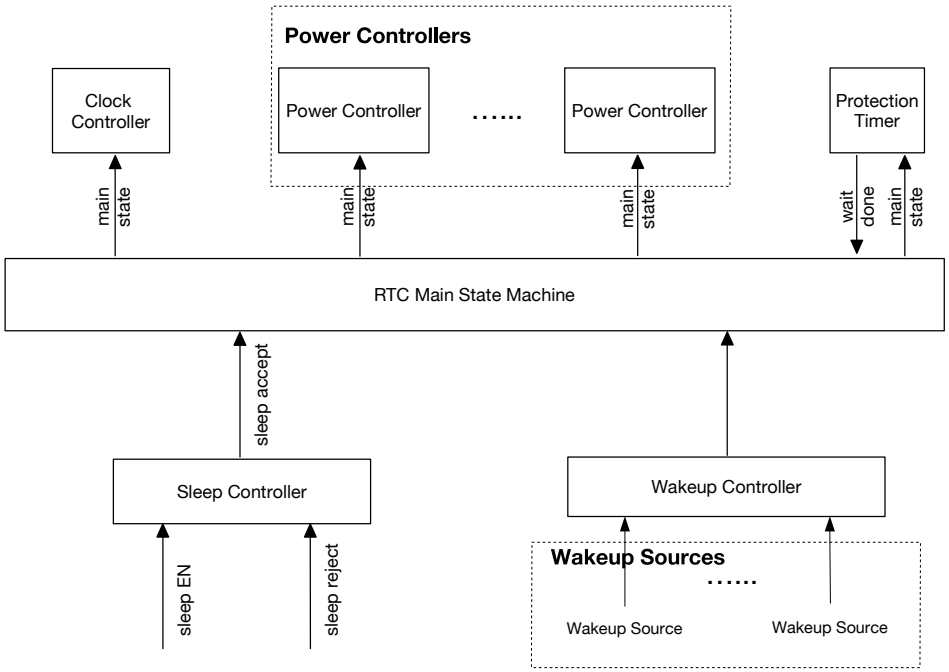



Figure 9.3-2. Power Management Unit Workflow

**Note:**

1. Each power domain has its own power controller. For a complete list of all the available power controllers controlling different power domains, please refer to Section 9.4.1.
2. For a complete list of all the available wakeup sources, please refer to Table 9.4-2.

### 9.3.2 Low-Power Clocks

In general, ESP32-C3 powers down its external main crystal oscillator XTAL_CLK and PLL to reduce power consumption when working in low-power modes. During this time, the chip's low-power clocks remain on to provide clocks to low power domains, such as the power management unit.

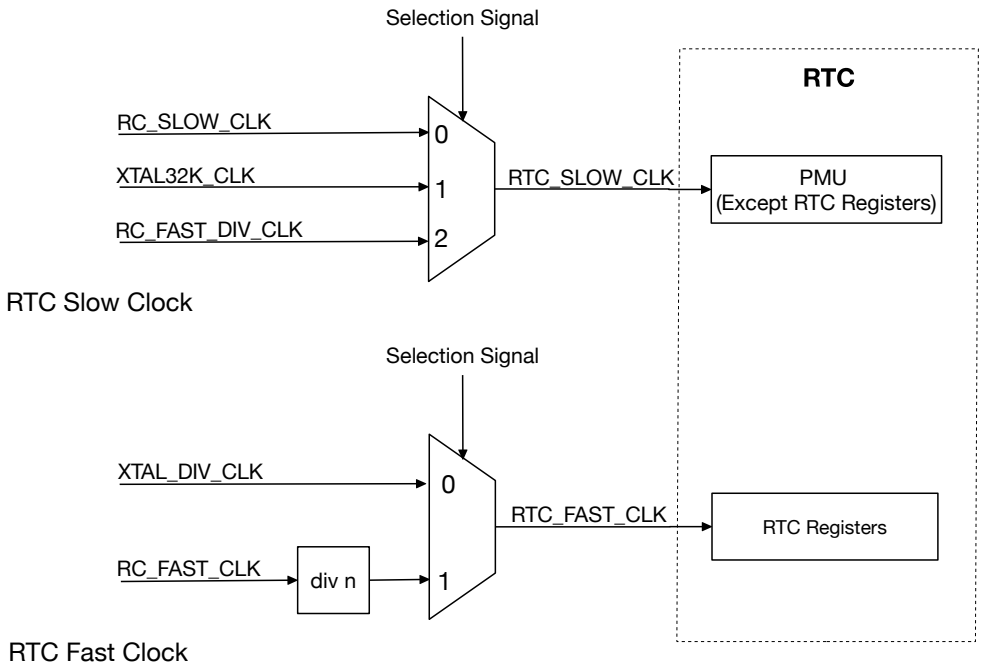



Figure 9.3-3. RTC Clocks

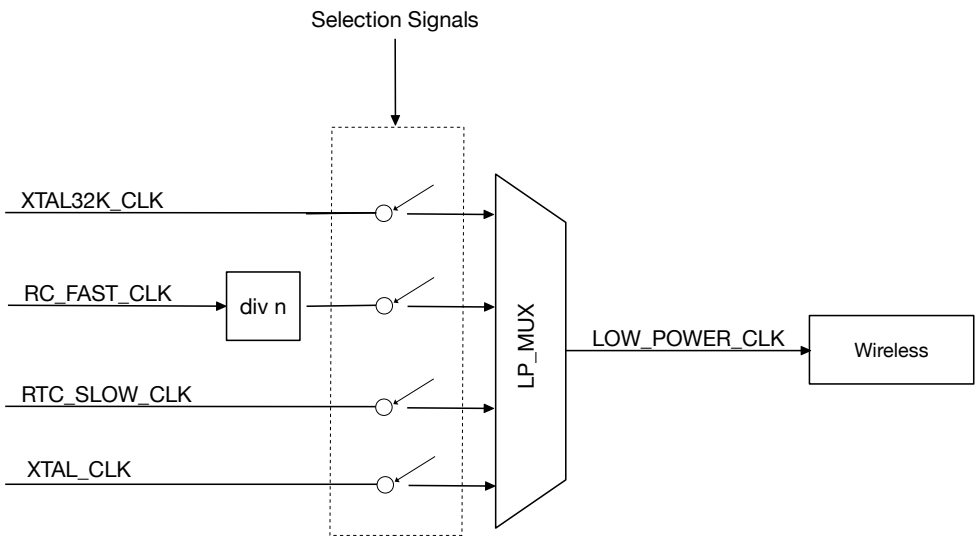



Figure 9.3-4. Wireless Clock

Table 9.3-1. Low-power Clocks

| Clock Type | Clock Source | Selection Signal | Power Domain |
|---|---|---|---|
| RTC Fast Clock | RC_FAST_CLK divided by n (Default) | RTC_CNTL_FAST_CLK_RTC_SEL | RTC Registers |
| | XTAL_DIV_CLK | | |
| RTC Slow Clock | XTAL32K_CLK | RTC_CNTL_ANA_CLK_RTC_SEL | Power Management System (except RTC registers) |
| | RC_FAST_DIV_CLK | | |
| | RC_SLOW_CLK (default) | | |
| Wireless Clock | XTAL32K_CLK | SYSTEM_LPCLK_SEL_XTAL32K | Wireless modules (Wi-Fi/BT) in the digital system domain working in low-power modes |
| | RC_FAST_CLK divided by n | SYSTEM_LPCLK_SEL_20M | |
| | RTC_SLOW_CLK | SYSTEM_LPCLK_RTC_SLOW | |
| | XTAL_CLK | SYSTEM_LPCLK_SEL_XTAL | |

When working under low-power modes, ESP32-C3's XTAL_CLK and PLL are usually powered down to reduce power consumption. However, the low-power clock remains on so the chip can operate properly under low-power modes. For more detailed description about clocks, please refer to 6 *Reset and Clock*.

### 9.3.3 Timers

ESP32-C3's low-power management uses RTC timer. The readable 48-bit RTC timer is a real-time counter (using RTC slow clock) that can be configured to log the time when one of the following events happens. For details, see Table 9.3-2.

Table 9.3-2. The Triggering Conditions for the RTC Timer

| Enabling Options | Descriptions |
|---|---|

| | |
|---|---|
| RTC_CNTL_TIMER_XTL_OFF | 1. RTC main state machine powers down; 2. 40 MHz crystal powers up. |
| RTC_CNTL_TIMER_SYS_STALL | CPU enters or exits the stall state. This is to ensure the SYS_TIMER is continuous in time. |
| RTC_CNTL_TIMER_SYS_RST | Resetting digital system completes. |
| RTC_CNTL_TIME_UPDATE | Register RTC_CNTL_TIME_UPDATE is configured by CPU (i.e. users). |

The RTC timer updates two groups of registers upon any new trigger. The first group logs the time of the current trigger, and the other logs the previous trigger. Detailed information about these two register groups is shown below:

- Register group 0: logs the status of RTC timer at the current trigger.
  - RTC_CNTL_TIME_HIGH0_REG
  - RTC_CNTL_TIME_LOW0_REG
- Register group 1: logs the status of RTC timer at the previous trigger.
  - RTC_CNTL_TIME_HIGH1_REG
  - RTC_CNTL_TIME_LOW1_REG

On a new trigger, information on previous trigger is moved from register group 0 to register group 1 (and the original trigger logged in register group 1 is overwritten), and this new trigger is logged in register group 0. Therefore, only the last two triggers can be logged at any time.

It should be noted that any reset / sleep other than power-up reset will not stop or reset the RTC timer.

Also, the RTC timer can be used as a wakeup source. For details, see Section 9.4.3.

## 9.3.4 Voltage Regulators

ESP32-C3 has two regulators to maintain a constant power supply voltage to different power domains:

- Digital system voltage regulator for digital power domains;
- Low-power voltage regulator for RTC power domains;

**Note:**
For more detailed description about power domains, please refer to Section 9.4.1.

### 9.3.4.1 Digital System Voltage Regulator

ESP32-C3's built-in digital system voltage regulator converts the external power supply (typically 3.3 V) to 1.1 V for digital power domains. This regulator is controlled by the xpd_dig_reg signal. For details, see description in 9.3-1. For the architecture of the ESP32-C3 digital system voltage regulator, see Figure 9.3-5.

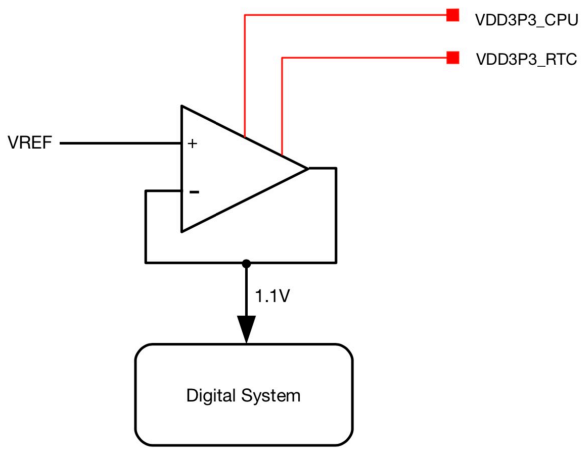



Figure 9.3-5. Digital System Regulator

#### 9.3.4.2 Low-power Voltage Regulator

ESP32-C3's built-in low-power voltage regulator converts the external power supply (typically 3.3 V) to 1.1 V for RTC power domains. Note when the pin CHIP_PU is at a high level, the low-power voltage regulator cannot be turned off. Otherwise, the low power voltage regulator is off when chip enters Light-sleep and Deep-sleep modes. In this case, the RTC domain is powered by an ultra low-power internal power source.

For the architecture of the ESP32-C3 low-power voltage regulator, see Figure 9.3-6.

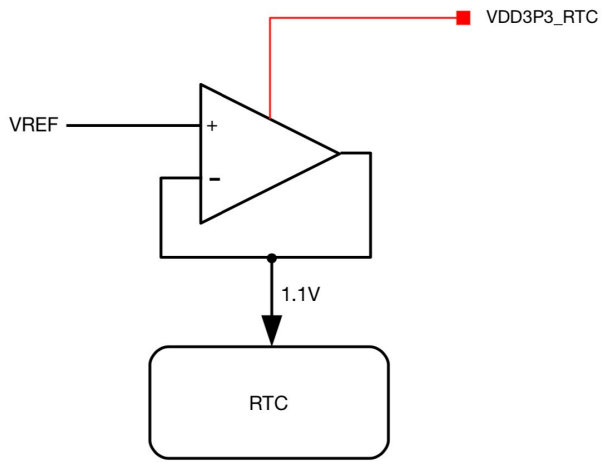



Figure 9.3-6. Low-power voltage regulator

#### 9.3.4.3 Brownout Detector

The brownout detector checks the voltage of pins VDD3P3_RTC, VDD3P3_CPU, VDDA1, and VDDA2. If the voltage of these pins drops below the predefined threshold (2.7 V by default), the detector would trigger a signal to shut down some power-consuming blocks (such as LNA, PA, etc.) to allow extra time for the digital system to save and transfer important data.

RTC_CNTL_BROWN_OUT_DET indicates the output level of brown-out detector. This register is low level by

default, and outputs high level when the voltage of the detected pin drops below the predefined threshold.

RTC_CNTL_BROWN_OUT_RST_SEL configures the reset type. For more information regarding chip reset and system reset, please refer to 6 *Reset and Clock*.

- 0: resets the chip
- 1: resets the system

The brownout detector has ultra-low power consumption and remains enabled whenever the chip is powered up. For the architecture of the ESP32-C3 brownout detector, see Figure 9.3-7.

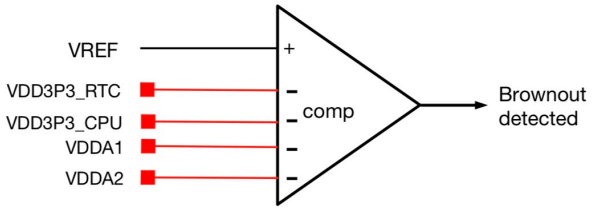



Figure 9.3-7. Brown-out detector

## 9.4 Power Modes Management

### 9.4.1 Power Domain

ESP32-C3 has 9 power domains in three power domain categories:

- RTC
  - Power management unit (PMU), including RTC timer, fast memory, Always-on registers
- Digital
  - PD peripherals, including SPI2, GDMA, SHA, RSA, AES, HMAC, DS, Secure Boot
  - Digital system
  - Wireless digital circuits
  - CPU
- Analog
  - RC_FAST_CLK
  - XTAL_CLK
  - PLL
  - RF circuits

### 9.4.2 Pre-defined Power Modes

As mentioned earlier, ESP32-C3 has four power modes, which are predefined configurations that power up different combinations of power domains. For details, please refer to Table 9.4-1.

Table 9.4-1. Predefined Power Modes

| Power Mode | Power Domain | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | PMU | PD Peripherals | Digital System | Wireless Digital Circuits | CPU | FOSC_ CLK | XTAL_ CLK | PLL | RF Circuits |
| Active | ON | ON | ON | ON | ON | ON | ON | ON | ON |
| Modem-sleep | ON | ON | ON | ON* | ON | ON | ON | ON | OFF |
| Light-sleep | ON | ON | ON | OFF* | OFF* | OFF* | OFF | OFF | OFF |
| Deep-sleep | ON | OFF | OFF | OFF | OFF | OFF | OFF | OFF | OFF |

* Configurable

By default, ESP32-C3 first enters the Modem-sleep mode after a system reset and can be configured to Active mode when transmitting or receiving packets. After the CPU stalls for a while, the chip can enter different low-power modes (including Modem-sleep, Light-sleep, and Deep-sleep) to save power. From Active to Deep-sleep, the number of available functionalities [1] and power consumption[2] decreases and wakeup latency increases. Also, the supported wakeup sources for different power modes are different[3]. Users can choose a power mode based on their requirements of functionality, power consumption, wakeup latency, and available wakeup sources.

**Note:**

1. For details, please refer to Table 9.4-1.
2. For details on power consumption, please refer to the Current Consumption Characteristics in *ESP32-C3 Datasheet*.
3. For details on the supported wakeup sources, please refer to Section 9.4.3.

### 9.4.3 Wakeup Source

The ESP32-C3 supports various wakeup sources, which could wake up the CPU in different sleep modes. The wakeup source is determined by RTC_CNTL_WAKEUP_ENA as shown in Table 9.4-2.

Table 9.4-2. Wakeup Source

| WAKEUP_ENA | Wakeup Source[5] | Light-sleep | Deep-sleep |
|---|---|---|---|
| 0x4 | GPIO[1] | Y | Y |
| 0x8 | RTC Timer | Y | Y |
| 0x20 | Wi-Fi[2] | Y | - |
| 0x40 | UART0[3] | Y | - |
| 0x80 | UART1[3] | Y | - |
| 0x400 | Bluetooth | Y | - |
| 0x1000 | XTAL32K_CLK[4] | Y | Y |

[1] In Deep-sleep mode, only the RTC GPIOs (not regular GPIOs) can work as a wakeup source.

[2] To wake up the chip with a Wi-Fi source, the chip switches between the Active, Modem-sleep, and Light-sleep modes. The CPU and RF modules are woken up at predetermined intervals to keep Wi-Fi connections active.

[3] A wakeup is triggered when the number of RX pulses received exceeds the setting in the threshold register UART_SLEEP_CONF_REG. For details, please refer to Chapter *26 UART Controller (UART)*.

[4] When the 32 kHz crystal is working as RTC slow clock, a wakeup is triggered upon any detection of any crystal stop by the 32 kHz watchdog timer.

[5] All wakeup sources can also be configured as the causes to reject sleep, except UART.

### 9.4.4 Reject Sleep

ESP32-C3 implements a hardware mechanism that equips the chip with the ability to reject to sleep, which prevents the chip from going to sleep unexpectedly when some peripherals are still working but not detected by the CPU, thus guaranteeing the proper functioning of the peripherals.

All the wakeup sources specified in Table 9.4-2 (except UART) can also be configured as the causes to reject sleep.

Users can configure the reject to sleep option via the following registers.

- Configure the RTC_CNTL_SLEEP_REJECT_ENA field to enable or disable the option to reject to sleep:
  - Set RTC_CNTL_LIGHT_SLP_REJECT_EN to enable reject-to-light-sleep.
  - Set RTC_CNTL_DEEP_SLP_REJECT_EN to enable reject-to-deep-sleep.
- Read RTC_CNTL_SLP_REJECT_CAUSE_REG to check the reason for rejecting to sleep.

## 9.5 Retention DMA

ESP32-C3 can power off the CPU in Light_sleep mode to further reduce the power consumption. To facilitate the CPU to wake up from light_sleep and resume execution from the previous breakpoint, ESP32-C3

introduced a retention module.

ESP32-C3's retention module stores CPU information to the Internal SRAM Block9 to Block12 before CPU enters into sleep, and restore such information from Internal SRAM to CPU after CPU wakes up from sleep, thus enabling the CPU to resume execution from the previous breakpoint.

ESP32-C3's Retention DMA:

- Retention DMA operates 128-bit wide data, and only supports address alignment of four words.
- Retention DMA's link list is specifically designed that it can be used to execute both write and read transactions. The configuration of Retention DMA is similar to that of GDMA:
  1. First allocate enough memory in SRAM before CPU enters sleep to store 432 words*: CPU registers (428 words) and configuration information (4 words).
  2. Then configure the link list according to the memory allocated in the first step. See details in Chapter 2 *GDMA Controller (GDMA)*.

> **Note:**
> * Note that if the memory allocated is smaller than 432 words, then chip can only enter the Light_sleep mode and cannot further power down CPU.

After configuration, users can enable the Retention function by configuring the RTC_CNTL_RETENTION_EN field in Register RTC_CNTL_RETENTION_CTRL_REG to:

- Use Retention DMA to store CPU information before the chip enters sleep
- Restore information from Retention DMA to CPU after CPU wakes up.

## 9.6 RTC Boot

The wakeup time from Deep-sleep is much longer, compared to the Light-sleep and Modem-sleep, because the ROMs and RAMs are both powered down in this case, and the CPU needs more time for SPI booting. However, it's worth noting that the RTC fast memory remains powered up in the Deep-sleep mode. Therefore, users can store code (so called "deep sleep wake stub" of up to 8 KB) in the RTC fast memory to avoid the above-mentioned SPI booting, thus speeding up the wakeup process. To use this function, see steps described below:

1. Set RTC_CNTL_STAT_VECTOR_SEL_PROCPU to 1.
2. Calculate CRC for the RTC fast memory, and save the result in RTC_CNTL_STORE7_REG[31:0].
3. Set RTC_CNTL_STORE6_REG[31:0] to the entry address of RTC fast memory.
4. Send the chip into sleep.
5. SPI boot and some of the initialization starts after the CPU is powered up. After that, calculate the CRC for the RTC fast memory again. If the result matches with register RTC_CNTL_STORE7_REG[31:0], the CPU jumps to the entry address.

The boot flow after ESP32-C3 wakeup is shown in Figure 9.6-1.

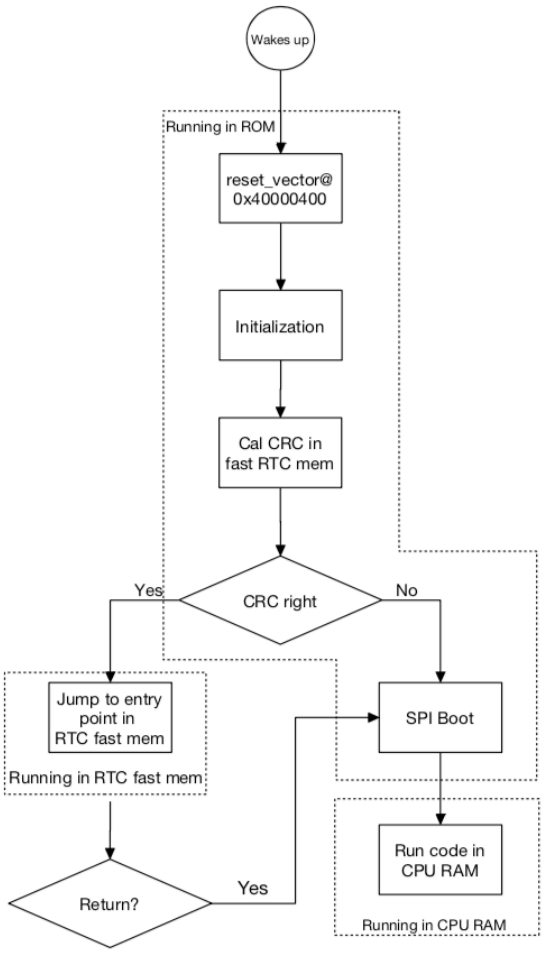



Figure 9.6-1. ESP32-C3 Boot Flow

## 9.7 Register Summary

The addresses in this section are relative to low-power management base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| RTC_CNTL_OPTIONS0_REG | Sets the power options of crystal and PLL clocks, and initiates reset by software | 0x0000 | Varies |
| RTC_CNTL_SLP_TIMER0_REG | RTC timer threshold register 0 | 0x0004 | R/W |
| RTC_CNTL_SLP_TIMER1_REG | RTC timer threshold register 1 | 0x0008 | varies |
| RTC_CNTL_TIME_UPDATE_REG | RTC timer update control register | 0x000C | varies |
| RTC_CNTL_TIME_LOW0_REG | Stores the lower 32 bits of RTC timer 0 | 0x0010 | RO |
| RTC_CNTL_TIME_HIGH0_REG | Stores the higher 16 bits of RTC timer 0 | 0x0014 | RO |
| RTC_CNTL_STATE0_REG | Configures the sleep / reject / wakeup state | 0x0018 | varies |
| RTC_CNTL_TIMER1_REG | Configures CPU stall options | 0x001C | R/W |
| RTC_CNTL_TIMER2_REG | Configures RTC slow clock and touch controller | 0x0020 | R/W |
| RTC_CNTL_TIMER5_REG | Configures the minimal sleep cycles | 0x002C | R/W |
| RTC_CNTL_ANA_CONF_REG | Configures the power options for I2C and PLLA | 0x0034 | R/W |
| RTC_CNTL_RESET_STATE_REG | Indicates the CPU reset source | 0x0038 | varies |
| RTC_CNTL_WAKEUP_STATE_REG | Wakeup bitmap enabling register | 0x003C | R/W |
| RTC_CNTL_INT_ENA_RTC_REG | RTC interrupt enabling register | 0x0040 | R/W |
| RTC_CNTL_INT_RAW_RTC_REG | RTC interrupt raw register | 0x0044 | RO |
| RTC_CNTL_INT_ST_RTC_REG | RTC interrupt state register | 0x0048 | RO |
| RTC_CNTL_INT_CLR_RTC_REG | RTC interrupt clear register | 0x004C | WO |
| RTC_CNTL_STORE0_REG | Reservation register 0 | 0x0050 | R/W |
| RTC_CNTL_STORE1_REG | Reservation register 1 | 0x0054 | R/W |
| RTC_CNTL_STORE2_REG | Reservation register 2 | 0x0058 | R/W |
| RTC_CNTL_STORE3_REG | Reservation register 3 | 0x005C | R/W |
| RTC_CNTL_EXT_XTL_CONF_REG | 32 kHz crystal oscillator configuration register | 0x0060 | varies |
| RTC_CNTL_EXT_WAKEUP_CONF_REG | GPIO wakeup configuration register | 0x0064 | R/W |
| RTC_CNTL_SLP_REJECT_CONF_REG | Configures sleep / reject options | 0x0068 | R/W |
| RTC_CNTL_CLK_CONF_REG | RTC timer configuration register | 0x0070 | R/W |
| RTC_CNTL_SLOW_CLK_CONF_REG | RTC slow clock configuration register | 0x0074 | R/W |
| RTC_CNTL_REG | RTC configuration register | 0x0080 | R/W |
| RTC_CNTL_PWC_REG | RTC power configuration register | 0x0084 | R/W |
| RTC_CNTL_DIG_PWC_REG | Digital system power configuration register | 0x0088 | R/W |
| RTC_CNTL_DIG_ISO_REG | Digital system isolation configuration register | 0x008C | varies |
| RTC_CNTL_WDTCONFIG0_REG | RTC watchdog configuration register | 0x0090 | R/W |
| RTC_CNTL_WDTCONFIG1_REG | Configures the hold time of RTC watchdog at level 1 | 0x0094 | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| RTC_CNTL_WDTCONFIG2_REG | Configures the hold time of RTC watchdog at level 2 | 0x0098 | R/W |
| RTC_CNTL_WDTCONFIG3_REG | Configures the hold time of RTC watchdog at level 3 | 0x009C | R/W |
| RTC_CNTL_WDTCONFIG4_REG | Configures the hold time of RTC watchdog at level 4 | 0x00A0 | R/W |
| RTC_CNTL_WDTFEED_REG | RTC watchdog SW feed configuration register | 0x00A4 | WO |
| RTC_CNTL_WDTWPROTECT_REG | RTC watchdog write protection configuration register | 0x00A8 | R/W |
| RTC_CNTL_SWD_CONF_REG | Super watchdog configuration register | 0x00AC | varies |
| RTC_CNTL_SWD_WPROTECT_REG | Super watchdog write protection configuration register | 0x00B0 | R/W |
| RTC_CNTL_SW_CPU_STALL_REG | CPU stall configuration register | 0x00B4 | R/W |
| RTC_CNTL_STORE4_REG | Reservation register 4 | 0x00B8 | R/W |
| RTC_CNTL_STORE5_REG | Reservation register 5 | 0x00BC | R/W |
| RTC_CNTL_STORE6_REG | Reservation register 6 | 0x00C0 | R/W |
| RTC_CNTL_STORE7_REG | Reservation register 7 | 0x00C4 | R/W |
| RTC_CNTL_LOW_POWER_ST_REG | RTC main state machine state register | 0x00C8 | RO |
| RTC_CNTL_PAD_HOLD_REG | Configures the hold options for RTC GPIOs | 0x00D0 | R/W |
| RTC_CNTL_DIG_PAD_HOLD_REG | Configures the hold options for digital GPIOs | 0x00D4 | R/W |
| RTC_CNTL_BROWN_OUT_REG | Brownout configuration register | 0x00D8 | varies |
| RTC_CNTL_TIME_LOW1_REG | Stores the lower 32 bits of RTC timer 1 | 0x00DC | RO |
| RTC_CNTL_TIME_HIGH1_REG | Stores the higher 16 bits of RTC timer 1 | 0x00E0 | RO |
| RTC_CNTL_XTAL32K_CLK_FACTOR_REG | Configures the divider for the backup clock of 32 kHz crystal oscillator | 0x00E4 | R/W |
| RTC_CNTL_XTAL32K_CONF_REG | 32 kHz crystal oscillator configuration register | 0x00E8 | R/W |
| RTC_CNTL_USB_CONF_REG | IO_MUX configuration register | 0x00EC | R/W |
| RTC_CNTL_SLP_REJECT_CAUSE_REG | Stores the reject-to-sleep cause | 0x00F0 | RO |
| RTC_CNTL_OPTION1_REG | RTC option register | 0x00F4 | R/W |
| RTC_CNTL_SLP_WAKEUP_CAUSE_REG | Stores the sleep-to-wakeup cause | 0x00F8 | RO |
| RTC_CNTL_INT_ENA_RTC_W1TS_REG | RTC RTC interrupt enabling register (W1TS) | 0x0100 | WO |
| RTC_CNTL_INT_ENA_RTC_W1TC_REG | RTC RTC interrupt clear register (W1TC) | 0x0104 | WO |
| RTC_CNTL_RETENTION_CTRL_REG | Retention configuration register | 0x0108 | R/W |
| RTC_CNTL_GPIO_WAKEUP_REG | GPIO wakeup configuration register | 0x0110 | varies |
| RTC_CNTL_SENSOR_CTRL_REG | SAR ADC control register | 0x011C | R/W |

## 9.8 Registers

The addresses in this section are relative to low-power management base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 9.1. RTC_CNTL_OPTIONS0_REG (0x0000)**

| 31 | 30 | 29 | 28 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RTC_CNTL_SW_SYS_RST | RTC_CNTL_DG_WRAP_FORCE_NORST | RTC_CNTL_DG_WRAP_FORCE_RST | (reserved) | | RTC_CNTL_XTL_FORCE_PU | RTC_CNTL_XTL_FORCE_PD | RTC_CNTL_BBPLL_FORCE_PU | RTC_CNTL_BBPLL_FORCE_PD | RTC_CNTL_BBPLL_I2C_FORCE_PU | RTC_CNTL_BBPLL_I2C_FORCE_PD | RTC_CNTL_BB_I2C_FORCE_PU | RTC_CNTL_BB_I2C_FORCE_PD | RTC_CNTL_SW_PROCPU_RST | (reserved) | RTC_CNTL_SW_STALL_PROCPU_C0 | | (reserved) | | |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | | Reset |

**RTC_CNTL_SW_STALL_PROCPU_C0** When RTC_CNTL_SW_STALL_PROCPU_C1 is configured to 0x21, setting this bit to 0x2 stalls the CPU by SW. (R/W)

**RTC_CNTL_SW_PROCPU_RST** Set this bit to reset the CPU by SW. (WO)

**RTC_CNTL_BB_I2C_FORCE_PD** Set this bit to FPD BB_I2C. (R/W)

**RTC_CNTL_BB_I2C_FORCE_PU** Set this bit to FPU BB_I2C. (R/W)

**RTC_CNTL_BBPLL_I2C_FORCE_PD** Set this bit to FPD BB_PLL_I2C. (R/W)

**RTC_CNTL_BBPLL_I2C_FORCE_PU** Set this bit to FPU BB_PLL_I2C. (R/W)

**RTC_CNTL_BBPLL_FORCE_PD** Set this bit to FPD BB_PLL. (R/W)

**RTC_CNTL_BBPLL_FORCE_PU** Set this bit to FPU BB_PLL. (R/W)

**RTC_CNTL_XTL_FORCE_PD** Set this bit to FPD the crystal oscillator. (R/W)

**RTC_CNTL_XTL_FORCE_PU** Set this bit to FPU the crystal oscillator. (R/W)

**RTC_CNTL_DG_WRAP_FORCE_RST** Set this bit to force reset the digital system in deep-sleep. (R/W)

**RTC_CNTL_DG_WRAP_FORCE_NORST** Set this bit to disable force reset to digital system in deep-sleep. (R/W)

**RTC_CNTL_SW_SYS_RST** Set this bit to reset the system via SW. (WO)

### Register 9.2. RTC_CNTL_SLP_TIMER0_REG (0x0004)

| RTC_CNTL_SLP_VAL_LO | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**RTC_CNTL_SLP_VAL_LO** Sets the lower 32 bits of the trigger threshold for the RTC timer. (R/W)

### Register 9.3. RTC_CNTL_SLP_TIMER1_REG (0x0008)

| (reserved) | RTC_CNTL_MAIN_TIMER_ALARM_EN | RTC_CNTL_SLP_VAL_HI | |
|---|---|---|---|
| 31 … 17 | 16 | 15 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0x00 | Reset |

**RTC_CNTL_SLP_VAL_HI** Sets the higher 16 bits of the trigger threshold for the RTC timer. (R/W)

**RTC_CNTL_MAIN_TIMER_ALARM_EN** Sets this bit to enable the timer alarm. (WO)

### Register 9.4. RTC_CNTL_TIME_UPDATE_REG (0x000C)

| RTC_CNTL_TIME_UPDATE | (reserved) | RTC_CNTL_TIMER_SYS_RST | RTC_CNTL_TIMER_XTL_OFF | RTC_CNTL_TIMER_SYS_STALL | (reserved) | |
|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 … 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 0 0 0 0 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 |

**RTC_CNTL_TIMER_SYS_STALL** Selects the triggering condition for the RTC timer. (R/W)

**RTC_CNTL_TIMER_XTL_OFF** Selects the triggering condition for the RTC timer. (R/W)

**RTC_CNTL_TIMER_SYS_RST** Selects the triggering condition for the RTC timer. (R/W)

**RTC_CNTL_TIME_UPDATE** Selects the triggering condition for the RTC timer. (WO)

**Register 9.5. RTC_CNTL_TIME_LOW0_REG (0x0010)**

| RTC_CNTL_TIMER_VALUE0_LOW (31:0) | |
|---|---|
| 0x000000 | Reset |

**RTC_CNTL_TIMER_VALUE0_LOW** Stores the lower 32 bits of RTC timer 0. (RO)

**Register 9.6. RTC_CNTL_TIME_HIGH0_REG (0x0014)**

| (reserved) (31:16) | RTC_CNTL_TIMER_VALUE0_HIGH (15:0) | |
|---|---|---|
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x00 | Reset |

**RTC_CNTL_TIMER_VALUE0_HIGH** Stores the higher 16 bits of RTC timer 0. (RO)

**Register 9.7. RTC_CNTL_STATE0_REG (0x0018)**

| RTC_CNTL_SLEEP_EN | RTC_CNTL_SLP_REJECT | RTC_CNTL_SLP_WAKEUP | (reserved) | RTC_CNTL_APB2RTC_BRIDGE_SEL | (reserved) | RTC_CNTL_SLP_REJECT_CAUSE_CLR | RTC_CNTL_SW_CPU_INT | |
|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 27 23 | 22 | 21 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 | Reset |

**RTC_CNTL_SW_CPU_INT** Sends a SW RTC interrupt to CPU. (WO)

**RTC_CNTL_SLP_REJECT_CAUSE_CLR** Clears the RTC reject-to-sleep cause. (WO)

**RTC_CNTL_APB2RTC_BRIDGE_SEL** 1: APB to RTC using bridge (R/W)

**RTC_CNTL_SLP_WAKEUP** Sleep wakeup bit. (R/W)

**RTC_CNTL_SLP_REJECT** Sleep reject bit. (R/W)

**RTC_CNTL_SLEEP_EN** Sends the chip to sleep. (R/W)

**Register 9.8. RTC_CNTL_TIMER1_REG (0x001C)**

| RTC_CNTL_PLL_BUF_WAIT | RTC_CNTL_XTL_BUF_WAIT | RTC_CNTL_FOSC_WAIT | RTC_CNTL_CPU_STALL_WAIT | RTC_CNTL_CPU_STALL_EN | |
|---|---|---|---|---|---|
| 31 24 | 23 14 | 13 6 | 5 1 | 0 | |
| 40 | 80 | 0x10 | 1 | 1 | Reset |

**RTC_CNTL_CPU_STALL_EN** Enables the CPU stalling. (R/W)

**RTC_CNTL_CPU_STALL_WAIT** Sets the CPU stall waiting cycles (using the RTC fast clock). (R/W)

**RTC_CNTL_FOSC_WAIT** Sets the FOSC clock waiting cycles (using the RTC slow clock). (R/W)

**RTC_CNTL_XTL_BUF_WAIT** Sets the XTAL waiting cycles (using the RTC slow clock). (R/W)

**RTC_CNTL_PLL_BUF_WAIT** Sets the PLL waiting cycles (using the RTC slow clock). (R/W)

**Register 9.9. RTC_CNTL_TIMER2_REG (0x0020)**

| RTC_CNTL_MIN_TIME_FOSC_OFF | (reserved) | |
|---|---|---|
| 31 24 | 23 0 | |
| 0x1 | 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 |

**RTC_CNTL_MIN_TIME_FOSC_OFF** Sets the minimal cycles for FOSC clock (using the RTC slow clock) when powered down. (R/W)

**Register 9.10. RTC_CNTL_TIMER5_REG (0x002C)**

| (reserved) | RTC_CNTL_MIN_SLP_VAL | (reserved) | |
|---|---|---|---|
| 31 16 | 15 8 | 7 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x80 | 0 0 0 0 0 0 0 0 | Reset |

**RTC_CNTL_MIN_SLP_VAL** Sets the minimal sleep cycles (using the RTC slow clock). (R/W)

**Register 9.11. RTC_CNTL_ANA_CONF_REG (0x0034)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 … 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RTC_CNTL_PLL_I2C_PU | RTC_CNTL_CKGEN_I2C_PU | (reserved) | RTC_CNTL_RFRX_PBUS_PU | RTC_CNTL_TXRF_I2C_PU | (reserved) | (reserved) | (reserved) | (reserved) | RTC_CNTL_SAR_I2C_PU | (reserved) | RTC_CNTL_GLITCH_RST_EN | RTC_CNTL_RESET_POR_FORCE_PU | RTC_CNTL_RESET_POR_FORCE_PD | (reserved) | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | Reset |

**RTC_CNTL_RESET_POR_FORCE_PD** Set this bit to force not bypass I2C power-on reset. (R/W)

**RTC_CNTL_RESET_POR_FORCE_PU** Set this bit to force bypass I2C power-on reset. (R/W)

**RTC_CNTL_GLITCH_RST_EN** Set this bit to enable reset when the system detects a glitch. (R/W)

**RTC_CNTL_SAR_I2C_PU** Set this bit to FPU the SAR_I2C. (R/W)

**RTC_CNTL_TXRF_I2C_PU** Set this bit to PU TXRF_I2C. (R/W)

**RTC_CNTL_RFRX_PBUS_PU** Set this bit to PU RFRX_PBUS. (R/W)

**RTC_CNTL_CKGEN_I2C_PU** Set this bit to PU CKGEN_I2C. (R/W)

**RTC_CNTL_PLL_I2C_PU** Set this bit to PU PLL I2C. (R/W)

**Register 9.12. RTC_CNTL_RESET_STATE_REG (0x0038)**

| 31–26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11–6 | 5–0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | RTC_CNTL_DRESET_MASK_PROCPU | (reserved) | (reserved) | RTC_CNTL_JTAG_RESET_FLAG_CLR_PROCPU | (reserved) | RTC_CNTL_JTAG_RESET_FLAG_PROCPU | RTC_CNTL_OCD_HALT_ON_RESET_PROCPU | (reserved) | (reserved) | RTC_CNTL_ALL_RESET_FLAG_CLR_PROCPU | (reserved) | RTC_CNTL_ALL_RESET_FLAG_PROCPU | RTC_CNTL_STAT_VECTOR_SEL_PROCPU | (reserved) | (reserved) | RTC_CNTL_RESET_CAUSE_PROCPU | |
| 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | Reset |

**RTC_CNTL_RESET_CAUSE_PROCPU** Stores the CPU reset cause. (RO)

**RTC_CNTL_STAT_VECTOR_SEL_PROCPU** Selects the CPU static vector. (R/W)

**RTC_CNTL_ALL_RESET_FLAG_PROCPU** Indicates the CPU reset flag. (RO)

**RTC_CNTL_ALL_RESET_FLAG_CLR_PROCPU** Clears the CPU reset flag. (WO)

**RTC_CNTL_OCD_HALT_ON_RESET_PROCPU** Set this bit to send CPU into halt state upon CPU reset. (R/W)

**RTC_CNTL_JTAG_RESET_FLAG_PROCPU** Indicates the JTAG reset flag. (RO)

**RTC_CNTL_JTAG_RESET_FLAG_CLR_PROCPU** Sets the JTAG reset flag. (WO)

**RTC_CNTL_DRESET_MASK_PROCPU** Set this bit to bybass D-reset. (R/W)

**Register 9.13. RTC_CNTL_WAKEUP_STATE_REG (0x003C)**

| 31–15 | 14–0 | |
|---|---|---|
| RTC_CNTL_WAKEUP_ENA | (reserved) | |
| 12 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | Reset |

**RTC_CNTL_WAKEUP_ENA** Selects the wakeup source. For details, please refer to Table 9.4-2. (R/W)

**Register 9.14. RTC_CNTL_INT_ENA_RTC_REG (0x0040)**

| 31–21 | 20 | 19 | 18–17 | 16 | 15 | 14–11 | 10 | 9 | 8–4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | RTC_CNTL_BBPLL_CAL_INT_ENA | RTC_CNTL_GLITCH_DET_INT_ENA | (reserved) | RTC_CNTL_XTAL32K_DEAD_INT_ENA | RTC_CNTL_SWD_INT_ENA | (reserved) | RTC_CNTL_MAIN_TIMER_INT_ENA | RTC_CNTL_BROWN_OUT_INT_ENA | (reserved) | RTC_CNTL_WDT_INT_ENA | (reserved) | RTC_CNTL_SLP_REJECT_INT_ENA | RTC_CNTL_SLP_WAKEUP_INT_ENA | |
| 0 0 0 0 0 0 0 0 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 |

**RTC_CNTL_SLP_WAKEUP_INT_ENA** Enables interrupts when chip wakes up from sleep. (R/W)

**RTC_CNTL_SLP_REJECT_INT_ENA** Enables interrupts when chip rejects to go to sleep. (R/W)

**RTC_CNTL_WDT_INT_ENA** Enables the RTC watchdog interrupt. (R/W)

**RTC_CNTL_BROWN_OUT_INT_ENA** Enables the brown-out interrupt. (R/W)

**RTC_CNTL_MAIN_TIMER_INT_ENA** Enables the RTC timer interrupt. (R/W)

**RTC_CNTL_SWD_INT_ENA** Enables the super watchdog interrupt. (R/W)

**RTC_CNTL_XTAL32K_DEAD_INT_ENA** Enables interrupts when the XTAL32K is dead. (R/W)

**RTC_CNTL_GLITCH_DET_INT_ENA** Enables interrupts when a glitch is detected. (R/W)

**RTC_CNTL_BBPLL_CAL_INT_ENA** Enables interrupts upon the ending of a bb_pll call. (R/W)

**Register 9.15. RTC_CNTL_INT_RAW_RTC_REG (0x0044)**

| 31–21 | 20 | 19 | 18–17 | 16 | 15 | 14–11 | 10 | 9 | 8–4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | RTC_CNTL_BBPLL_CAL_INT_RAW | RTC_CNTL_GLITCH_DET_INT_RAW | (reserved) | RTC_CNTL_XTAL32K_DEAD_INT_RAW | RTC_CNTL_SWD_INT_RAW | (reserved) | RTC_CNTL_MAIN_TIMER_INT_RAW | RTC_CNTL_BROWN_OUT_INT_RAW | (reserved) | RTC_CNTL_WDT_INT_RAW | (reserved) | RTC_CNTL_SLP_REJECT_INT_RAW | RTC_CNTL_SLP_WAKEUP_INT_RAW | |
| 0 0 0 0 0 0 0 0 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 |

**RTC_CNTL_SLP_WAKEUP_INT_RAW** Stores the raw interrupt triggered when the chip wakes up from sleep. (RO)

**RTC_CNTL_SLP_REJECT_INT_RAW** Stores the raw interrupt triggered when the chip rejects to go to sleep. (RO)

**RTC_CNTL_WDT_INT_RAW** Stores the raw watchdog interrupt. (RO)

**RTC_CNTL_BROWN_OUT_INT_RAW** Stores the raw brownout interrupt. (RO)

**RTC_CNTL_MAIN_TIMER_INT_RAW** Stores the raw RTC main timer interrupt. (RO)

**RTC_CNTL_SWD_INT_RAW** Stores the raw super watchdog interrupt. (RO)

**RTC_CNTL_XTAL32K_DEAD_INT_RAW** Stores the raw interrupt triggered when the XTAL32K is dead. (RO)

**RTC_CNTL_GLITCH_DET_INT_RAW** Stores the raw interrupt triggered when a glitch is detected. (RO)

**RTC_CNTL_BBPLL_CAL_INT_RAW** Stores the raw interrupt upon the ending of a bb_pll call. (RO)

**Register 9.16. RTC_CNTL_INT_ST_RTC_REG (0x0048)**

| (reserved) | RTC_CNTL_BBPLL_CAL_INT_ST | RTC_CNTL_GLITCH_DET_INT_ST | (reserved) | RTC_CNTL_XTAL32K_DEAD_INT_ST | RTC_CNTL_SWD_INT_ST | (reserved) | RTC_CNTL_MAIN_TIMER_INT_ST | RTC_CNTL_BROWN_OUT_INT_ST | (reserved) | RTC_CNTL_WDT_INT_ST | (reserved) | RTC_CNTL_SLP_REJECT_INT_ST | RTC_CNTL_SLP_WAKEUP_INT_ST | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 – 21 | 20 | 19 | 18 – 17 | 16 | 15 | 14 – 11 | 10 | 9 | 8 – 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 | Reset |

**RTC_CNTL_SLP_WAKEUP_INT_ST** Stores the status of the interrupt triggered when the chip wakes up from sleep. (RO)

**RTC_CNTL_SLP_REJECT_INT_ST** Stores the status of the interrupt triggered when the chip rejects to go to sleep. (RO)

**RTC_CNTL_WDT_INT_ST** Stores the status of the RTC watchdog interrupt. (RO)

**RTC_CNTL_BROWN_OUT_INT_ST** Stores the status of the brownout interrupt. (RO)

**RTC_CNTL_MAIN_TIMER_INT_ST** Stores the status of the RTC main timer interrupt. (RO)

**RTC_CNTL_SWD_INT_ST** Stores the status of the super watchdog interrupt. (RO)

**RTC_CNTL_XTAL32K_DEAD_INT_ST** Stores the status of the interrupt triggered when the XTAL32K is dead. (RO)

**RTC_CNTL_GLITCH_DET_INT_ST** Stores the status of the interrupt triggered when a glitch is detected. (RO)

**RTC_CNTL_BBPLL_CAL_INT_ST** Stores the status of the interrupt triggered upon the ending of a bbpll call. (RO)

**Register 9.17. RTC_CNTL_INT_CLR_RTC_REG (0x004C)**

| 31–21 | 20 | 19 | 18–17 | 16 | 15 | 14–11 | 10 | 9 | 8–4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | RTC_CNTL_BBPLL_CAL_INT_CLR | RTC_CNTL_GLITCH_DET_INT_CLR | (reserved) | RTC_CNTL_XTAL32K_DEAD_INT_CLR | RTC_CNTL_SWD_INT_CLR | (reserved) | RTC_CNTL_MAIN_TIMER_INT_CLR | RTC_CNTL_BROWN_OUT_INT_CLR | (reserved) | RTC_CNTL_WDT_INT_CLR | (reserved) | RTC_CNTL_SLP_REJECT_INT_CLR | RTC_CNTL_SLP_WAKEUP_INT_CLR | |
| 0 0 0 0 0 0 0 0 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 |

**RTC_CNTL_SLP_WAKEUP_INT_CLR** Clears the interrupt triggered when the chip wakes up from sleep. (WO)

**RTC_CNTL_SLP_REJECT_INT_CLR** Clears the interrupt triggered when the chip rejects to go to sleep. (WO)

**RTC_CNTL_WDT_INT_CLR** Clears the RTC watchdog interrupt. (WO)

**RTC_CNTL_BROWN_OUT_INT_CLR** Clears the brownout interrupt. (WO)

**RTC_CNTL_MAIN_TIMER_INT_CLR** Clears the RTC main timer interrupt. (WO)

**RTC_CNTL_SWD_INT_CLR** Clears the super watchdog interrupt. (WO)

**RTC_CNTL_XTAL32K_DEAD_INT_CLR** Clears the RTC watchdog interrupt. (WO)

**RTC_CNTL_GLITCH_DET_INT_CLR** Clears the interrupt triggered when a glitch is detected. (WO)

**RTC_CNTL_BBPLL_CAL_INT_CLR** Clears the interrupt triggered upon the ending of a bbpll call. (WO)

**Register 9.18. RTC_CNTL_STORE0_REG (0x0050)**

| 31–0 | |
|---|---|
| RTC_CNTL_SCRATCH0 | |
| 0 | Reset |

**RTC_CNTL_SCRATCH0** Reservation register 0. (R/W)

### Register 9.19. RTC_CNTL_STORE1_REG (0x0054)

| RTC_CNTL_SCRATCH1 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**RTC_CNTL_SCRATCH1** Reservation register 1. (R/W)

### Register 9.20. RTC_CNTL_STORE2_REG (0x0058)

| RTC_CNTL_SCRATCH2 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**RTC_CNTL_SCRATCH2** Reservation register 2. (R/W)

### Register 9.21. RTC_CNTL_STORE3_REG (0x005C)

| RTC_CNTL_SCRATCH3 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**RTC_CNTL_SCRATCH3** Reservation register 3. (R/W)

**Register 9.22. RTC_CNTL_EXT_XTL_CONF_REG (0x0060)**

| 31 | 30 | 29 | 23 | 22 | 19 | 16 | 15 | 12 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | (reserved) | (reserved) | RTC_CNTL_XTAL32K_GPIO_SEL | RTC_CNTL_WDT_STATE | RTC_CNTL_DAC_XTAL_32K | RTC_CNTL_XPD_XTAL_32K | RTC_CNTL_DRES_XTAL_32K | RTC_CNTL_DGM_XTAL_32K | RTC_CNTL_DBUF_XTAL_32K | RTC_CNTL_ENCKINIT_XTAL_32K | RTC_CNTL_XTAL32K_XPD_FORCE | RTC_CNTL_XTAL32K_AUTO_RETURN | RTC_CNTL_XTAL32K_AUTO_RESTART | RTC_CNTL_XTAL32K_AUTO_BACKUP | RTC_CNTL_XTAL32K_EXT_CLK_FO | RTC_CNTL_XTAL32K_WDT_RESET | RTC_CNTL_XTAL32K_WDT_CLK_FO | RTC_CNTL_XTAL32K_WDT_EN | |
| 0 | 0 | 0 0 0 0 0 0 | 0 | 0x0 | 3 | 0 | 3 | 3 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**RTC_CNTL_XTAL32K_WDT_EN** Set this bit to enable the XTAL32K watchdog. (R/W)

**RTC_CNTL_XTAL32K_WDT_CLK_FO** Set this bit to FPU the XTAL32K watchdog clock. (R/W)

**RTC_CNTL_XTAL32K_WDT_RESET** Set this bit to reset the XTAL32K watchdog by SW. (R/W)

**RTC_CNTL_XTAL32K_EXT_CLK_FO** Set this bit to FPU the external clock of XTAL32K. (R/W)

**RTC_CNTL_XTAL32K_AUTO_BACKUP** Set this bit to switch to the backup clock when the XTAL32K is dead. (R/W)

**RTC_CNTL_XTAL32K_AUTO_RESTART** Set this bit to restart the XTAL32K automatically when the XTAL32K is dead. (R/W)

**RTC_CNTL_XTAL32K_AUTO_RETURN** Set this bit to switch back to XTAL32K when the XTAL32K is restarted. (R/W)

**RTC_CNTL_XTAL32K_XPD_FORCE** Set this bit to allow the software to FPD the XTAL32K; Reset this bit to allow the FSM to FPD the XTAL32K. (R/W)

**RTC_CNTL_ENCKINIT_XTAL_32K** Set this bit to apply an internal clock to help the XTAL32K to start. (R/W)

**RTC_CNTL_DBUF_XTAL_32K** 0: single-end buffer 1: differential buffer. (R/W)

**RTC_CNTL_DGM_XTAL_32K** Configures the xtal_32k gm control. (R/W)

**RTC_CNTL_DRES_XTAL_32K** Configures DRES_XTAL_32K. (R/W)

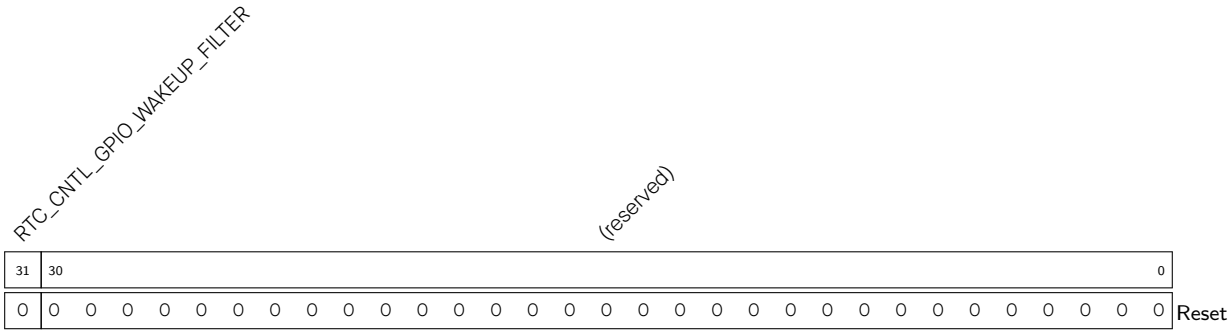
**RTC_CNTL_XPD_XTAL_32K** Configures XPD_XTAL_32K. (R/W)

**RTC_CNTL_DAC_XTAL_32K** Configures DAC_XTAL_32K. (R/W)

**RTC_CNTL_WDT_STATE** Indicates the 32 kHz watchdog timer state. (RO)

**RTC_CNTL_XTAL32K_GPIO_SEL** Set this bit to select the XTAL32K. Clear this bit to select external XTAL32K. (R/W)

Register 9.23. RTC_CNTL_EXT_WAKEUP_CONF_REG (0x0064)




**RTC_CNTL_GPIO_WAKEUP_FILTER** Set this bit to enable the GPIO wakeup event filter. (R/W)

Register 9.24. RTC_CNTL_SLP_REJECT_CONF_REG (0x0068)

**RTC_CNTL_SLEEP_REJECT_ENA** Set this bit to enable reject-to-sleep. (R/W)

**RTC_CNTL_LIGHT_SLP_REJECT_EN** Set this bit to enable reject-to-light-sleep. (R/W)

**RTC_CNTL_DEEP_SLP_REJECT_EN** Set this bit to enable reject-to-deep-sleep. (R/W)

**Register 9.25. RTC_CNTL_CLK_CONF_REG (0x0070)**

| 31 30 | 29 | 28 | 27 | 26 | 25 | 24 17 | 16 | 15 | 14 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RTC_CNTL_ANA_CLK_RTC_SEL | RTC_CNTL_FAST_CLK_RTC_SEL | RTC_CNTL_XTAL_GLOBAL_FORCE_NOGATING | RTC_CNTL_XTAL_GLOBAL_FORCE_GATING | RTC_CNTL_FOSC_FORCE_PU | RTC_CNTL_FOSC_FORCE_PD | RTC_CNTL_FOSC_DFREQ | RTC_CNTL_FOSC_FORCE_NOGATING | RTC_CNTL_XTAL_FORCE_NOGATING | RTC_CNTL_FOSC_DIV_SEL | (reserved) | RTC_CNTL_DIG_FOSC_EN | RTC_CNTL_DIG_FOSC_D256_EN | RTC_CNTL_DIG_XTAL32K_EN | RTC_CNTL_ENB_FOSC_DIV | RTC_CNTL_ENB_FOSC | RTC_CNTL_FOSC_DIV | RTC_CNTL_FOSC_DIV_SEL_VLD | RTC_CNTL_EFUSE_CLK_FORCE_NOGATING | RTC_CNTL_EFUSE_CLK_FORCE_GATING | (reserved) | | |
| 0 | 0 | 1 | 0 | 0 | 0 | 172 | 0 | 0 | 3 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Reset |

**RTC_CNTL_EFUSE_CLK_FORCE_GATING** Set this bit to FPU the eFuse clock gating. (R/W)

**RTC_CNTL_EFUSE_CLK_FORCE_NOGATING** Set this bit to FPD the eFuse clock gating. (R/W)

**RTC_CNTL_FOSC_DIV_SEL_VLD** Synchronizes reg_fosc_div_sel. Note that you have to invalidate the bus before modifying the frequency divider, then validate the new divider clock. (R/W)

**RTC_CNTL_FOSC_DIV** Set the FOSC_D256_OUT divider. 00: divided by 128, 01: divided by 256, 10: divided by 512, 11: divided by 1024. (R/W)

**RTC_CNTL_ENB_FOSC** Set this bit to disable FOSC and FOSC_D256_OUT. (R/W)

**RTC_CNTL_ENB_FOSC_DIV** Selects the FOSC_D256_OUT. 1: FOSC, 0: FOSC divided by 256. (R/W)

**RTC_CNTL_DIG_XTAL32K_EN** Set this bit to enable CK_XTAL_32K clock for the digital system. (R/W)

**RTC_CNTL_DIG_FOSC_D256_EN** Set this bit to enable FOSC_D256_OUT clock for the digital system. (R/W)

**RTC_CNTL_DIG_FOSC_EN** Set this bit to enable FOSC for the digital system. (R/W)

**RTC_CNTL_FOSC_DIV_SEL** Stores the FOSC divider, which is reg_FOSC_div_sel + 1. (R/W)

**RTC_CNTL_XTAL_FORCE_NOGATING** Set this bit to force no gating to crystal during sleep. (R/W)

**RTC_CNTL_FOSC_FORCE_NOGATING** Set this bit to disable force gating to crystal during sleep. (R/W)

Continued on the next page...

Register 9.25. RTC_CNTL_CLK_CONF_REG (0x0070)

Continued from the previous page...

**RTC_CNTL_FOSC_DFREQ** Configures the FOSC frequency. (R/W)

**RTC_CNTL_FOSC_FORCE_PD** Set this bit to FPD FOSC. (R/W)

**RTC_CNTL_FOSC_FORCE_PU** Set this bit to FPU FOSC. (R/W)

**RTC_CNTL_XTAL_GLOBAL_FORCE_GATING** Set this bit to force enable XTAL clock gating. (R/W)

**RTC_CNTL_XTAL_GLOBAL_FORCE_NOGATING** Set this bit to force bypass the XTAL clock gating. (R/W)

**RTC_CNTL_FAST_CLK_RTC_SEL** Selects the RTC fast clock. 0: XTAL_DIV_CLK, 1: RC_FAST_CLK div n. (R/W)

**RTC_CNTL_ANA_CLK_RTC_SEL** Selects the RTC slow clock. 0: RC_SLOW_CLK, 1: XTAL32K_CLK, 2: RC_FAST_DIV_CLK. (R/W)

Register 9.26. RTC_CNTL_SLOW_CLK_CONF_REG (0x0074)

| (reserved) | RTC_CNTL_ANA_CLK_DIV | RTC_CNTL_ANA_CLK_DIV_VLD | (reserved) | |
|---|---|---|---|---|
| 31 | 30 23 | 22 | 21 0 | |
| 0 | 0 | 1 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | Reset |

**RTC_CNTL_ANA_CLK_DIV_VLD** Synchronizes the reg_fosc_div_sel. Note that you have to invalidate the bus before modifying the frequency divider, and then validate the new divider clock. (R/W)

**RTC_CNTL_ANA_CLK_DIV** Configures the divider for the RTC clock. (R/W)4

**Register 9.27. RTC_CNTL_REG (0x0080)**

| RTC_CNTL_REGULATOR_FORCE_PU | RTC_CNTL_REGULATOR_FORCE_PD | (reserved) | (reserved) | (reserved) | RTC_CNTL_SCK_DCAP | (reserved) | RTC_CNTL_DIG_REG_CAL_EN | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 – 22 | 21 – 14 | 13 – 8 | 7 | 6 – 0 | |
| 1 | 0 | 1 | 0 | 0 0 0 0 0 0 | 0 | 0 0 0 0 0 0 | 0 | 0 0 0 0 0 0 0 | Reset |

**RTC_CNTL_DIG_REG_CAL_EN** Set this bit to enable digital regulator calibration by software. (R/W)

**RTC_CNTL_SCK_DCAP** Configures the RC_SLOW_CLK frequency. (R/W)

**RTC_CNTL_REGULATOR_FORCE_PD** Set this bit to FPD the low-power voltage regulator, which means decreasing its voltage to 0.8 V or lower. (R/W)

**RTC_CNTL_REGULATOR_FORCE_PU** Set this bit to FPU the low-power voltage regulator, which means increasing its voltage to higher than 0.8 V. (R/W)

**Register 9.28. RTC_CNTL_PWC_REG (0x0084)**

| (reserved) | RTC_CNTL_PAD_FORCE_HOLD | (reserved) | |
|---|---|---|---|
| 31 – 22 | 21 | 20 – 0 | |
| 0 0 0 0 0 0 0 0 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 |

**RTC_CNTL_PAD_FORCE_HOLD** Set this bit to force RTC pad into hold state. (R/W)

**Register 9.29. RTC_CNTL_DIG_PWC_REG (0x0088)**

| 31 | 30 | 29 | 28 | 27 | 26 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RTC_CNTL_DG_WRAP_PD_EN | RTC_CNTL_WIFI_PD_EN | RTC_CNTL_CPU_TOP_PD_EN | RTC_CNTL_DG_PERI_PD_EN | (reserved) | (reserved) | | RTC_CNTL_CPU_TOP_FORCE_PU | RTC_CNTL_CPU_TOP_FORCE_PD | RTC_CNTL_DG_WRAP_FORCE_PU | RTC_CNTL_DG_WRAP_FORCE_PD | RTC_CNTL_WIFI_FORCE_PU | RTC_CNTL_WIFI_FORCE_PD | RTC_CNTL_FASTMEM_FORCE_LPU | RTC_CNTL_FASTMEM_FORCE_LPD | RTC_CNTL_DG_PERI_FORCE_PU | RTC_CNTL_DG_PERI_FORCE_PD | (reserved) | (reserved) | (reserved) | | RTC_CNTL_LSLP_MEM_FORCE_PU | RTC_CNTL_LSLP_MEM_FORCE_PD | RTC_CNTL_VDD_SPI_PWR_FORCE | RTC_CNTL_VDD_SPI_PWR_DRV | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | | Reset |

**RTC_CNTL_VDD_SPI_PWR_DRV** Configures the vdd_spi's drive intensity. (R/W)

**RTC_CNTL_VDD_SPI_PWR_FORCE** Set this bit to allow software to configure vdd_spi's drive intensity. (R/W)

**RTC_CNTL_LSLP_MEM_FORCE_PD** Set this bit to force the memories in the digital system not to enter retention mode in sleep. (R/W)

**RTC_CNTL_LSLP_MEM_FORCE_PU** Set this bit to force the memories in the digital system into retention mode in sleep. (R/W)

**RTC_CNTL_DG_PERI_FORCE_PD** Set this bit to FPD the digital peripherals. (R/W)

**RTC_CNTL_DG_PERI_FORCE_PU** Set this bit to FPU the digital peripherals. (R/W)

**RTC_CNTL_FASTMEM_FORCE_LPD** Set this bit to force the fast memory not to enter retention mode in sleep. (R/W)

**RTC_CNTL_FASTMEM_FORCE_LPU** Set this bit to force the fast memory into retention mode in sleep. (R/W)

**RTC_CNTL_WIFI_FORCE_PD** Set this bit to FPD wireless. (R/W)

**RTC_CNTL_WIFI_FORCE_PU** Set this bit to FPU wireless. (R/W)

**RTC_CNTL_DG_WRAP_FORCE_PD** Set this bit to FPD the digital system. (R/W)

**RTC_CNTL_DG_WRAP_FORCE_PU** Set this bit to FPU the digital system. (R/W)

**RTC_CNTL_CPU_TOP_FORCE_PD** Set this bit to FPD the CPU. (R/W)

**RTC_CNTL_CPU_TOP_FORCE_PU** Set this bit to FPU the CPU. (R/W)

**RTC_CNTL_DG_PERI_PD_EN** Set this bit to enable FPD digital peripherals in sleep. (R/W)

**RTC_CNTL_CPU_TOP_PD_EN** Set this bit to enable FPD CPU in sleep. (R/W)

**RTC_CNTL_WIFI_PD_EN** Set this bit to enable FPD wireless in sleep. (R/W)

**RTC_CNTL_DG_WRAP_PD_EN** Set this bit to enable FPD digital system in sleep. (R/W)

**Register 9.30. RTC_CNTL_DIG_ISO_REG (0x008C)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | | | | | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | | | | | | | | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RTC_CNTL_DG_WRAP_FORCE_NOISO | RTC_CNTL_DG_WRAP_FORCE_ISO | RTC_CNTL_WIFI_FORCE_NOISO | RTC_CNTL_WIFI_FORCE_ISO | RTC_CNTL_CPU_TOP_FORCE_NOISO | RTC_CNTL_CPU_TOP_FORCE_ISO | RTC_CNTL_DG_PERI_FORCE_NOISO | RTC_CNTL_DG_PERI_FORCE_ISO | (reserved) | (reserved) | (reserved) | | | | | | RTC_CNTL_DG_PAD_FORCE_HOLD | RTC_CNTL_DG_PAD_FORCE_UNHOLD | RTC_CNTL_DG_PAD_FORCE_ISO | RTC_CNTL_DG_PAD_FORCE_NOISO | RTC_CNTL_DG_PAD_AUTOHOLD_EN | RTC_CNTL_CLR_DG_PAD_AUTOHOLD | RTC_CNTL_DG_PAD_AUTOHOLD | (reserved) | | | | | | | | | |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**RTC_CNTL_DG_PAD_AUTOHOLD** Indicates the auto-hold status of the digital GPIOs. (RO)

**RTC_CNTL_CLR_DG_PAD_AUTOHOLD** Set this bit to clear the auto-hold enabler for the digital GPIOs. (WO)

**RTC_CNTL_DG_PAD_AUTOHOLD_EN** Set this bit to allow the digital GPIOs to enter the auto-hold status. (R/W)

**RTC_CNTL_DG_PAD_FORCE_NOISO** Set this bit to disable the force isolation of the digital GPIOs. (R/W)

**RTC_CNTL_DG_PAD_FORCE_ISO** Set this bit to force isolation of the digital GPIOs. (R/W)

**RTC_CNTL_DG_PAD_FORCE_UNHOLD** Set this bit the force unhold the digital GPIOs. (R/W)

**RTC_CNTL_DG_PAD_FORCE_HOLD** Set this bit the force hold the digital GPIOs. (R/W)

**RTC_CNTL_DG_PERI_FORCE_ISO** Set this bit to force isolation of the digital peripherals. (R/W)

**RTC_CNTL_DG_PERI_FORCE_NOISO** Set this bit to disable the force isolation of the digital peripherals. (R/W)

**RTC_CNTL_CPU_TOP_FORCE_ISO** Set this bit to force hold the CPU. (R/W)

**RTC_CNTL_CPU_TOP_FORCE_NOISO** Set this bit to force unhold the CPU. (R/W)

**RTC_CNTL_WIFI_FORCE_ISO** Set this bit to force isolation of the wireless circuits. (R/W)

**RTC_CNTL_WIFI_FORCE_NOISO** Set this bit to disable the force isolation of the wireless circuits. (R/W)

**RTC_CNTL_DG_WRAP_FORCE_ISO** Set this bit to force isolation of the digital system. (R/W)

**RTC_CNTL_DG_WRAP_FORCE_NOISO** Set this bit to disable the force isolation of the digital system. (R/W)

**Register 9.31. RTC_CNTL_WDTCONFIG0_REG (0x0090)**

| RTC_CNTL_WDT_EN | RTC_CNTL_WDT_STG0 | RTC_CNTL_WDT_STG1 | RTC_CNTL_WDT_STG2 | RTC_CNTL_WDT_STG3 | RTC_CNTL_WDT_CPU_RESET_LENGTH | RTC_CNTL_WDT_SYS_RESET_LENGTH | RTC_CNTL_WDT_FLASHBOOT_MOD_EN | (reserved) | RTC_CNTL_WDT_PROCPU_RESET_EN | RTC_CNTL_WDT_PAUSE_IN_SLP | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 28 | 27 25 | 24 22 | 21 19 | 18 16 | 15 13 | 12 | 11 | 10 | 9 | 8 0 | |
| 0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x1 | 0x1 | 1 | 0 | 0 | 1 | 0 0 0 0 0 0 0 0 0 | Reset |

**RTC_CNTL_WDT_PAUSE_IN_SLP** Set this bit to pause the watchdog in sleep. (R/W)

**RTC_CNTL_WDT_PROCPU_RESET_EN** enable WDT reset CPU (R/W)

**RTC_CNTL_WDT_FLASHBOOT_MOD_EN** Set this bit to enable watchdog when the chip boots from flash. (R/W)

**RTC_CNTL_WDT_SYS_RESET_LENGTH** Sets the length of the system reset counter. (R/W)

**RTC_CNTL_WDT_CPU_RESET_LENGTH** Sets the length of the CPU reset counter. (R/W)

**RTC_CNTL_WDT_STG3** 1: enable at the interrupt stage, 2: enable at the CPU stage, 3: enable at the system stage, 4: enable at the system and RTC stage. (R/W)

**RTC_CNTL_WDT_STG2** 1: enable at the interrupt stage, 2: enable at the CPU stage, 3: enable at the system stage, 4: enable at the system and RTC stage. (R/W)

**RTC_CNTL_WDT_STG1** 1: enable at the interrupt stage, 2: enable at the CPU stage, 3: enable at the system stage, 4: enable at the system and RTC stage. (R/W)

**RTC_CNTL_WDT_STG0** 1: enable at the interrupt stage, 2: enable at the CPU stage, 3: enable at the system stage, 4: enable at the system and RTC stage. (R/W)

**RTC_CNTL_WDT_EN** Set this bit to enable the RTC watchdog. (R/W)

**Register 9.32. RTC_CNTL_WDTCONFIG1_REG (0x0094)**

| RTC_CNTL_WDT_STG0_HOLD | |
|---|---|
| 31 0 | |
| 200000 | Reset |

**RTC_CNTL_WDT_STG0_HOLD** Configures the hold time of RTC watchdog at level 1. (R/W)

**Register 9.33. RTC_CNTL_WDTCONFIG2_REG (0x0098)**

| RTC_CNTL_WDT_STG1_HOLD | |
|---|---|
| 31 … 0 | |
| 80000 | Reset |

**RTC_CNTL_WDT_STG1_HOLD** Configures the hold time of RTC watchdog at level 2. (R/W)

**Register 9.34. RTC_CNTL_WDTCONFIG3_REG (0x009C)**

| RTC_CNTL_WDT_STG2_HOLD | |
|---|---|
| 31 … 0 | |
| 0x000fff | Reset |

**RTC_CNTL_WDT_STG2_HOLD** Configures the hold time of RTC watchdog at level 3. (R/W)

**Register 9.35. RTC_CNTL_WDTCONFIG4_REG (0x00A0)**

| RTC_CNTL_WDT_STG3_HOLD | |
|---|---|
| 31 … 0 | |
| 0x000fff | Reset |

**RTC_CNTL_WDT_STG3_HOLD** Configures the hold time of RTC watchdog at level 4. (R/W)

Register 9.36. RTC_CNTL_WDTFEED_REG (0x00A4)

| RTC_CNTL_WDT_FEED | (reserved) | |
|---|---|---|
| 31 | 30 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 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 |

**RTC_CNTL_WDT_FEED** Set this bit to feed the RTC watchdog. (WO)

Register 9.37. RTC_CNTL_WDTWPROTECT_REG (0x00A8)

| RTC_CNTL_WDT_WKEY | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**RTC_CNTL_WDT_WKEY** If the register contains a different value than 0x50d83aa1, write protection for the RTC watchdog (RWDT) is enabled. (R/W)

Register 9.38. RTC_CNTL_SWD_CONF_REG (0x00AC)

| RTC_CNTL_SWD_AUTO_FEED_EN | RTC_CNTL_SWD_DISABLE | RTC_CNTL_SWD_FEED | RTC_CNTL_SWD_RST_FLAG_CLR | RTC_CNTL_SWD_SIGNAL_WIDTH | RTC_CNTL_SWD_BYPASS_RST | (reserved) | RTC_CNTL_SWD_FEED_INT | RTC_CNTL_SWD_RESET_FLAG | |
|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 … 18 | 17 | 16 … 2 | 1 | 0 | |
| 0 | 0 | 0 | 0 | 300 | 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**RTC_CNTL_SWD_RESET_FLAG** Indicates the super watchdog reset flag. (RO)

**RTC_CNTL_SWD_FEED_INT** Receiving this interrupt leads to feeding the super watchdog via SW. (RO)

**RTC_CNTL_SWD_BYPASS_RST** Set this bit to bypass super watchdog reset. (R/W)

**RTC_CNTL_SWD_SIGNAL_WIDTH** Adjusts the signal width sent to the super watchdog. (R/W)

**RTC_CNTL_SWD_RST_FLAG_CLR** Set to reset the super watchdog reset flag. (WO)

**RTC_CNTL_SWD_FEED** Set to feed the super watchdog via SW. (WO)

**RTC_CNTL_SWD_DISABLE** Set this bit to disable super watchdog. (R/W)

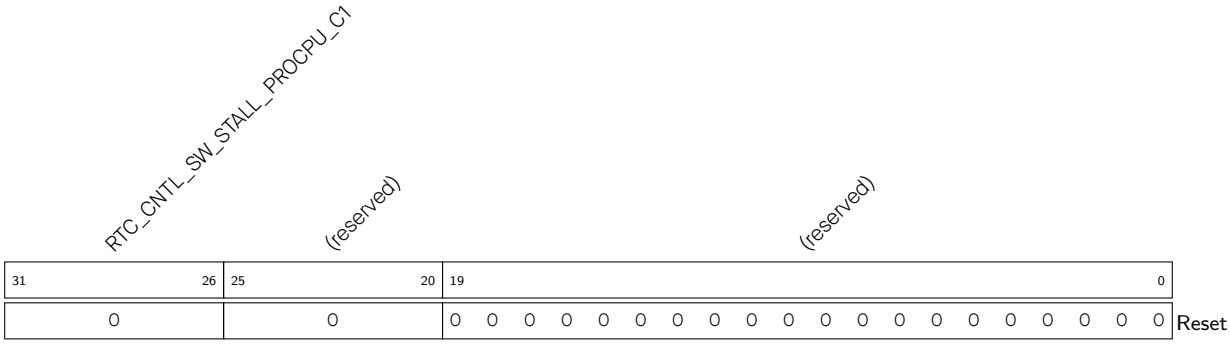
**RTC_CNTL_SWD_AUTO_FEED_EN** Set this bit to enable automatic watchdog feeding upon interrupts. (R/W)

Register 9.39. RTC_CNTL_SWD_WPROTECT_REG (0x00B0)

| RTC_CNTL_SWD_WKEY | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

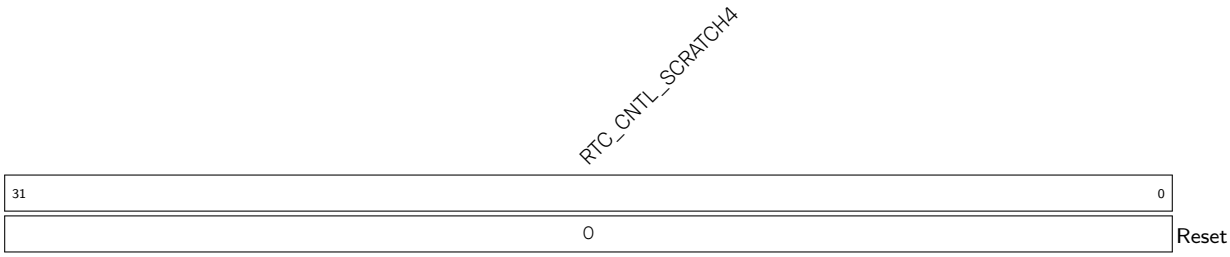
**RTC_CNTL_SWD_WKEY** Sets the write protection key of the super watchdog. (R/W)

### Register 9.40. RTC_CNTL_SW_CPU_STALL_REG (0x00B4)



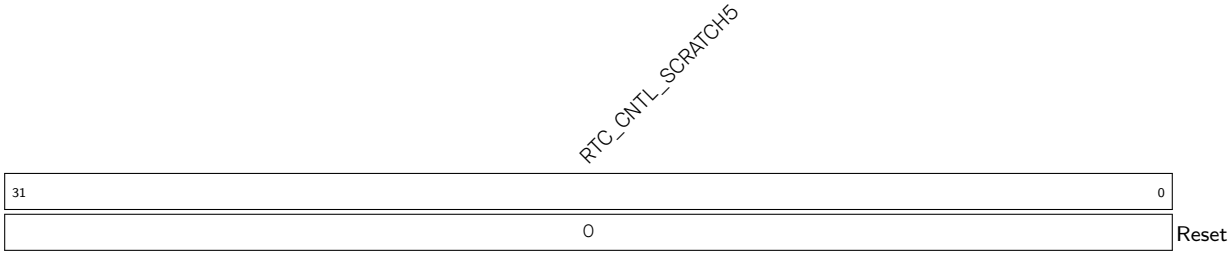

**RTC_CNTL_SW_STALL_PROCPU_C1** When RTC_CNTL_SW_STALL_PROCPU_C0 is configured to 0x2, setting this bit to 0x21 stalls the CPU by SW. (R/W)

### Register 9.41. RTC_CNTL_STORE4_REG (0x00B8)




**RTC_CNTL_SCRATCH4** Reservation register 4. (R/W)

### Register 9.42. RTC_CNTL_STORE5_REG (0x00BC)




**RTC_CNTL_SCRATCH5** Reservation register 5. (R/W)

**Register 9.43. RTC_CNTL_STORE6_REG (0x00C0)**

| RTC_CNTL_SCRATCH6 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**RTC_CNTL_SCRATCH6** Reservation register 6. (R/W)

**Register 9.44. RTC_CNTL_STORE7_REG (0x00C4)**

| RTC_CNTL_SCRATCH7 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**RTC_CNTL_SCRATCH7** Reservation register 7. (R/W)

Register 9.45. RTC_CNTL_LOW_POWER_ST_REG (0x00C8)

| (reserved) | RTC_CNTL_MAIN_STATE_IN_IDLE | (reserved) | RTC_CNTL_RDY_FOR_WAKEUP | (reserved) | |
|---|---|---|---|---|---|
| 31 28 | 27 | 26 20 | 19 | 18 0 | |
| 0 0 0 0 | 0 | 0 0 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 |

**RTC_CNTL_RDY_FOR_WAKEUP** Indicates the RTC is ready to be triggered by any wakeup source. (RO)

**RTC_CNTL_MAIN_STATE_IN_IDLE** Indicates the RTC state.

- 0: the chip can be either
  - in sleep modes.
  - entering sleep modes. In this case, wait until RTC_CNTL_RDY_FOR_WAKEUP bit is set, then you can wake up the chip.
  - exiting sleep mode. In this case, RTC_CNTL_MAIN_STATE_IN_IDLE will eventually become 1.
- 1: the chip is not in sleep modes (i.e. running normally).

(RO)

Register 9.46. RTC_CNTL_PAD_HOLD_REG (0x00D0)

| (reserved) | RTC_CNTL_GPIO_PIN5_HOLD | RTC_CNTL_GPIO_PIN4_HOLD | RTC_CNTL_GPIO_PIN3_HOLD | RTC_CNTL_GPIO_PIN2_HOLD | RTC_CNTL_GPIO_PIN1_HOLD | RTC_CNTL_GPIO_PIN0_HOLD | |
|---|---|---|---|---|---|---|---|
| 31 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 | Reset |

**RTC_CNTL_GPIO_PIN0_HOLD** Sets the GPIO 0 to the holding state. (R/W)

**RTC_CNTL_GPIO_PIN1_HOLD** Sets the GPIO 1 to the holding state. (R/W)

**RTC_CNTL_GPIO_PIN2_HOLD** Sets the GPIO 2 to the holding state. (R/W)

**RTC_CNTL_GPIO_PIN3_HOLD** Sets the GPIO 3 to the holding state. (R/W)

**RTC_CNTL_GPIO_PIN4_HOLD** Sets the GPIO 4 to the holding state. (R/W)

**RTC_CNTL_GPIO_PIN5_HOLD** Sets the GPIO 5 to the holding state. (R/W)

**Register 9.47. RTC_CNTL_DIG_PAD_HOLD_REG (0x00D4)**

| RTC_CNTL_DIG_PAD_HOLD | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**RTC_CNTL_DIG_PAD_HOLD** Set GPIO 6 to GPIO 21 to the holding state. (See bitmap to locate any GPIO). (R/W)

**Register 9.48. RTC_CNTL_BROWN_OUT_REG (0x00D8)**

| RTC_CNTL_BROWN_OUT_DET | RTC_CNTL_BROWN_OUT_ENA | RTC_CNTL_BROWN_OUT_CNT_CLR | RTC_CNTL_BROWN_OUT_ANA_RST_EN | RTC_CNTL_BROWN_OUT_RST_SEL | RTC_CNTL_BROWN_OUT_RST_ENA | RTC_CNTL_BROWN_OUT_RST_WAIT | RTC_CNTL_BROWN_OUT_PD_RF_ENA | RTC_CNTL_BROWN_OUT_CLOSE_FLASH_ENA | RTC_CNTL_BROWN_OUT_INT_WAIT | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 … 16 | 15 | 14 | 13 … 4 | 3 … 0 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0x3ff | 0 | 0 | 0x1 | 0 0 0 0 | Reset |

**RTC_CNTL_BROWN_OUT_INT_WAIT** Configures the waiting cycles before sending an interrupt. (R/W)

**RTC_CNTL_BROWN_OUT_CLOSE_FLASH_ENA** Set this bit to enable PD the flash when a brown-out happens. (R/W)

**RTC_CNTL_BROWN_OUT_PD_RF_ENA** Set this bit to enable PD the RF circuits when a brown-out happens. (R/W)

**RTC_CNTL_BROWN_OUT_RST_WAIT** Configures the waiting cycles before the reset after a brown-out. (R/W)

**RTC_CNTL_BROWN_OUT_RST_ENA** Enables to reset brown-out. (R/W)

**RTC_CNTL_BROWN_OUT_RST_SEL** Selects the reset type when a brown-out happens. 1: chip reset, 0: system reset. (R/W)

**RTC_CNTL_BROWN_OUT_ANA_RST_EN** Enables to reset brown-out. (R/W)

**RTC_CNTL_BROWN_OUT_CNT_CLR** Clears the brown-out counter. (WO)

**RTC_CNTL_BROWN_OUT_ENA** Set this bit to enable brown-out detection. (R/W)

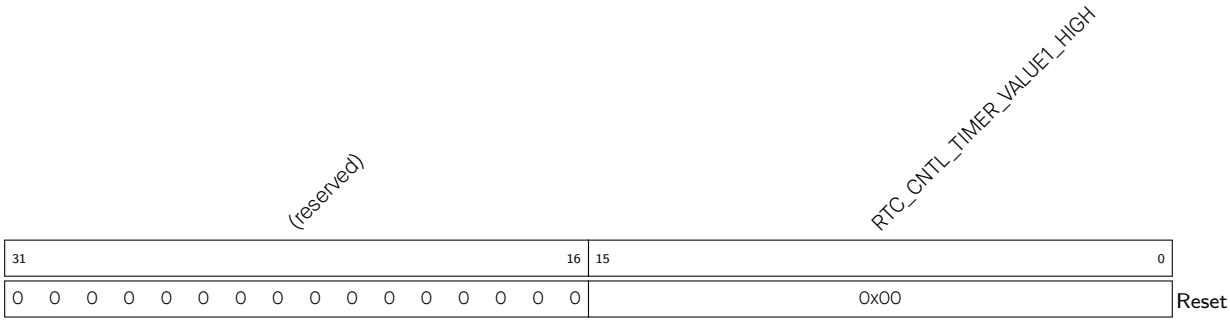
**RTC_CNTL_BROWN_OUT_DET** Indicates the status of the brown-out signal. (RO)

**Register 9.49. RTC_CNTL_TIME_LOW1_REG (0x00DC)**

| RTC_CNTL_TIMER_VALUE1_LOW | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

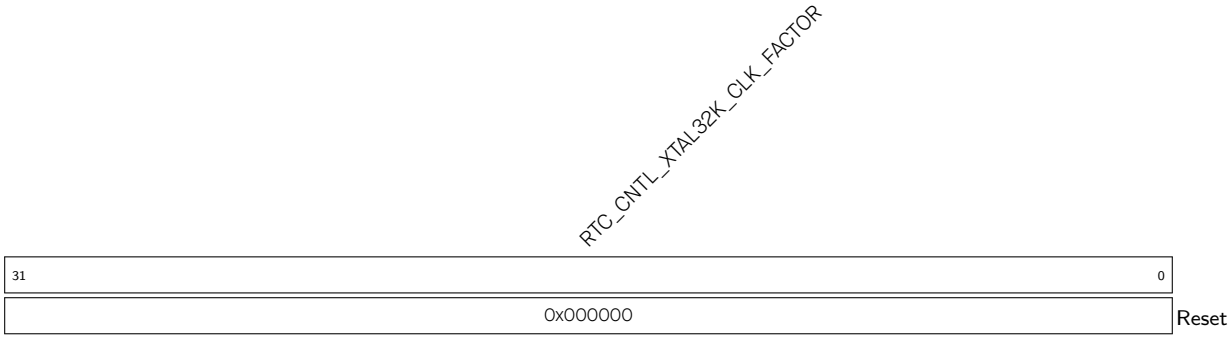
**RTC_CNTL_TIMER_VALUE1_LOW** Stores the lower 32 bits of RTC timer 1. (RO)

**Register 9.50. RTC_CNTL_TIME_HIGH1_REG (0x00E0)**




**RTC_CNTL_TIMER_VALUE1_HIGH** Stores the higher 16 bits of RTC timer. (RO)

**Register 9.51. RTC_CNTL_XTAL32K_CLK_FACTOR_REG (0x00E4)**




**RTC_CNTL_XTAL32K_CLK_FACTOR** Configures the divider factor for the XTAL32K oscillator. (R/W)

**Register 9.52. RTC_CNTL_XTAL32K_CONF_REG (0x00E8)**

| RTC_CNTL_XTAL32K_STABLE_THRES | RTC_CNTL_XTAL32K_WDT_TIMEOUT | RTC_CNTL_XTAL32K_RESTART_WAIT | RTC_CNTL_XTAL32K_RETURN_WAIT | |
|---|---|---|---|---|
| 31 28 | 27 20 | 19 4 | 3 0 | |
| 0x0 | 0xff | 0x00 | 0x0 | Reset |

**RTC_CNTL_XTAL32K_RETURN_WAIT** Defines the waiting cycles before returning to the normal XTAL32K oscillator. (R/W)

**RTC_CNTL_XTAL32K_RESTART_WAIT** Defines the waiting cycles before restarting the XTAL32K oscillator. (R/W)

**RTC_CNTL_XTAL32K_WDT_TIMEOUT** Defines the waiting period for clock detection. If no clock is detected after this period, the XTAL32K oscillator can be regarded as dead. (R/W)

**RTC_CNTL_XTAL32K_STABLE_THRES** Defines the allowed restarting period, within which the XTAL32K oscillator can be regarded as stable. (R/W)

**Register 9.53. RTC_CNTL_USB_CONF_REG (0x00EC)**

| (reserved) | RTC_CNTL_IO_MUX_RESET_DISABLE | (reserved) | |
|---|---|---|---|
| 31 19 | 18 | 17 0 | |
| 0 0 0 0 0 0 0 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 |

**RTC_CNTL_IO_MUX_RESET_DISABLE** Set this bit to disable io_mux reset. (R/W)

**Register 9.54. RTC_CNTL_SLP_REJECT_CAUSE_REG (0x00F0)**

| (reserved) | RTC_CNTL_REJECT_CAUSE | |
|---|---|---|
| 31 … 18 | 17 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**RTC_CNTL_REJECT_CAUSE** Stores the reject-to-sleep cause. (RO)

**Register 9.55. RTC_CNTL_OPTION1_REG (0x00F4)**

| (reserved) | RTC_CNTL_FORCE_DOWNLOAD_BOOT | |
|---|---|---|
| 31 … 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 | Reset |

**RTC_CNTL_FORCE_DOWNLOAD_BOOT** Set this bit to force the chip to boot from the download mode. (R/W)

**Register 9.56. RTC_CNTL_SLP_WAKEUP_CAUSE_REG (0x00F8)**

| (reserved) | RTC_CNTL_WAKEUP_CAUSE | |
|---|---|---|
| 31 … 17 | 16 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**RTC_CNTL_WAKEUP_CAUSE** Stores the wakeup cause. (RO)

**Register 9.57. RTC_CNTL_INT_ENA_RTC_W1TS_REG (0x0100)**

| 31–21 | 20 | 19 | 18–17 | 16 | 15 | 14–11 | 10 | 9 | 8–4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | RTC_CNTL_BBPLL_CAL_INT_ENA_W1TS | RTC_CNTL_GLITCH_DET_INT_ENA_W1TS | (reserved) | RTC_CNTL_XTAL32K_DEAD_INT_ENA_W1TS | RTC_CNTL_SWD_INT_ENA_W1TS | (reserved) | RTC_CNTL_MAIN_TIMER_INT_ENA_W1TS | RTC_CNTL_BROWN_OUT_INT_ENA_W1TS | (reserved) | RTC_CNTL_WDT_INT_ENA_W1TS | (reserved) | RTC_CNTL_SLP_REJECT_INT_ENA_W1TS | RTC_CNTL_SLP_WAKEUP_INT_ENA_W1TS | |
| 0 0 0 0 0 0 0 0 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 |

**RTC_CNTL_SLP_WAKEUP_INT_ENA_W1TS** Enables interrupts when the chip wakes up from sleep by writing 1 to set (W1TS). (WO)

**RTC_CNTL_SLP_REJECT_INT_ENA_W1TS** Enables interrupts when the chip rejects to go to sleep by writing 1 to set (W1TS). (WO)

**RTC_CNTL_WDT_INT_ENA_W1TS** Enables the RTC watchdog interrupt by writing 1 to set (W1TS). (WO)

**RTC_CNTL_BROWN_OUT_INT_ENA_W1TS** Enables the brownout interrupt by writing 1 to set. (WO)

**RTC_CNTL_MAIN_TIMER_INT_ENA_W1TS** Enables the RTC main timer interrupt by writing 1 to set (W1TS). (WO)

**RTC_CNTL_SWD_INT_ENA_W1TS** Enables the super watchdog interrupt by writing 1 to set (W1TS). (WO)

**RTC_CNTL_XTAL32K_DEAD_INT_ENA_W1TS** Enables interrupts when the XTAL32K is dead by writing 1 to set (W1TS). (WO)

**RTC_CNTL_GLITCH_DET_INT_ENA_W1TS** Enables interrupts when a glitch is detected by writing 1 to set (W1TS). (WO)

**RTC_CNTL_BBPLL_CAL_INT_ENA_W1TS** Enables interrupts upon the ending of a bb_pll call by writing 1 to set (W1TS). (WO)

**Register 9.58. RTC_CNTL_INT_ENA_RTC_W1TC_REG (0x0104)**

| 31–21 | 20 | 19 | 18–17 | 16 | 15 | 14–11 | 10 | 9 | 8–4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | RTC_CNTL_BBPLL_CAL_INT_ENA_W1TC | RTC_CNTL_GLITCH_DET_INT_ENA_W1TC | (reserved) | RTC_CNTL_XTAL32K_DEAD_INT_ENA_W1TC | RTC_CNTL_SWD_INT_ENA_W1TC | (reserved) | RTC_CNTL_MAIN_TIMER_INT_ENA_W1TC | RTC_CNTL_BROWN_OUT_INT_ENA_W1TC | (reserved) | RTC_CNTL_WDT_INT_ENA_W1TC | (reserved) | RTC_CNTL_SLP_REJECT_INT_ENA_W1TC | RTC_CNTL_SLP_WAKEUP_INT_ENA_W1TC | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**RTC_CNTL_SLP_WAKEUP_INT_ENA_W1TC** Clear the interrupt enable bit when the chip wakes up from sleep by writing 1 to clear. (W1TC). (WO)

**RTC_CNTL_SLP_REJECT_INT_ENA_W1TC** Clear the interrupt enable bit when the chip rejects to go to sleep by writing 1 to clear (W1TC). (WO)

**RTC_CNTL_WDT_INT_ENA_W1TC** Clear the RTC watchdog interrupt enable bit by writing 1 to clear (W1TC). (WO)

**RTC_CNTL_BROWN_OUT_INT_ENA_W1TC** Clear the brownout interrupt enable bit by writing 1 to clear (W1TC). (WO)

**RTC_CNTL_MAIN_TIMER_INT_ENA_W1TC** Clear the RTC timer interrupt enable bit by writing 1 to clear. (W1TC). (WO)

**RTC_CNTL_SWD_INT_ENA_W1TC** Clear the super watchdog interrupt enable bit by writing 1 to clear (W1TC). (WO)

**RTC_CNTL_XTAL32K_DEAD_INT_ENA_W1TC** Clear the interrupt enable bit when the XTAL32K is dead by writing 1 to clear (W1TC). (WO)

**RTC_CNTL_GLITCH_DET_INT_ENA_W1TC** Clear the interrupt enable bit when a glitch is detected by writing 1 to clear (W1TC). (WO)

**RTC_CNTL_BBPLL_CAL_INT_ENA_W1TC** Clear the interrupt enable bit upon the ending of a bb_pll call by writing 1 to clear (W1TC).(WO)

**Register 9.59. RTC_CNTL_RETENTION_CTRL_REG (0x0108)**

| (reserved) | RTC_CNTL_RETENTION_EN | (reserved) | (reserved) | RTC_CNTL_RETENTION_CLK_SEL | (reserved) | |
|---|---|---|---|---|---|---|
| 31 27 | 26 | 25 22 | 21 19 | 18 | 17 0 | |
| 20 | 0 | 3 | 2 | 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | Reset |

**RTC_CNTL_RETENTION_CLK_SEL** Selects the retention clock. 0: RC_FAST_CLK; 1: XTAL_CLK. (R/W)

**RTC_CNTL_RETENTION_EN** Set to enable the CPU retention in light sleep. (R/W)

**Register 9.60. RTC_CNTL_GPIO_WAKEUP_REG (0x0110)**

| RTC_CNTL_GPIO_PIN0_WAKEUP_ENABLE | RTC_CNTL_GPIO_PIN1_WAKEUP_ENABLE | RTC_CNTL_GPIO_PIN2_WAKEUP_ENABLE | RTC_CNTL_GPIO_PIN3_WAKEUP_ENABLE | RTC_CNTL_GPIO_PIN4_WAKEUP_ENABLE | RTC_CNTL_GPIO_PIN5_WAKEUP_ENABLE | RTC_CNTL_GPIO_PIN0_INT_TYPE | RTC_CNTL_GPIO_PIN1_INT_TYPE | RTC_CNTL_GPIO_PIN2_INT_TYPE | RTC_CNTL_GPIO_PIN3_INT_TYPE | RTC_CNTL_GPIO_PIN4_INT_TYPE | RTC_CNTL_GPIO_PIN5_INT_TYPE | RTC_CNTL_GPIO_PIN_CLK_GATE | RTC_CNTL_GPIO_WAKEUP_STATUS_CLR | RTC_CNTL_GPIO_WAKEUP_STATUS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 23 | 22 20 | 19 17 | 16 14 | 13 11 | 10 8 | 7 | 6 | 5 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**RTC_CNTL_GPIO_WAKEUP_STATUS** Indicates the RTC GPIO that woke up the chip, with Bit0 to Bit5 representing RTC GPIO 0 to RTC GPIO 5, respectively. For example, 010000 indicates it is the RTC GPIO 4 that woke up the chip. (RO)

**RTC_CNTL_GPIO_WAKEUP_STATUS_CLR** Clears the RTC GPIO wakeup flag. (R/W)

**RTC_CNTL_GPIO_PIN_CLK_GATE** Enables the RTC GPIO clock gate. (R/W)

**RTC_CNTL_GPIO_PIN5_INT_TYPE** Configures RTC GPIO 5 wakeup type.

0: disable wakeup by RTC GPIO
1: wake up the chip upon the rising edge
2: wake up the chip upon the failing edge
3: wake up the chip upon the rising edge or the failing edge
4: wake up the chip upon low level
5: wake up the chip upon high level
(R/W)

**RTC_CNTL_GPIO_PIN4_INT_TYPE** Configures RTC GPIO 4 wakeup type.

0: disable wakeup by RTC GPIO
1: wake up the chip upon the rising edge
2: wake up the chip upon the failing edge
3: wake up the chip upon the rising edge or the failing edge
4: wake up the chip upon low level
5: wake up the chip upon high level
(R/W)

**RTC_CNTL_GPIO_PIN3_INT_TYPE** Configures RTC GPIO 3 wakeup type.

0: disable wakeup by RTC GPIO
1: wake up the chip upon the rising edge
2: wake up the chip upon the failing edge
3: wake up the chip upon the rising edge or the failing edge
4: wake up the chip upon low level
5: wake up the chip upon high level
(R/W)

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

Register 9.60. RTC_CNTL_GPIO_WAKEUP_REG (0x0110)

Continued from the previous page...

**RTC_CNTL_GPIO_PIN2_INT_TYPE** Configures RTC GPIO 2 wakeup type.

0: disable wakeup by RTC GPIO
1: wake up the chip upon the rising edge
2: wake up the chip upon the failing edge
3: wake up the chip upon the rising edge or the failing edge
4: wake up the chip upon low level
5: wake up the chip upon high level
(R/W)

**RTC_CNTL_GPIO_PIN1_INT_TYPE** Configures RTC GPIO 1 wakeup type.

0: disable wakeup by RTC GPIO
1: wake up the chip upon the rising edge
2: wake up the chip upon the failing edge
3: wake up the chip upon the rising edge or the failing edge
4: wake up the chip upon low level
5: wake up the chip upon high level
(R/W)

**RTC_CNTL_GPIO_PIN0_INT_TYPE** Configures RTC GPIO 0 wakeup type.

0: disable wakeup by RTC GPIO
1: wake up the chip upon the rising edge
2: wake up the chip upon the failing edge
3: wake up the chip upon the rising edge or the failing edge
4: wake up the chip upon low level
5: wake up the chip upon high level
(R/W)

**RTC_CNTL_GPIO_PIN5_WAKEUP_ENABLE** Enables wakeup from RTC GPIO 5. (R/W)

**RTC_CNTL_GPIO_PIN4_WAKEUP_ENABLE** Enables wakeup from RTC GPIO 4. (R/W)

**RTC_CNTL_GPIO_PIN3_WAKEUP_ENABLE** Enables wakeup from RTC GPIO 3. (R/W)

**RTC_CNTL_GPIO_PIN2_WAKEUP_ENABLE** Enables wakeup from RTC GPIO 2. (R/W)

**RTC_CNTL_GPIO_PIN1_WAKEUP_ENABLE** Enables wakeup from RTC GPIO 1. (R/W)

**RTC_CNTL_GPIO_PIN0_WAKEUP_ENABLE** Enables wakeup from RTC GPIO 0. (R/W)

**Register 9.61. RTC_CNTL_SENSOR_CTRL_REG (0x011C)**

| RTC_CNTL_FORCE_XPD_SAR | (reserved) | (reserved) | |
|---|---|---|---|
| 31 30 | 29 27 | 26 0 | |
| 0 | 0 | 0 0 0 0 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 |

**RTC_CNTL_FORCE_XPD_SAR** Set this field to FPU SAR ADC. (R/W)

# Chapter 10

# System Timer (SYSTIMER)

## 10.1 Overview

ESP32-C3 provides a 52-bit timer, which can be used to generate tick interrupts for operating system, or be used as a general timer to generate periodic interrupts or one-time interrupts.

The timer consists of two counters UNIT0 and UNIT1. The count values can be monitored by three comparators COMP0, COMP1 and COMP2. See the timer block diagram on Figure 10.1-1.

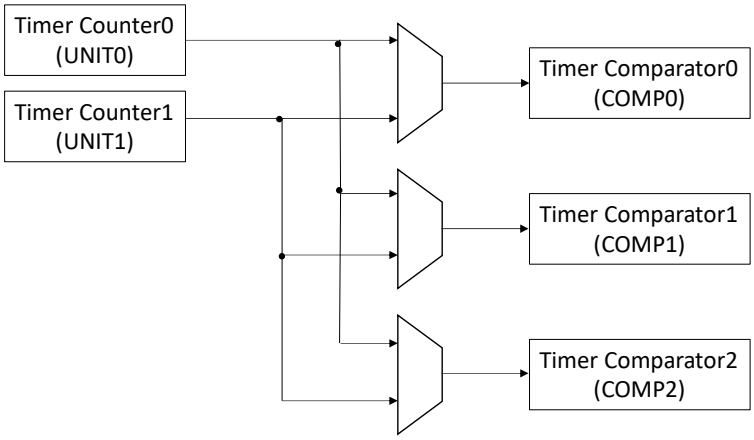



Figure 10.1-1. System Timer Structure

## 10.2 Features

- Consist of two 52-bit counters and three 52-bit comparators
- Software accessing registers is clocked by APB_CLK
- Use CNT_CLK for counting, with an average frequency of 16 MHz in two counting cycles
- Use 40 MHz XTAL_CLK as the clock source of CNT_CLK
- Support for 52-bit alarm values (t) and 26-bit alarm periods ($\delta$t)
- Provide two modes to generate alarms:
  - Target mode: only a one-time alarm is generated based on the alarm value (t)
  - Period mode: periodic alarms are generated based on the alarm period ($\delta$t)
- Three comparators can generate three independent interrupts based on configured alarm value (t) or alarm period ($\delta$t)

- Software configuring the reference count value. For example, the system timer is able to load back the sleep time recorded by the RTC timer via software after Light-sleep
- Can be configured to stall or continue running when CPU stalls or enters on-chip-debugging mode

## 10.3 Clock Source Selection

The counters and comparators are driven using XTAL_CLK. After scaled by a fractional divider, a $f_{XTAL_CLK}/3$ clock is generated in one count cycle and a $f_{XTAL_CLK}/2$ clock in another count cycle. The average clock frequency is $f_{XTAL_CLK}/2.5$, which is 16 MHz, i.e. the CNT_CLK in Figure 10.4-1. The timer counting is incremented by 1/16 $\mu s$ on each CNT_CLK cycle.

Software operation such as configuring registers is clocked by APB_CLK. For more information about APB_CLK, see Chapter 6 *Reset and Clock*.

The following two bits of system registers are also used to control the system timer:

- SYSTEM_SYSTIMER_CLK_EN in register SYSTEM_PERIP_CLK_EN0_REG: enable APB_CLK signal to system timer.
- SYSTEM_SYSTIMER_RST in register SYSTEM_PERIP_RST_EN0_REG: reset system timer.

Note that if the timer is reset, its registers will be restored to their default values. For more information, please refer to Table Peripheral Clock Gating and Reset in Chapter 16 *System Registers (SYSREG)*.

## 10.4 Functional Description

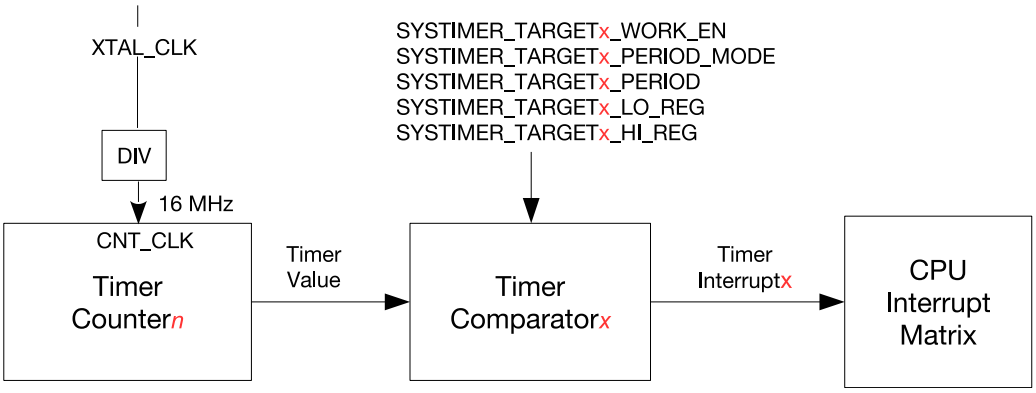



Figure 10.4-1. System Timer Alarm Generation

Figure 10.4-1 shows the procedure to generate alarm in system timer. In this process, one timer counter and one timer comparator are used. An alarm interrupt will be generated accordingly based on the comparison result in comparator.

### 10.4.1 Counter

The system timer has two 52-bit timer counters, shown as UNIT*n* (*n* = 0 or 1). Their counting clock source is a 16 MHz clock, i.e. CNT_CLK. Whether UNIT*n* works or not is controlled by two bits in register SYSTIMER_CONF_REG:

- SYSTIMER_TIMER_UNIT*n*_WORK_EN: set this bit to enable the counter UNIT*n* in system timer.
- SYSTIMER_TIMER_UNIT*n*_COREO_STALL_EN: if this bit is set, the counter UNIT*n* stops when CPU is stalled. The counter continues its counting after the CPU resumes.

The configuration of the two bits to control the counter UNIT*n* is shown below, assuming that CPU is stalled.

**Table 10.4-1. UNIT*n* Configuration Bits**

| SYSTIMER_TIMER_ UNIT*n*_WORK_EN | SYSTIMER_TIMER_ UNIT*n*_COREO_STALL_EN | Counter UNIT*n* |
|---|---|---|
| 0 | x * | Not at work |
| 1 | 1 | Stop counting, but will continue its counting after the CPU resumes. |
| 1 | 0 | Keep counting |

* x: Don't-care.

When the counter UNIT*n* is at work, the count value is incremented on each counting cycle. When the counter UNIT*n* is stopped, the count value stops increasing and keeps unchanged.

The lower 32 and higher 20 bits of initial count value are loaded from the registers SYSTIMER_TIMER_UNIT*n*_LOAD _LO and SYSTIMER_TIMER_UNIT*n*_LOAD_HI. Writing 1 to the bit SYSTIMER_TIMER_UNIT*n*_LOAD will trigger a reload event, and the current count value will be changed immediately. If UNIT*n* is at work, the counter will continue to count up from the new reloaded value.

Writing 1 to SYSTIMER_TIMER_UNIT*n*_UPDATE will trigger an update event. The lower 32 and higher 20 bits of current count value will be locked into the registers SYSTIMER_TIMER_UNIT*n*_VALUE_LO and SYSTIMER_TIMER_ UNIT*n*_VALUE_HI, and then SYSTIMER_TIMER_UNIT*n*_VALUE_VALID is asserted. Before the next update event, the values of SYSTIMER_TIMER_UNIT*n*_VALUE_LO and SYSTIMER_TIMER_UNIT*n*_VALUE_HI remain unchanged.

### 10.4.2 Comparator and Alarm

The system timer has three 52-bit comparators, shown as COMP*x* (*x* = 0, 1, or 2). The comparators can generate independent interrupts based on different alarm values (t) or alarm periods ($\delta$t).

Configure SYSTIMER_TARGET*x*_PERIOD_MODE to choose from the two alarm modes for each COMP*x*:

- 1: select period mode
- 0: select target mode

In period mode, the alarm period ($\delta$t) is provided by the register SYSTIMER_TARGET*x*_PERIOD. Assuming that current count value is t1, when it reaches (t1 + $\delta$t), an alarm interrupt will be generated. Another alarm interrupt also will be generated when the count value reaches (t1 + 2*$\delta$t). By such way, periodic alarms are generated.

In target mode, the lower 32 bits and higher 20 bits of the alarm value (t) are provided by SYSTIMER_TIMER_TARGET

x_LO and SYSTIMER_TIMER_TARGETx_HI. Assuming that current count value is t2 (t2 <= t), an alarm interrupt will be generated when the count value reaches the alarm value (t). Unlike in period mode, only one alarm interrupt is generated in target mode.

SYSTIMER_TARGETx_TIMER_UNIT_SEL is used to choose the count value from which timer counter to be compared for alarm:

- 1: use the count value from UNIT1
- 0: use the count value from UNIT0

Finally, set SYSTIMER_TARGETx_WORK_EN and COMPx starts to compare the count value with the alarm value (t) in target mode or with the alarm period (t1 + n*$\delta$t) in period mode.

An alarm is generated when the count value equals to the alarm value (t) in target mode or to the start value + n*alarm period $\delta$t (n = 1,2,3...) in period mode. But if the alarm value (t) set in registers is less than current count value, i.e. the target has already passed, or current count value is larger than the target value (t) within a range (0 ~ $2^{51}$ -1), an alarm interrupt also is generated immediately. The relationship between current count value $t_c$, the alarm value $t_t$ and alarm trigger point is shown below.

Table 10.4-2. Trigger Point

| Relationship Between $t_c$ and $t_t$ | Trigger Point |
|---|---|
| $t_c$- $t_t$ <= 0 | $t_c$ = $t_t$, an alarm is triggered. |
| 0 <= $t_c$ - $t_t$ < $2^{51}$ - 1<br>( $t_c$ < $2^{51}$ and $t_t$ < $2^{51}$,<br>or $t_c$ >= $2^{51}$ and $t_t$ >= $2^{51}$) | An alarm is triggered immediately. |
| $t_c$ - $t_t$ >= $2^{51}$ - 1 | $t_c$ overflows after counting to its maximum value 52'hfffffffffffff, and then starts counting up from 0. When its value reaches $t_t$, an alarm is triggered. |

## 10.4.3 Synchronization Operation

The clock (APB_CLK) used in software operation is not the same one as the timer counters and comparators working on CNT_CLK. Synchronization is needed for some configuration registers. A complete synchronization action takes two steps:

1. Software writes suitable values to configuration fields, see the first column in Table 10.4-3.
2. Software writes 1 to corresponding bits to start synchronization, see the second column in Table 10.4-3.

Table 10.4-3. Synchronization Operation

| Configuration Fields | Synchronization Enable Bit |
|---|---|
| SYSTIMER_TIMER_UNITn_LOAD_LO<br>SYSTIMER_TIMER_UNITn_LOAD_HI | SYSTIMER_TIMER_UNITn_LOAD |
| SYSTIMER_TARGETx_PERIOD<br>SYSTIMER_TIMER_TARGETx_HI<br>SYSTIMER_TIMER_TARGETx_LO | SYSTIMER_TIMER_COMPx_LOAD |

### 10.4.4 Interrupt

Each comparator has one level-type alarm interrupt, named as SYSTIMER_TARGETx_INT. Interrupts signal is asserted high when the comparator starts to alarm. Until the interrupt is cleared by software, it remains high. To enable interrupts, set the bit SYSTIMER_TARGETx_INT_ENA.

## 10.5 Programming Procedure

When configuring COMPx and UNITn, please ensure the corresponding COMP and UNIT are at work.

### 10.5.1 Read Current Count Value

1. Set SYSTIMER_TIMER_UNITn_UPDATE to update the current count value into SYSTIMER_TIMER_UNITn_VALUE_HI and SYSTIMER_TIMER_UNITn_VALUE_LO.
2. Poll the reading of SYSTIMER_TIMER_UNITn_VALUE_VALID, till it's 1, which means user now can read the count value from SYSTIMER_TIMER_UNITn_VALUE_HI and SYSTIMER_TIMER_UNITn_VALUE_LO.
3. Read the lower 32 bits and higher 20 bits from SYSTIMER_TIMER_UNITn_VALUE_LO and SYSTIMER_TIMER_UNITn_VALUE_HI.

### 10.5.2 Configure One-Time Alarm in Target Mode

1. Set SYSTIMER_TARGETx_TIMER_UNIT_SEL to select the counter (UNIT0 or UNIT1) used for COMPx.
2. Read current count value, see Section 10.5.1. This value will be used to calculate the alarm value (t) in Step 4.
3. Clear SYSTIMER_TARGETx_PERIOD_MODE to enable target mode.
4. Set an alarm value (t), and fill its lower 32 bits to SYSTIMER_TIMER_TARGETx_LO, and the higher 20 bits to SYSTIMER_TIMER_TARGETx_HI.
5. Set SYSTIMER_TIMER_COMPx_LOAD to synchronize the alarm value (t) to COMPx, i.e. load the alarm value (t) to the COMPx.
6. Set SYSTIMER_TARGETx_WORK_EN to enable the selected COMPx. COMPx starts comparing the count value with the alarm value (t).
7. Set SYSTIMER_TARGETx_INT_ENA to enable timer interrupt. When Unitn counts to the alarm value (t), a SYSTIMER_TARGETx_INT interrupt is triggered.

### 10.5.3 Configure Periodic Alarms in Period Mode

1. Set SYSTIMER_TARGETx_TIMER_UNIT_SEL to select the counter (UNIT0 or UNIT1) used for COMPx.
2. Set an alarm period ($\delta$t), and fill it to SYSTIMER_TARGETx_PERIOD.
3. Set SYSTIMER_TIMER_COMPx_LOAD to synchronize the alarm period ($\delta$t) to COMPx, i.e. load the alarm period ($\delta$t) to COMPx.
4. Clear and then set SYSTIMER_TARGETx_PERIOD_MODE to configure COMPx into period mode.

5. Set SYSTIMER_TARGETx_WORK_EN to enable the selected COMPx. COMPx starts comparing the count value with the sum of start value + n*δt (n = 1, 2, 3...).
6. Set SYSTIMER_TARGETx_INT_ENA to enable timer interrupt. A SYSTIMER_TARGETx_INT interrupt is triggered when Unitn counts to start value + n*δt (n = 1, 2, 3...) set in step 2.

### 10.5.4 Update After Light-sleep

1. Configure the RTC timer before the chip goes to Light-sleep, to record the exact sleep time. For more information, see Chapter 9 *Low-power Management*.
2. Read the sleep time from the RTC timer when the chip is woken up from Light-sleep.
3. Read current count value of system timer, see Section 10.5.1.
4. Convert the time value recorded by the RTC timer from the clock cycles based on RTC_SLOW_CLK to that based on 16 MHz CNT_CLK. For example, if the frequency of RTC_SLOW_CLK is 32 KHz, the recorded RTC timer value should be converted by multiplying by 500.
5. Add the converted RTC value to the current count value of the system timer:
   - Fill the new value into SYSTIMER_TIMER_UNITn_LOAD_LO (low 32 bits) and SYSTIMER_TIMER_UNITn_LOAD_HI (high 20 bits).
   - Set SYSTIMER_TIMER_UNITn_LOAD to load new timer value into system timer. By such way, the system timer is updated.

## 10.6 Register Summary

The addresses in this section are relative to system timer base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Clock Control Register** | | | |
| SYSTIMER_CONF_REG | Configure system timer clock | 0x0000 | R/W |
| **UNIT0 Control and Configuration Registers** | | | |
| SYSTIMER_UNIT0_OP_REG | Read UNIT0 value to registers | 0x0004 | varies |
| SYSTIMER_UNIT0_LOAD_HI_REG | High 20 bits to be loaded to UNIT0 | 0x000C | R/W |
| SYSTIMER_UNIT0_LOAD_LO_REG | Low 32 bits to be loaded to UNIT0 | 0x0010 | R/W |
| SYSTIMER_UNIT0_VALUE_HI_REG | UNIT0 value, high 20 bits | 0x0040 | RO |
| SYSTIMER_UNIT0_VALUE_LO_REG | UNIT0 value, low 32 bits | 0x0044 | RO |
| SYSTIMER_UNIT0_LOAD_REG | UNIT0 synchronization register | 0x005C | WT |
| **UNIT1 Control and Configuration Registers** | | | |
| SYSTIMER_UNIT1_OP_REG | Read UNIT1 value to registers | 0x0008 | varies |
| SYSTIMER_UNIT1_LOAD_HI_REG | High 20 bits to be loaded to UNIT1 | 0x0014 | R/W |
| SYSTIMER_UNIT1_LOAD_LO_REG | Low 32 bits to be loaded to UNIT1 | 0x0018 | R/W |
| SYSTIMER_UNIT1_VALUE_HI_REG | UNIT1 value, high 20 bits | 0x0048 | RO |
| SYSTIMER_UNIT1_VALUE_LO_REG | UNIT1 value, low 32 bits | 0x004C | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| SYSTIMER_UNIT1_LOAD_REG | UNIT1 synchronization register | 0x0060 | WT |
| **Comparator0 Control and Configuration Registers** | | | |
| SYSTIMER_TARGET0_HI_REG | Alarm value to be loaded to COMP0, high 20 bits | 0x001C | R/W |
| SYSTIMER_TARGET0_LO_REG | Alarm value to be loaded to COMP0, low 32 bits | 0x0020 | R/W |
| SYSTIMER_TARGET0_CONF_REG | Configure COMP0 alarm mode | 0x0034 | R/W |
| SYSTIMER_COMP0_LOAD_REG | COMP0 synchronization register | 0x0050 | WT |
| **Comparator1 Control and Configuration Registers** | | | |
| SYSTIMER_TARGET1_HI_REG | Alarm value to be loaded to COMP1, high 20 bits | 0x0024 | R/W |
| SYSTIMER_TARGET1_LO_REG | Alarm value to be loaded to COMP1, low 32 bits | 0x0028 | R/W |
| SYSTIMER_TARGET1_CONF_REG | Configure COMP1 alarm mode | 0x0038 | R/W |
| SYSTIMER_COMP1_LOAD_REG | COMP1 synchronization register | 0x0054 | WT |
| **Comparator2 Control and Configuration Registers** | | | |
| SYSTIMER_TARGET2_HI_REG | Alarm value to be loaded to COMP2, high 20 bits | 0x002C | R/W |
| SYSTIMER_TARGET2_LO_REG | Alarm value to be loaded to COMP2, low 32 bits | 0x0030 | R/W |
| SYSTIMER_TARGET2_CONF_REG | Configure COMP2 alarm mode | 0x003C | R/W |
| SYSTIMER_COMP2_LOAD_REG | COMP2 synchronization register | 0x0058 | WT |
| **Interrupt Registers** | | | |
| SYSTIMER_INT_ENA_REG | Interrupt enable register of system timer | 0x0064 | R/W |
| SYSTIMER_INT_RAW_REG | Interrupt raw register of system timer | 0x0068 | R/WTC/SS |
| SYSTIMER_INT_CLR_REG | Interrupt clear register of system timer | 0x006C | WT |
| SYSTIMER_INT_ST_REG | Interrupt status register of system timer | 0x0070 | RO |
| **Version Register** | | | |
| SYSTIMER_DATE_REG | Version control register | 0x00FC | R/W |

## 10.7 Registers

The addresses in this section are relative to system timer base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 10.1. SYSTIMER_CONF_REG (0x0000)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | | | | | | | | | | | | | | | | | | | | | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SYSTIMER_CLK_EN | SYSTIMER_TIMER_UNIT0_WORK_EN | SYSTIMER_TIMER_UNIT1_WORK_EN | SYSTIMER_TIMER_UNIT0_COREO_STALL_EN | (reserved) | SYSTIMER_TIMER_UNIT1_COREO_STALL_EN | (reserved) | SYSTIMER_TARGET0_WORK_EN | SYSTIMER_TARGET1_WORK_EN | SYSTIMER_TARGET2_WORK_EN | (reserved) | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 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 | 0 | 0 | 0 | 0 | Reset |

**SYSTIMER_TARGET2_WORK_EN** COMP2 work enable bit. (R/W)

**SYSTIMER_TARGET1_WORK_EN** COMP1 work enable bit. (R/W)

**SYSTIMER_TARGET0_WORK_EN** COMP0 work enable bit. (R/W)

**SYSTIMER_TIMER_UNIT1_COREO_STALL_EN** UNIT1 is stalled when CPU stalled. (R/W)

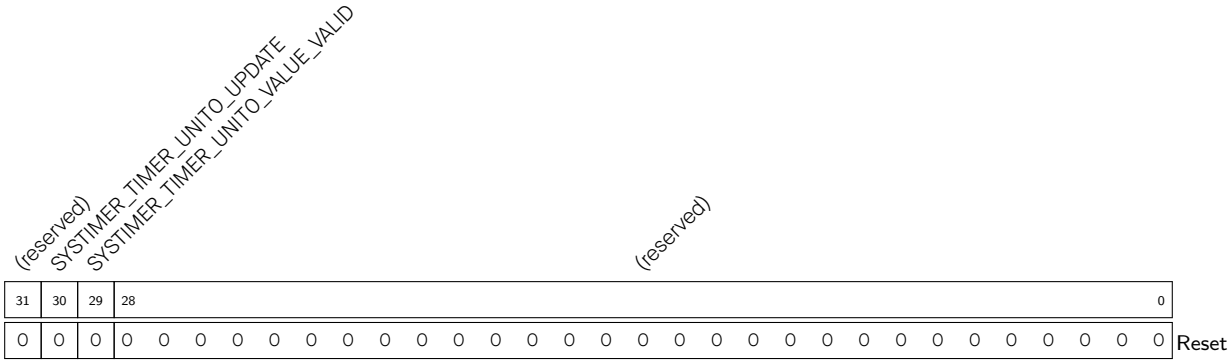
**SYSTIMER_TIMER_UNIT0_COREO_STALL_EN** UNIT0 is stalled when CPU stalled. (R/W)

**SYSTIMER_TIMER_UNIT1_WORK_EN** UNIT1 work enable bit. (R/W)

**SYSTIMER_TIMER_UNIT0_WORK_EN** UNIT0 work enable bit. (R/W)

**SYSTIMER_CLK_EN** Register clock gating. 1: Register clock is always enabled for read and write operations. 0: Only enable needed clock for register read or write operations. (R/W)

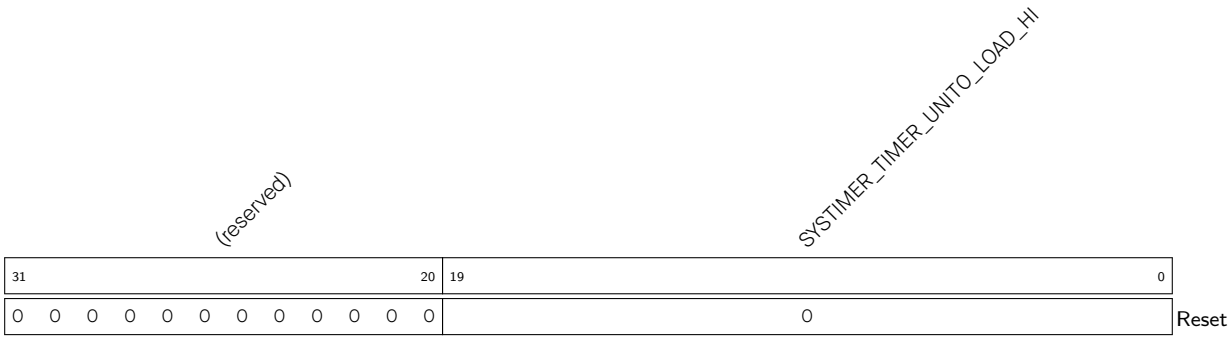
Register 10.2. SYSTIMER_UNIT0_OP_REG (0x0004)




**SYSTIMER_TIMER_UNIT0_VALUE_VALID** Timer value is synchronized and valid. (R/SS/WTC)

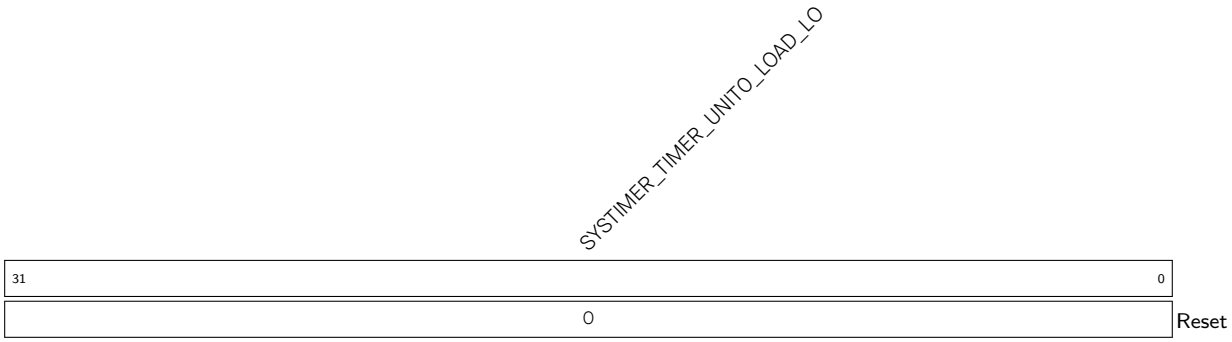
**SYSTIMER_TIMER_UNIT0_UPDATE** Update timer UNIT0, i.e. read the UNIT0 count value to SYSTIMER_TIMER_UNIT0_VALUE_HI and SYSTIMER_TIMER_UNIT0_VALUE_LO. (WT)

Register 10.3. SYSTIMER_UNIT0_LOAD_HI_REG (0x000C)




**SYSTIMER_TIMER_UNIT0_LOAD_HI** The value to be loaded to UNIT0, high 20 bits. (R/W)

Register 10.4. SYSTIMER_UNIT0_LOAD_LO_REG (0x0010)




**SYSTIMER_TIMER_UNIT0_LOAD_LO** The value to be loaded to UNIT0, low 32 bits. (R/W)

Register 10.5. SYSTIMER_UNIT0_VALUE_HI_REG (0x0040)

| (reserved) | SYSTIMER_TIMER_UNIT0_VALUE_HI | |
|---|---|---|
| 31 … 20 | 19 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**SYSTIMER_TIMER_UNIT0_VALUE_HI** UNIT0 read value, high 20 bits. (RO)

Register 10.6. SYSTIMER_UNIT0_VALUE_LO_REG (0x0044)

| SYSTIMER_TIMER_UNIT0_VALUE_LO | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

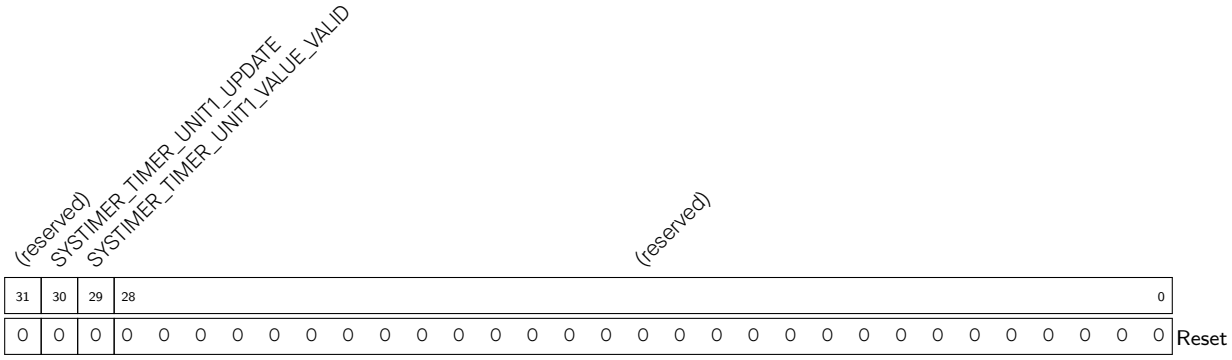
**SYSTIMER_TIMER_UNIT0_VALUE_LO** UNIT0 read value, low 32 bits. (RO)

Register 10.7. SYSTIMER_UNIT0_LOAD_REG (0x005C)

| (reserved) | SYSTIMER_TIMER_UNIT0_LOAD | |
|---|---|---|
| 31 … 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 | Reset |

**SYSTIMER_TIMER_UNIT0_LOAD** UNIT0 synchronization enable signal. Set this bit to reload the values of SYSTIMER_TIMER_UNIT0_LOAD_HI and SYSTIMER_TIMER_UNIT0_LOAD_LO to UNIT0. (WT)

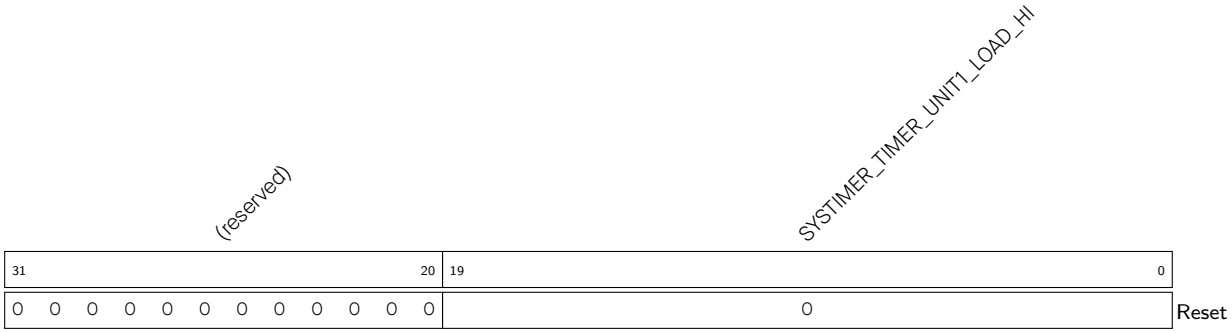
**Register 10.8. SYSTIMER_UNIT1_OP_REG (0x0008)**




**SYSTIMER_TIMER_UNIT1_VALUE_VALID** UNIT1 value is synchronized and valid. (R/SS/WTC)

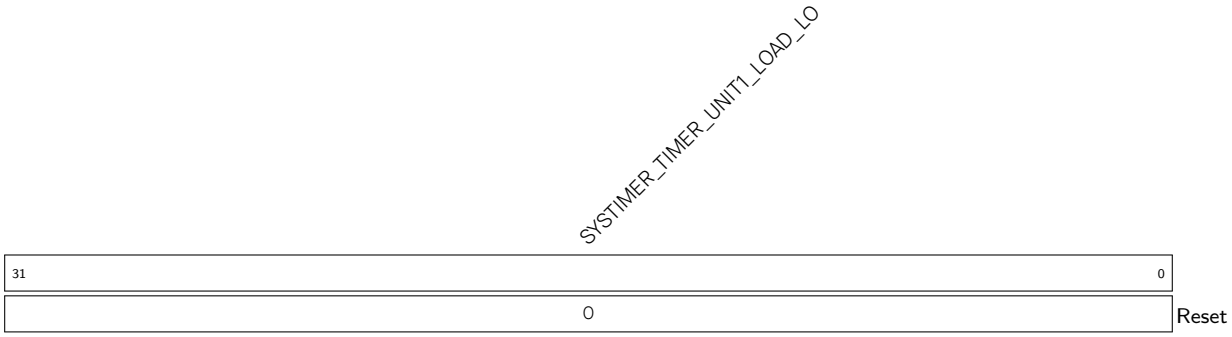
**SYSTIMER_TIMER_UNIT1_UPDATE** Update timer UNIT1, i.e. read the UNIT1 count value to SYSTIMER_TIMER_UNIT1_VALUE_HI and SYSTIMER_TIMER_UNIT1_VALUE_LO. (WT)

**Register 10.9. SYSTIMER_UNIT1_LOAD_HI_REG (0x0014)**



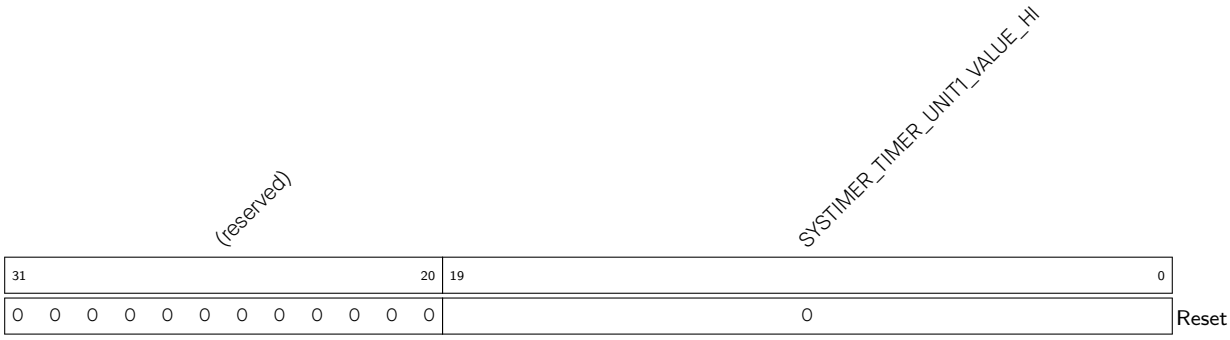

**SYSTIMER_TIMER_UNIT1_LOAD_HI** The value to be loaded to UNIT1, high 20 bits. (R/W)

**Register 10.10. SYSTIMER_UNIT1_LOAD_LO_REG (0x0018)**



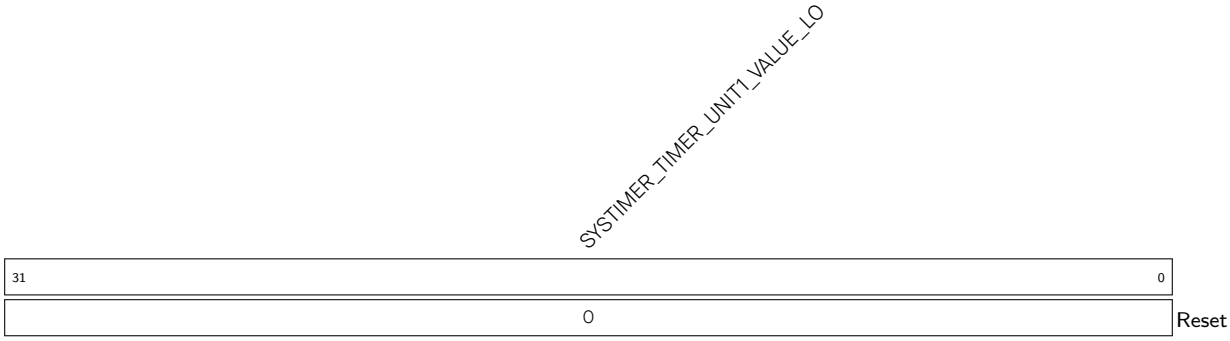

**SYSTIMER_TIMER_UNIT1_LOAD_LO** The value to be loaded to UNIT1, low 32 bits. (R/W)

**Register 10.11. SYSTIMER_UNIT1_VALUE_HI_REG (0x0048)**



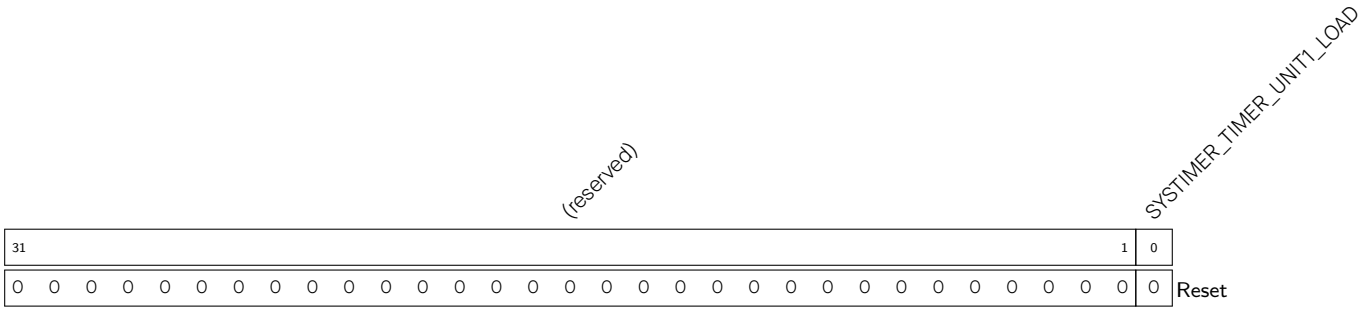

**SYSTIMER_TIMER_UNIT1_VALUE_HI** UNIT1 read value, high 20 bits. (RO)

**Register 10.12. SYSTIMER_UNIT1_VALUE_LO_REG (0x004C)**




**SYSTIMER_TIMER_UNIT1_VALUE_LO** UNIT1 read value, low 32 bits. (RO)

**Register 10.13. SYSTIMER_UNIT1_LOAD_REG (0x0060)**




**SYSTIMER_TIMER_UNIT1_LOAD** UNIT1 synchronization enable signal. Set this bit to reload the values of SYSTIMER_TIMER_UNIT1_LOAD_HI and SYSTIMER_TIMER_UNIT1_LOAD_LO to UNIT1. (WT)

**Register 10.14. SYSTIMER_TARGET0_HI_REG (0x001C)**

| (reserved) | SYSTIMER_TIMER_TARGET0_HI | |
|---|---|---|
| 31 … 20 | 19 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**SYSTIMER_TIMER_TARGET0_HI** The alarm value to be loaded to COMP0, high 20 bits. (R/W)

**Register 10.15. SYSTIMER_TARGET0_LO_REG (0x0020)**

| SYSTIMER_TIMER_TARGET0_LO | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SYSTIMER_TIMER_TARGET0_LO** The alarm value to be loaded to COMP0, low 32 bits. (R/W)

**Register 10.16. SYSTIMER_TARGET0_CONF_REG (0x0034)**

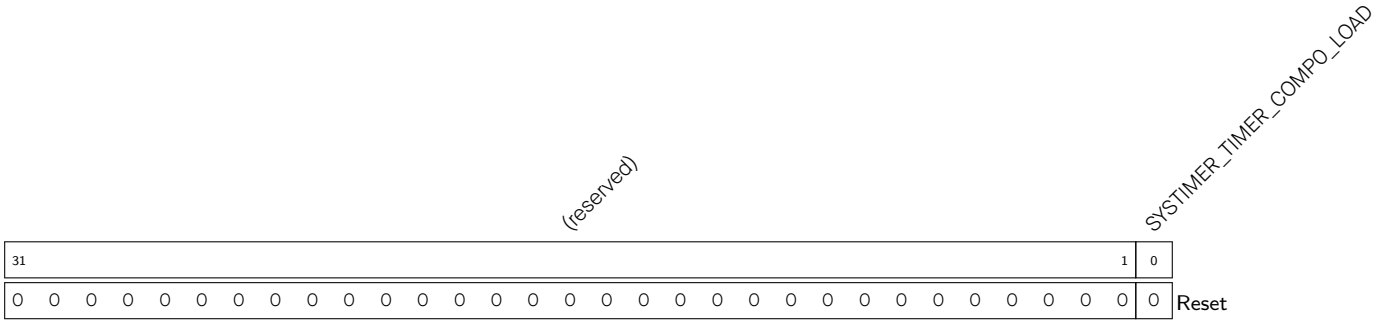
| SYSTIMER_TARGET0_TIMER_UNIT_SEL | SYSTIMER_TARGET0_PERIOD_MODE | (reserved) | SYSTIMER_TARGET0_PERIOD | |
|---|---|---|---|---|
| 31 | 30 | 29 … 26 | 25 … 0 | |
| 0 | 0 | 0 0 0 0 | 0x00000 | Reset |

**SYSTIMER_TARGET0_PERIOD** COMP0 alarm period. (R/W)

**SYSTIMER_TARGET0_PERIOD_MODE** Set COMP0 to period mode. (R/W)

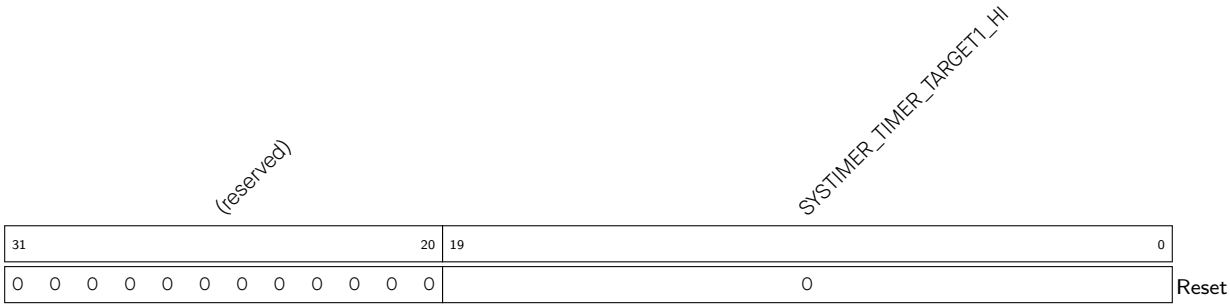
**SYSTIMER_TARGET0_TIMER_UNIT_SEL** Select which unit to compare for COMP0. (R/W)

### Register 10.17. SYSTIMER_COMP0_LOAD_REG (0x0050)



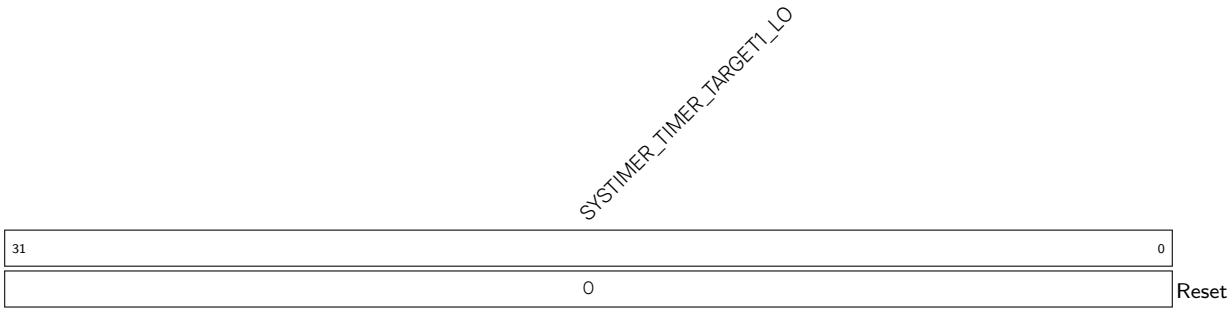

**SYSTIMER_TIMER_COMP0_LOAD** COMP0 synchronization enable signal. Set this bit to reload the alarm value/period to COMP0. (WT)

### Register 10.18. SYSTIMER_TARGET1_HI_REG (0x0024)



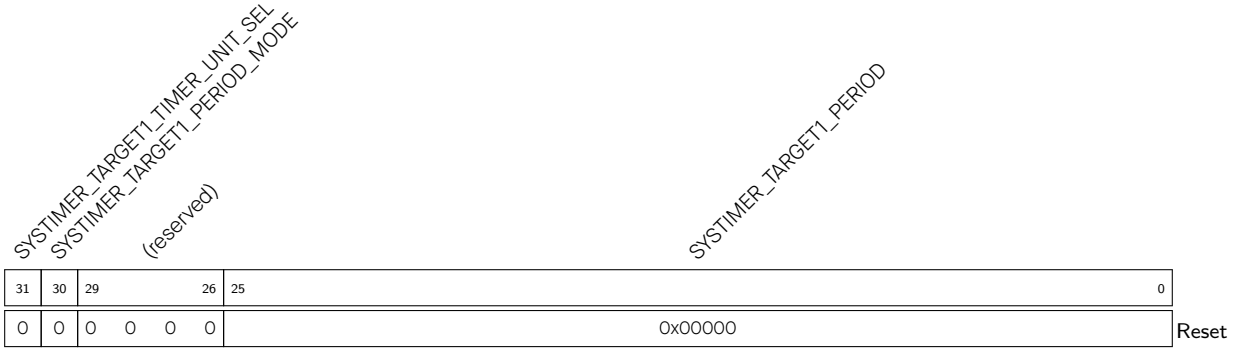

**SYSTIMER_TIMER_TARGET1_HI** The alarm value to be loaded to COMP1, high 20 bits. (R/W)

### Register 10.19. SYSTIMER_TARGET1_LO_REG (0x0028)




**SYSTIMER_TIMER_TARGET1_LO** The alarm value to be loaded to COMP1, low 32 bits. (R/W)

### Register 10.20. SYSTIMER_TARGET1_CONF_REG (0x0038)

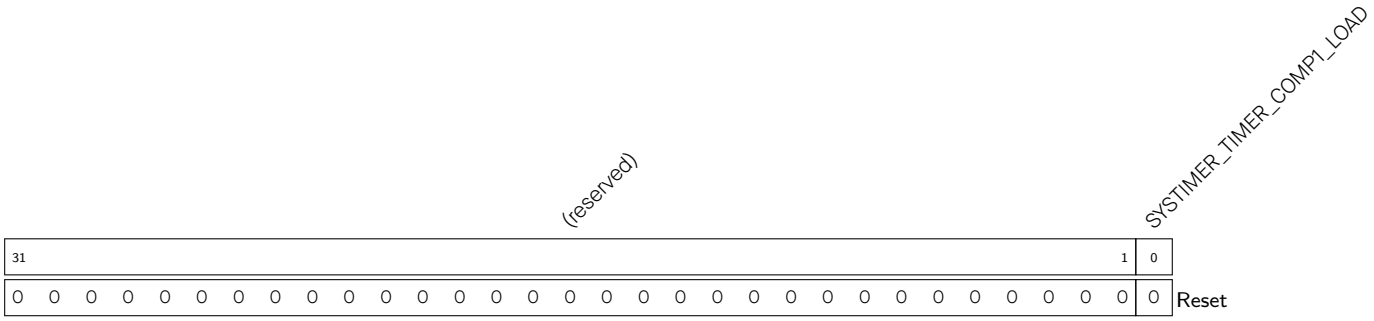



**SYSTIMER_TARGET1_PERIOD** COMP1 alarm period. (R/W)

**SYSTIMER_TARGET1_PERIOD_MODE** Set COMP1 to period mode. (R/W)

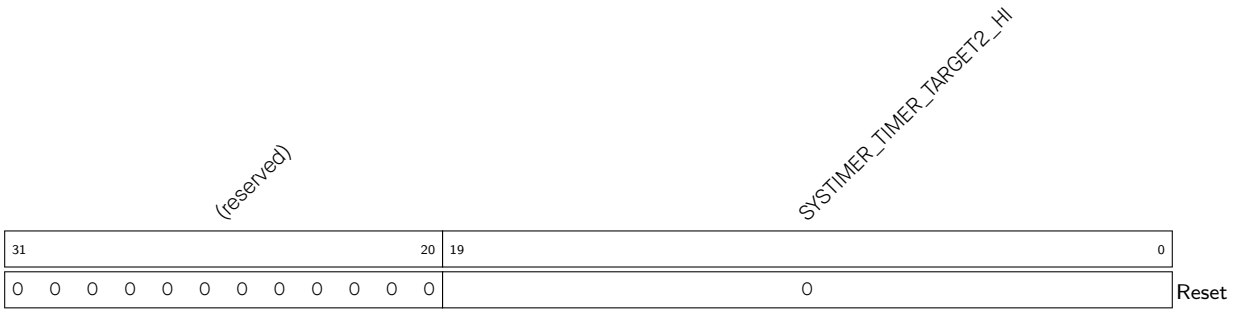
**SYSTIMER_TARGET1_TIMER_UNIT_SEL** Select which unit to compare for COMP1. (R/W)

### Register 10.21. SYSTIMER_COMP1_LOAD_REG (0x0054)




**SYSTIMER_TIMER_COMP1_LOAD** COMP1 synchronization enable signal. Set this bit to reload the alarm value/period to COMP1. (WT)

### Register 10.22. SYSTIMER_TARGET2_HI_REG (0x002C)




**SYSTIMER_TIMER_TARGET2_HI** The alarm value to be loaded to COMP2, high 20 bits. (R/W)

**Register 10.23. SYSTIMER_TARGET2_LO_REG (0x0030)**

| SYSTIMER_TIMER_TARGET2_LO | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SYSTIMER_TIMER_TARGET2_LO** The alarm value to be loaded to COMP2, low 32 bits. (R/W)

**Register 10.24. SYSTIMER_TARGET2_CONF_REG (0x003C)**

| SYSTIMER_TARGET2_TIMER_UNIT_SEL | SYSTIMER_TARGET2_PERIOD_MODE | (reserved) | SYSTIMER_TARGET2_PERIOD | |
|---|---|---|---|---|
| 31 | 30 | 29 … 26 | 25 … 0 | |
| 0 | 0 | 0 0 0 0 | 0x00000 | Reset |

**SYSTIMER_TARGET2_PERIOD** COMP2 alarm period. (R/W)

**SYSTIMER_TARGET2_PERIOD_MODE** Set COMP2 to period mode. (R/W)

**SYSTIMER_TARGET2_TIMER_UNIT_SEL** Select which unit to compare for COMP2. (R/W)

**Register 10.25. SYSTIMER_COMP2_LOAD_REG (0x0058)**

| (reserved) | SYSTIMER_TIMER_COMP2_LOAD | |
|---|---|---|
| 31 … 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 | Reset |

**SYSTIMER_TIMER_COMP2_LOAD** COMP2 synchronization enable signal. Set this bit to reload the alarm value/period to COMP2. (WT)

**Register 10.26. SYSTIMER_INT_ENA_REG (0x0064)**

| (reserved) | SYSTIMER_TARGET2_INT_ENA | SYSTIMER_TARGET1_INT_ENA | SYSTIMER_TARGET0_INT_ENA | |
|---|---|---|---|---|
| 31 … 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 |

**SYSTIMER_TARGET0_INT_ENA** SYSTIMER_TARGET0_INT enable bit. (R/W)

**SYSTIMER_TARGET1_INT_ENA** SYSTIMER_TARGET1_INT enable bit. (R/W)

**SYSTIMER_TARGET2_INT_ENA** SYSTIMER_TARGET2_INT enable bit. (R/W)

**Register 10.27. SYSTIMER_INT_RAW_REG (0x0068)**

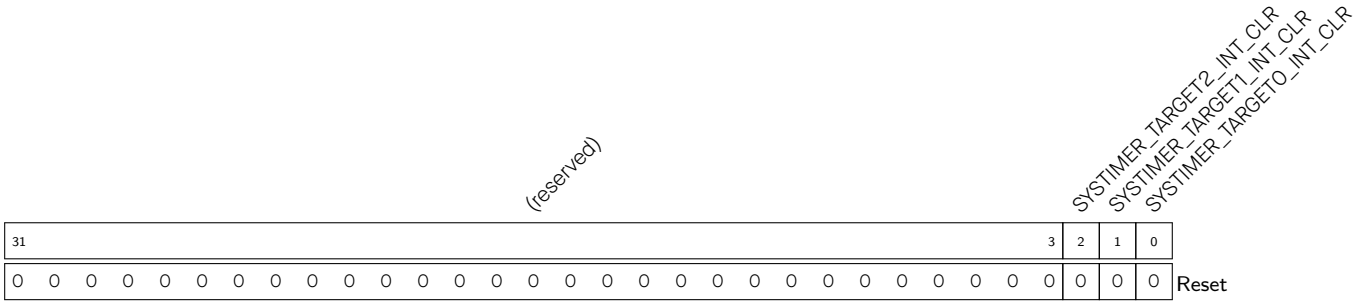
| (reserved) | SYSTIMER_TARGET2_INT_RAW | SYSTIMER_TARGET1_INT_RAW | SYSTIMER_TARGET0_INT_RAW | |
|---|---|---|---|---|
| 31 … 3 | 2 | 1 | 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | Reset |

**SYSTIMER_TARGET0_INT_RAW** SYSTIMER_TARGET0_INT raw bit. (R/WTC/SS)

**SYSTIMER_TARGET1_INT_RAW** SYSTIMER_TARGET1_INT raw bit. (R/WTC/SS)

**SYSTIMER_TARGET2_INT_RAW** SYSTIMER_TARGET2_INT raw bit. (R/WTC/SS)

### Register 10.28. SYSTIMER_INT_CLR_REG (0x006C)

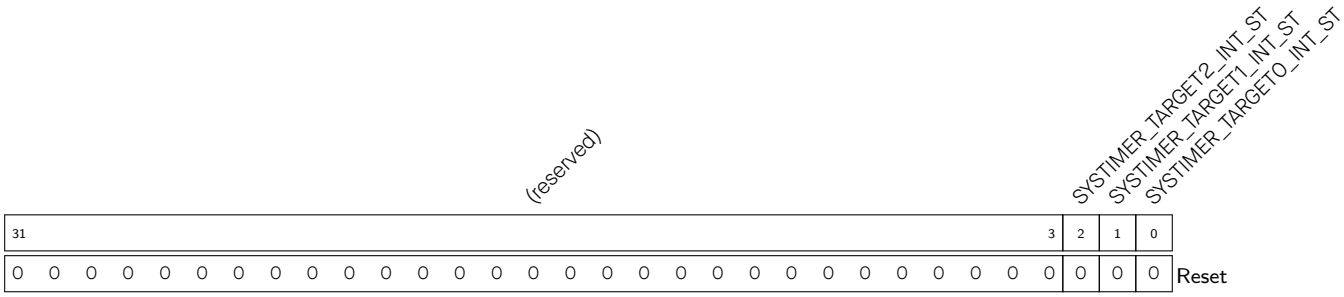



**SYSTIMER_TARGET0_INT_CLR** SYSTIMER_TARGET0_INT clear bit. (WT)

**SYSTIMER_TARGET1_INT_CLR** SYSTIMER_TARGET1_INT clear bit. (WT)

**SYSTIMER_TARGET2_INT_CLR** SYSTIMER_TARGET2_INT clear bit. (WT)

### Register 10.29. SYSTIMER_INT_ST_REG (0x0070)

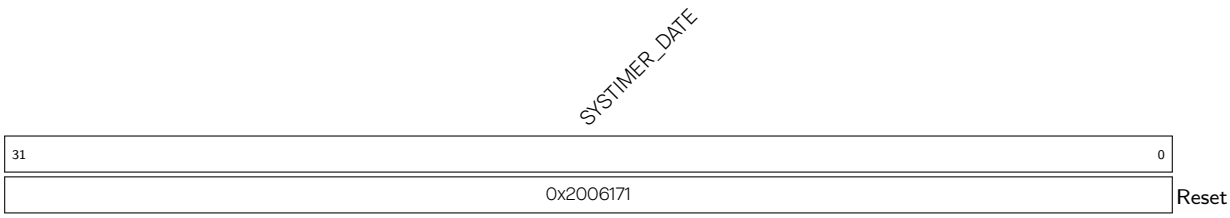



**SYSTIMER_TARGET0_INT_ST** SYSTIMER_TARGET0_INT status bit. (RO)

**SYSTIMER_TARGET1_INT_ST** SYSTIMER_TARGET1_INT status bit. (RO)

**SYSTIMER_TARGET2_INT_ST** SYSTIMER_TARGET2_INT status bit. (RO)

### Register 10.30. SYSTIMER_DATE_REG (0x00FC)




**SYSTIMER_DATE** Version control register. (R/W)

# Chapter 11

# Timer Group (TIMG)

## 11.1 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 11.1-1, the ESP32-C3 chip contains two timer groups, namely timer group 0 and timer group 1. Each timer group consists of one general purpose timer referred to as T0 and one Main System Watchdog Timer. All general purpose timers are based on 16-bit prescalers and 54-bit auto-reload-capable up-down counters.

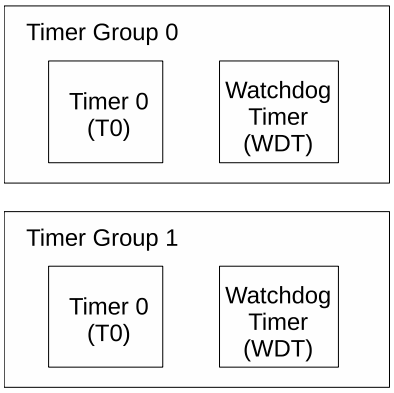



**Figure 11.1-1. Timer Units within Groups**

Note that while the Main System Watchdog Timer registers are described in this chapter, their functional description is included in the Chapter 12 *Watchdog Timers (WDT)*. 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 2 to 65536
- A 54-bit time-base counter programmable to 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 interrupt generation

## 11.2 Functional Description

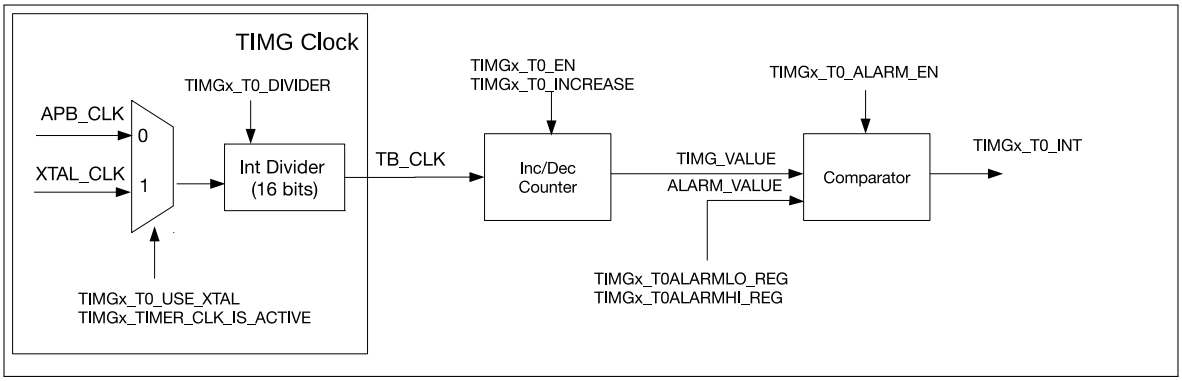



Figure 11.2-1. Timer Group Architecture

Figure11.2-1 is a diagram of timer T0 in a timer group. T0 contains a clock selector, a 16-bit integer divider as a prescaler, a timer-based counter and a comparator for alarm generation.

### 11.2.1 16-bit Prescaler and Clock Selection

The timer can select between the APB clock (APB_CLK) or external clock (XTAL_CLK) as its clock source by setting the TIMG_T0_USE_XTAL field of the TIMG_T0CONFIG_REG register. The selected clock is switched on by setting TIMG_TIMER_CLK_IS_ACTIVE field of the TIMG_REGCLK_REG register to 1 and switched off by setting it to 0. 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. When the TIMG_T0_DIVIDER field is configured as 2 ~ 65536, the divisor of the prescaler would be 2 ~ 65536. Note that programming value 0 to TIMG_T0_DIVIDER will result in the divisor being 65536. When the TIMG_T0_DIVIDER is set to 1, the actual divisor is 2 so the timer counter value represents the half of real time.

To modify the 16-bit prescaler, please first configure the TIMG_T0_DIVIDER field, and then set TIMG_T0_DIVIDER_RST to 1. Meanwhile, the timer must be disabled (i.e. TIMG_T0_EN should be cleared). Otherwise, the result can be unpredictable.

### 11.2.2 54-bit Time-base Counter

The 54-bit time-base counters are based on TB_CLK and can be configured to increment or decrement via the TIMG_T0_INCREASE field. The time-base counter can be enabled or disabled by setting or clearing the TIMG_T0_EN field, respectively. When enabled, the time-base counter increments or decrements on each cycle of TB_CLK. When disabled, the time-base counter is essentially frozen. Note that the TIMG_T0_INCREASE field can be changed while TIMG_T0_EN is set and this will cause the time-base counter to change direction instantly.

To read the 54-bit 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_T0UPDATE_REG, the current value of the 54-bit timer starts to be latched into the TIMG_T0LO_REG and TIMG_T0HI_REG registers containing the lower 32-bits and higher 22-bits, respectively. When TIMG_T0UPDATE_REG is cleared by hardware, it indicates the latch operation has been completed and current timer value can be read from the TIMG_T0LO_REG and TIMG_T0HI_REG registers. TIMG_T0LO_REG and TIMG_T0HI_REG registers will remain unchanged for the CPU to read in its own time until TIMG_T0UPDATE_REG is written to again.

### 11.2.3 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 11.2.4).

The 54-bit alarm value is configured using TIMG_TOALARMLO_REG and TIMG_TOALARMHI_REG, which represent the lower 32-bits and higher 22-bits of the alarm value, respectively. However, the configured alarm value is ineffective until the alarm is enabled by setting the TIMG_TO_ALARM_EN field. To avoid alarm being enabled 'too late' (i.e. the timer value has already passed the alarm value when the alarm is enabled), the hardware will trigger the alarm immediately if the current timer value is higher than the alarm value (within a defined range) when the up-down counter increments, or lower than the alarm value (within a defined range) when the up-down counter decrements. Table 11.2-1 and Table 11.2-2 show the relationship between the current value of the timer, the alarm value, and when an alarm is triggered.The current time value and the alarm value are defined as follows:

- TIMG_VALUE = {TIMG_TOHI_REG, TIMG_TOLO_REG}
- ALARM_VALUE = {TIMG_TOALARMHI_REG, TIMG_TOALARMLO_REG}

Table 11.2-1. Alarm Generation When Up-Down Counter Increments

| Scenario | Range | Alarm |
|---|---|---|
| 1 | ALARM_VALUE − TIMG_VALUE $> 2^{53}$ | Triggered |
| 2 | 0 $<$ ALARM_VALUE − TIMG_VALUE $\leq 2^{53}$ | Triggered when the up-down counter counts TIMG_VALUE up to ALARM_VALUE |
| 3 | 0 $\leq$ TIMG_VALUE − ALARM_VALUE $< 2^{53}$ | Triggered |
| 4 | TIMG_VALUE − ALARM_VALUE $\geq 2^{53}$ | Triggered when the up-down counter restarts counting up from 0 after reaching the timer's maximum value and counts TIMG_VALUE up to ALARM_VALUE |

Table 11.2-2. Alarm Generation When Up-Down Counter Decrements

| Scenario | Range | Alarm |
|---|---|---|
| 5 | TIMG_VALUE − ALARM_VALUE $> 2^{53}$ | Triggered |
| 6 | 0 $<$ TIMG_VALUE − ALARM_VALUE $\leq 2^{53}$ | Triggered when the up-down counter counts TIMG_VALUE down to ALARM_VALUE |
| 7 | 0 $\leq$ ALARM_VALUE − TIMG_VALUE $< 2^{53}$ | Triggered |
| 8 | ALARM_VALUE − TIMG_VALUE $\geq 2^{53}$ | Triggered when the up-down counter restarts counting down from the timer's maximum value after reaching the minimum value and counts TIMG_VALUE down to ALARM_VALUE |

When an alarm occurs, the TIMG_TO_ALARM_EN field is automatically cleared and no alarm will occur again until the TIMG_TO_ALARM_EN is set next time.

### 11.2.4 Timer Reload

A timer is reloaded when a timer's current value is overwritten with a reload value stored in the TIMG_T0_LOAD_LO and TIMG_T0_LOAD_HI fields that correspond to the lower 32-bits and higher 22-bits of the timer's new value, respectively. However, writing a reload value to TIMG_T0_LOAD_LO and TIMG_T0_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_T0LOAD_REG, which causes the timer's current value to be instantly reloaded. If TIMG_T0_EN is set, the timer will continue incrementing or decrementing from the new value. If TIMG_T0_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 or 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_T0_AUTORELOAD field should be set. If not enabled, the timer's value will continue to increment or decrement past the alarm value after an alarm.

### 11.2.5 RTC_SLOW_CLK Frequency Calculation

Via XTAL_CLK, a timer could calculate the frequency of clock sources for RTC_SLOW_CLK (i.e. RC_RTC_SLOW_CLK, RC_FAST_DIV_CLK, and XTAL32K_CLK) as follows:

1. Start periodic or one-shot frequency calculation;
2. Once receiving the signal to start calculation, the counter of XTAL_CLK and the counter of RTC_SLOW_CLK begin to work at the same time. When the counter of RTC_SLOW_CLK counts to C0, the two counters stop counting simultaneously;
3. Assume the value of XTAL_CLK's counter is C1, and the frequency of RTC_SLOW_CLK would be calculated as: $f_rtc = \frac{C0 \times f_XTAL_CLK}{C1}$

### 11.2.6 Interrupts

Each timer has its own interrupt line that can be routed to the CPU, and thus each timer group has a total of two interrupt lines. Timers generate level interrupts that must be explicitly cleared by the CPU on each triggering.

Interrupts are triggered after an alarm (or stage timeout for watchdog timers) occurs. Level interrupts will be held high after an alarm (or stage timeout) occurs, and will remain so until manually cleared. To enable a timer's interrupt, the TIMG_T0_INT_ENA bit should be set.

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

- TIMG_T0_INT_RAW : An alarm event sets it to 1. The bit will remain set until the timer's corresponding bit in TIMG_T0_INT_CLR is written.
- TIMG_WDT_INT_RAW : A stage time out will set the timer's bit to 1. The bit will remain set until the timer's corresponding bit in TIMG_WDT_INT_CLR is written.

- TIMG_TO_INT_ST : Reflects the status of each timer's interrupt and is generated by masking the bits of TIMG_TO_INT_RAW with TIMG_TO_INT_ENA.
- 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.
- TIMG_TO_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_TO_INT_CLR : Used to clear a timer's interrupt by setting its corresponding bit to 1. The timer's corresponding bit in TIMG_TO_INT_RAW and TIMG_TO_INT_ST will be cleared as a result. Note that a timer's interrupt must be cleared before the next interrupt occurs.
- 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.

## 11.3 Configuration and Usage

### 11.3.1 Timer as a Simple Clock

1. Configure the time-base counter
   - Select clock source by setting or clearing TIMG_TO_USE_XTAL field.
   - Configure the 16-bit prescaler by setting TIMG_TO_DIVIDER.
   - Configure the timer direction by setting or clearing TIMG_TO_INCREASE.
   - Set the timer's starting value by writing the starting value to TIMG_TO_LOAD_LO and TIMG_TO_LOAD_HI, then reloading it into the timer by writing any value to TIMG_TOLOAD_REG.
2. Start the timer by setting TIMG_TO_EN.
3. Get the timer's current value.
   - Write any value to TIMG_TOUPDATE_REG to latch the timer's current value.
   - Wait until TIMG_TOUPDATE_REG is cleared by hardware.
   - Read the latched timer value from TIMG_TOLO_REG and TIMG_TOHI_REG.

### 11.3.2 Timer as One-shot Alarm

1. Configure the time-base counter following step 1 of Section 11.3.1.
2. Configure the alarm.
   - Configure the alarm value by setting TIMG_TOALARMLO_REG and TIMG_TOALARMHI_REG.
   - Enable interrupt by setting TIMG_TO_INT_ENA.
3. Disable auto reload by clearing TIMG_TO_AUTORELOAD.
4. Start the alarm by setting TIMG_TO_ALARM_EN.

5. Handle the alarm interrupt.
   - Clear the interrupt by setting the timer's corresponding bit in TIMG_TO_INT_CLR.
   - Disable the timer by clearing TIMG_TO_EN.

### 11.3.3 Timer as Periodic Alarm

1. Configure the time-base counter following step 1 in Section 11.3.1.
2. Configure the alarm following step 2 in Section 11.3.2.
3. Enable auto reload by setting TIMG_TO_AUTORELOAD and configure the reload value via TIMG_TO_LOAD_LO and TIMG_TO_LOAD_HI.
4. Start the alarm by setting TIMG_TO_ALARM_EN.
5. Handle the alarm interrupt (repeat on each alarm iteration).
   - Clear the interrupt by setting the timer's corresponding bit in TIMG_TO_INT_CLR.
   - If the next alarm requires a new alarm value and reload value (i.e. different alarm interval per iteration), then TIMG_TOALARMLO_REG, TIMG_TOALARMHI_REG, TIMG_TO_LOAD_LO, and TIMG_TO_LOAD_HI should be reconfigured as needed. Otherwise, the aforementioned registers should remain unchanged.
   - Re-enable the alarm by setting TIMG_TO_ALARM_EN.
6. Stop the timer (on final alarm iteration).
   - Clear the interrupt by setting the timer's corresponding bit in TIMG_TO_INT_CLR.
   - Disable the timer by clearing TIMG_TO_EN.

### 11.3.4 RTC_SLOW_CLK Frequency Calculation

1. One-shot frequency calculation
   - Select the clock whose frequency is to be calculated (clock source of RTC_SLOW_CLK) via TIMG_RTC_CALI_CLK_SEL, and configure the time of calculation via TIMG_RTC_CALI_MAX.
   - Select one-shot frequency calculation by clearing TIMG_RTC_CALI_START_CYCLING, and enable the two counters via TIMG_RTC_CALI_START.
   - Once TIMG_RTC_CALI_RDY becomes 1, read TIMG_RTC_CALI_VALUE to get the value of XTAL_CLK's counter, and calculate the frequency of RTC_SLOW_CLK.
2. Periodic frequency calculation
   - Select the clock whose frequency is to be calculated (clock source of RTC_SLOW_CLK) via TIMG_RTC_CALI_CLK_SEL, and configure the time of calculation via TIMG_RTC_CALI_MAX.
   - Select periodic frequency calculation by enabling TIMG_RTC_CALI_START_CYCLING.
   - When TIMG_RTC_CALI_CYCLING_DATA_VLD is 1, TIMG_RTC_CALI_VALUE is valid.
3. Timeout
   If the counter of RTC_SLOW_CLK cannot finish counting in TIMG_RTC_CALI_TIMEOUT_RST_CNT cycles, TIMG_RTC_CALI_TIMEOUT will be set to indicate a timeout.

## 11.4 Register Summary

The addresses in this section are relative to Timer Group base addresses (one for Timer Group 0 and another one for Timer Group 1) provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **T0 control and configuration registers** | | | |
| TIMG_T0CONFIG_REG | Timer 0 configuration register | 0x0000 | varies |
| TIMG_T0LO_REG | Timer 0 current value, low 32 bits | 0x0004 | RO |
| TIMG_T0HI_REG | Timer 0 current value, high 22 bits | 0x0008 | RO |
| TIMG_T0UPDATE_REG | Write to copy current timer value to TIMGn_T0_(LO/HI)_REG | 0x000C | R/W/SC |
| TIMG_T0ALARMLO_REG | Timer 0 alarm value, low 32 bits | 0x0010 | R/W |
| TIMG_T0ALARMHI_REG | Timer 0 alarm value, high bits | 0x0014 | R/W |
| TIMG_T0LOADLO_REG | Timer 0 reload value, low 32 bits | 0x0018 | R/W |
| TIMG_T0LOADHI_REG | Timer 0 reload value, high 22 bits | 0x001C | R/W |
| TIMG_T0LOAD_REG | Write to reload timer from TIMG_T0_(LOADLO/LOADHI)_REG | 0x0020 | WT |
| **WDT control and configuration registers** | | | |
| TIMG_WDTCONFIG0_REG | Watchdog timer configuration register | 0x0048 | varies |
| TIMG_WDTCONFIG1_REG | Watchdog timer prescaler register | 0x004C | varies |
| 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 | WT |
| TIMG_WDTWPROTECT_REG | Watchdog write protect register | 0x0064 | R/W |
| **RTC frequency calculation control and configuration registers** | | | |
| TIMG_RTCCALICFG_REG | RTC frequency calculation configuration register 0 | 0x0068 | varies |
| TIMG_RTCCALICFG1_REG | RTC frequency calculation configuration register 1 | 0x006C | RO |
| TIMG_RTCCALICFG2_REG | RTC frequency calculation configuration register 2 | 0x0080 | varies |
| **Interrupt registers** | | | |
| TIMG_INT_ENA_TIMERS_REG | Interrupt enable bits | 0x0070 | R/W |
| TIMG_INT_RAW_TIMERS_REG | Raw interrupt status | 0x0074 | R/SS/WTC |
| TIMG_INT_ST_TIMERS_REG | Masked interrupt status | 0x0078 | RO |
| TIMG_INT_CLR_TIMERS_REG | Interrupt clear bits | 0x007C | WT |
| **Version register** | | | |
| TIMG_NTIMERS_DATE_REG | Timer version control register | 0x00F8 | R/W |
| **Clock configuration registers** | | | |
| TIMG_REGCLK_REG | Timer group clock gate register | 0x00FC | R/W |

## 11.5 Registers

The addresses in this section are relative to Timer Group base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 11.1. TIMG_T0CONFIG_REG (0x0000)**

| TIMG_T0_EN | TIMG_T0_INCREASE | TIMG_T0_AUTORELOAD | TIMG_T0_DIVIDER | TIMG_T0_DIVIDER_RST | (reserved) | TIMG_T0_ALARM_EN | TIMG_T0_USE_XTAL | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 … 13 | 12 | 11 | 10 | 9 | 8 … 0 | |
| 0 | 1 | 1 | 0x01 | 0 | 0 | 0 | 0 | 0 0 0 0 0 0 0 0 0 | Reset |

**TIMG_T0_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_T0_ALARM_EN** When set, the alarm is enabled. This bit is automatically cleared once an alarm occurs. (R/W/SC)

**TIMG_T0_DIVIDER_RST** When set, Timer 0 's clock divider counter will be reset. (WT)

**TIMG_T0_DIVIDER** Timer 0 clock (T0_clk) prescaler value. (R/W)

**TIMG_T0_AUTORELOAD** When set, Timer 0 auto-reload at alarm is enabled. (R/W)

**TIMG_T0_INCREASE** When set, the Timer 0 time-base counter will increment every clock tick. When cleared, the Timer 0 time-base counter will decrement. (R/W)

**TIMG_T0_EN** When set, the Timer 0 time-base counter is enabled. (R/W)

**Register 11.2. TIMG_T0LO_REG (0x0004)**

| TIMG_T0_LO | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**TIMG_T0_LO** After writing to TIMG_T0UPDATE_REG, the low 32 bits of the time-base counter of Timer 0 can be read here. (RO)

**Register 11.3. TIMG_T0HI_REG (0x0008)**

| (reserved) | TIMG_T0_HI | |
|---|---|---|
| 31 … 22 | 21 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 | 0x0000 | Reset |

**TIMG_T0_HI** After writing to TIMG_T0UPDATE_REG, the high 22 bits of the time-base counter of Timer 0 can be read here. (RO)

**Register 11.4. TIMG_T0UPDATE_REG (0x000C)**

| TIMG_T0_UPDATE | (reserved) | |
|---|---|---|
| 31 | 30 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 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 |

**TIMG_T0_UPDATE** After writing 0 or 1 to TIMG_T0UPDATE_REG, the counter value is latched. (R/W/SC)

**Register 11.5. TIMG_T0ALARMLO_REG (0x0010)**

| TIMG_T0_ALARM_LO | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

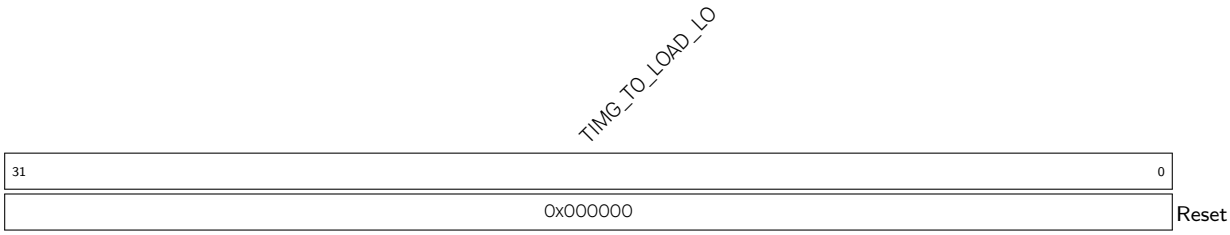
**TIMG_T0_ALARM_LO** Timer 0 alarm trigger time-base counter value, low 32 bits. (R/W)

**Register 11.6. TIMG_T0ALARMHI_REG (0x0014)**

| (reserved) | TIMG_T0_ALARM_HI | |
|---|---|---|
| 31 … 22 | 21 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 | 0x0000 | Reset |

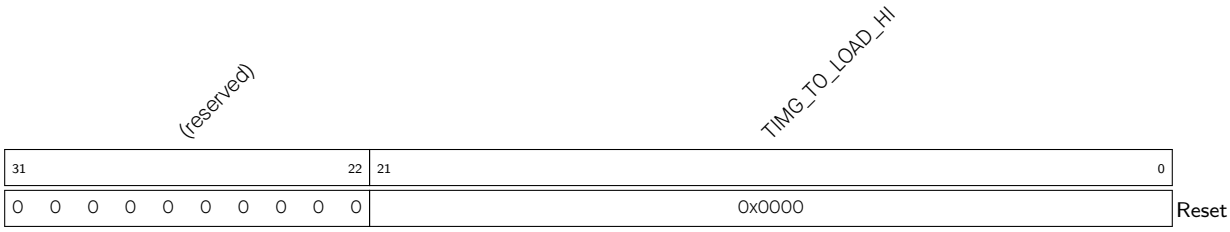
**TIMG_T0_ALARM_HI** Timer 0 alarm trigger time-base counter value, high 22 bits. (R/W)

### Register 11.7. TIMG_TOLOADLO_REG (0x0018)



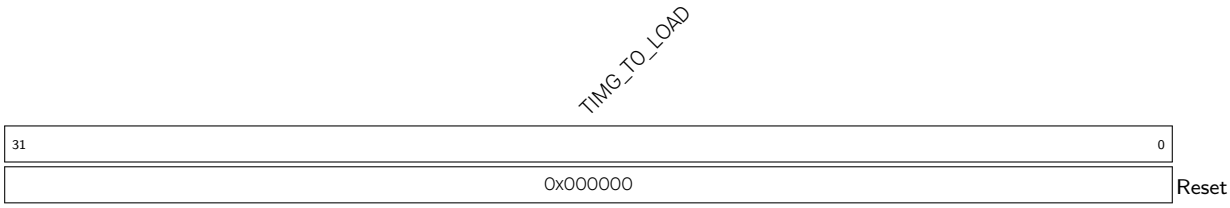

**TIMG_T0_LOAD_LO** Low 32 bits of the value that a reload will load onto Timer 0 time-base counter. (R/W)

### Register 11.8. TIMG_TOLOADHI_REG (0x001C)




**TIMG_T0_LOAD_HI** High 22 bits of the value that a reload will load onto Timer 0 time-base counter. (R/W)

### Register 11.9. TIMG_TOLOAD_REG (0x0020)




**TIMG_T0_LOAD** Write any value to trigger a Timer 0 time-base counter reload. (WT)

**Register 11.10. TIMG_WDTCONFIG0_REG (0x0048)**

| TIMG_WDT_EN | TIMG_WDT_STG0 | TIMG_WDT_STG1 | TIMG_WDT_STG2 | TIMG_WDT_STG3 | TIMG_WDT_CONF_UPDATE_EN | TIMG_WDT_USE_XTAL | TIMG_WDT_CPU_RESET_LENGTH | TIMG_WDT_SYS_RESET_LENGTH | TIMG_WDT_FLASHBOOT_MOD_EN | TIMG_WDT_PROCPU_RESET_EN | TIMG_WDT_APPCPU_RESET_EN | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 29 | 28 27 | 26 25 | 24 23 | 22 | 21 | 20 18 | 17 15 | 14 | 13 | 12 | 11 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x1 | 0x1 | 1 | 0 | 0 | 0 0 0 0 0 0 0 0 0 0 0 0 | Reset |

**TIMG_WDT_APPCPU_RESET_EN** WDT reset CPU enable. (R/W)

**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 $\mu$s, 7: 3.2 $\mu$s. (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 $\mu$s, 7: 3.2 $\mu$s. (R/W)

**TIMG_WDT_USE_XTAL** Chooses WDT clock. 0: APB_CLK; 1:XTAL_CLK. (R/W)

**TIMG_WDT_CONF_UPDATE_EN** Updates the WDT configuration registers. (WT)

**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)

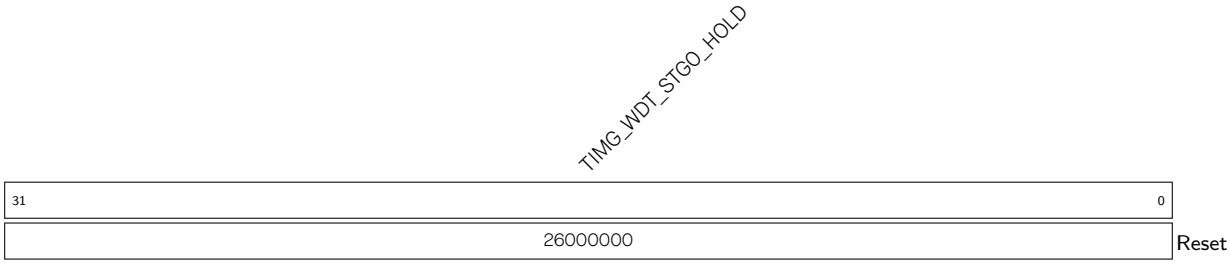
**Register 11.11. TIMG_WDTCONFIG1_REG (0x004C)**

| TIMG_WDT_CLK_PRESCALE | (reserved) | TIMG_WDT_DIVCNT_RST | |
|---|---|---|---|
| 31 16 | 15 1 | 0 | |
| 0x01 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**TIMG_WDT_DIVCNT_RST** When set, WDT 's clock divider counter will be reset. (WT)

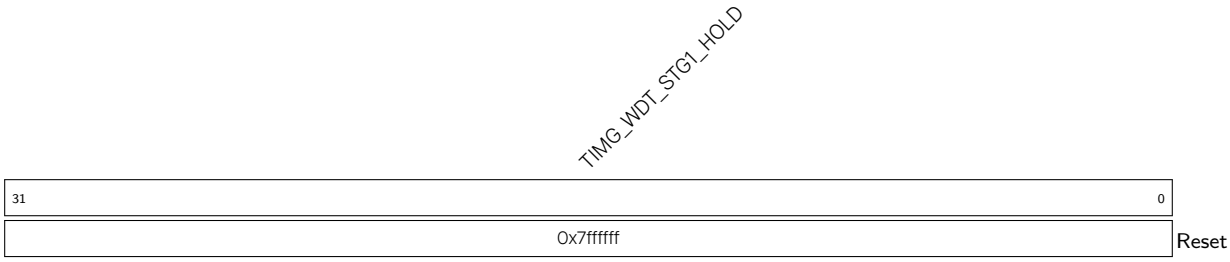
**TIMG_WDT_CLK_PRESCALE** MWDT clock prescaler value. MWDT clock period = MWDT's clock source period * TIMG_WDT_CLK_PRESCALE. (R/W)

**Register 11.12. TIMG_WDTCONFIG2_REG (0x0050)**



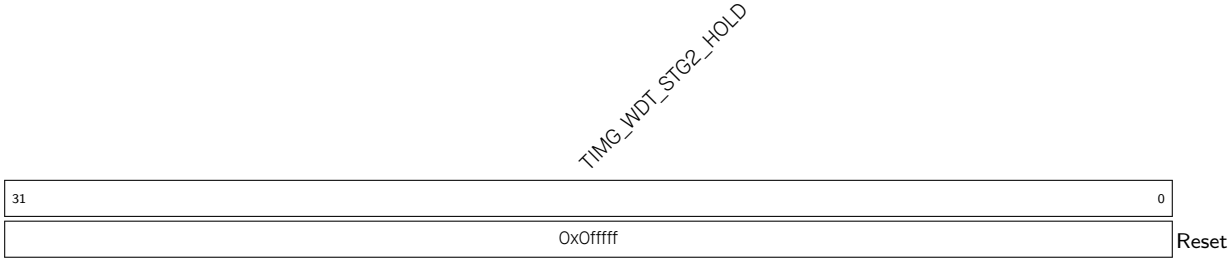

**TIMG_WDT_STG0_HOLD** Stage 0 timeout value, in MWDT clock cycles. (R/W)

**Register 11.13. TIMG_WDTCONFIG3_REG (0x0054)**




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

**Register 11.14. TIMG_WDTCONFIG4_REG (0x0058)**




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

**Register 11.15. TIMG_WDTCONFIG5_REG (0x005C)**

| 31 | 0 | |
|---|---|---|
| TIMG_WDT_STG3_HOLD | | |
| 0x0fffff | | Reset |

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

**Register 11.16. TIMG_WDTFEED_REG (0x0060)**

| 31 | 0 | |
|---|---|---|
| TIMG_WDT_FEED | | |
| 0x000000 | | Reset |

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

**Register 11.17. TIMG_WDTWPROTECT_REG (0x0064)**

| 31 | 0 | |
|---|---|---|
| TIMG_WDT_WKEY | | |
| 0x50d83aa1 | | Reset |

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

**Register 11.18. TIMG_RTCCALICFG_REG (0x0068)**

| TIMG_RTC_CALI_START | TIMG_RTC_CALI_MAX | TIMG_RTC_CALI_RDY | TIMG_RTC_CALI_CLK_SEL | TIMG_RTC_CALI_START_CYCLING | (reserved) | |
|---|---|---|---|---|---|---|
| 31 | 30 … 16 | 15 | 14 … 13 | 12 | 11 … 0 | |
| 0 | 0x01 | 0 | 0x1 | 1 | 0 0 0 0 0 0 0 0 0 0 0 0 | Reset |

**TIMG_RTC_CALI_START_CYCLING** Enables periodic frequency calculation. (R/W)

**TIMG_RTC_CALI_CLK_SEL** 0: RC_SLOW_CLK; 1: RC_FAST_DIV_CLK; 2: XTAL32K_CLK. (R/W)

**TIMG_RTC_CALI_RDY** Marks the completion of one-shot frequency calculation. (RO)

**TIMG_RTC_CALI_MAX** Configures the time to calculate the frequency of RTC_SLOW_CLK. Measurement unit: RTC_SLOW_CLK cycle. (R/W)

**TIMG_RTC_CALI_START** Set this bit to enable one-shot frequency calculation. (R/W)

**Register 11.19. TIMG_RTCCALICFG1_REG (0x006C)**

| TIMG_RTC_CALI_VALUE | (reserved) | TIMG_RTC_CALI_CYCLING_DATA_VLD | |
|---|---|---|---|
| 31 … 7 | 6 … 1 | 0 | |
| 0x00000 | 0 0 0 0 0 0 | 0 | Reset |

**TIMG_RTC_CALI_CYCLING_DATA_VLD** Marks the completion of periodic frequency calculation. (RO)

**TIMG_RTC_CALI_VALUE** When one-shot or periodic frequency calculation completes, read this value to calculate the frequency of RTC_SLOW_CLK. Measurement unit: XTAL_CLK cycle. (RO)

**Register 11.20. TIMG_RTCCALICFG2_REG (0x0080)**

| TIMG_RTC_CALI_TIMEOUT_THRES | TIMG_RTC_CALI_TIMEOUT_RST_CNT | (reserved) | TIMG_RTC_CALI_TIMEOUT | |
|---|---|---|---|---|
| 31 … 7 | 6 … 3 | 2 … 1 | 0 | |
| 0x1fffffff | 3 | 0 0 | 0 | Reset |

**TIMG_RTC_CALI_TIMEOUT** Indicates frequency calculation timeout. (RO)

**TIMG_RTC_CALI_TIMEOUT_RST_CNT** Cycles to reset frequency calculation timeout. (R/W)

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

**Register 11.21. TIMG_INT_ENA_TIMERS_REG (0x0070)**

| (reserved) | TIMG_WDT_INT_ENA | TIMG_TO_INT_ENA | |
|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**TIMG_TO_INT_ENA** The interrupt enable bit for the TIMG_TO_INT interrupt. (R/W)

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

**Register 11.22. TIMG_INT_RAW_TIMERS_REG (0x0074)**

| (reserved) | TIMG_WDT_INT_RAW | TIMG_TO_INT_RAW | |
|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**TIMG_TO_INT_RAW** The raw interrupt status bit for the TIMG_TO_INT interrupt. (R/SS/WTC)

**TIMG_WDT_INT_RAW** The raw interrupt status bit for the TIMG_WDT_INT interrupt. (R/SS/WTC)

### Register 11.23. TIMG_INT_ST_TIMERS_REG (0x0078)

| (reserved) | TIMG_WDT_INT_ST | TIMG_TO_INT_ST | |
|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**TIMG_TO_INT_ST** The masked interrupt status bit for the TIMG_TO_INT interrupt. (RO)

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

### Register 11.24. TIMG_INT_CLR_TIMERS_REG (0x007C)

| (reserved) | TIMG_WDT_INT_CLR | TIMG_TO_INT_CLR | |
|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**TIMG_TO_INT_CLR** Set this bit to clear the TIMG_TO_INT interrupt. (WT)

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

### Register 11.25. TIMG_NTIMERS_DATE_REG (0x00F8)

| (reserved) | TIMG_NTIMGS_DATE | |
|---|---|---|
| 31 … 28 | 27 … 0 | |
| 0 0 0 0 | 0x2006191 | Reset |

**TIMG_NTIMGS_DATE** Timer version control register (R/W)

**Register 11.26. TIMG_REGCLK_REG (0x00FC)**

| TIMG_CLK_EN | TIMG_TIMER_CLK_IS_ACTIVE | TIMG_WDT_CLK_IS_ACTIVE | (reserved) | |
|---|---|---|---|---|
| 31 | 30 | 29 | 28 … 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 0 0 0 0 0 0 0 0 0 | Reset |

**TIMG_WDT_CLK_IS_ACTIVE** enable WDT's clock (R/W)

**TIMG_TIMER_CLK_IS_ACTIVE** enable Timer 0's clock (R/W)

**TIMG_CLK_EN** Register clock gate signal. 0: The clock used by software to read and write registers is on only when there is software operation. 1: The clock used by software to read and write registers is always on. (R/W)

# Chapter 12

# Watchdog Timers (WDT)

## 12.1 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 timeout. Therefore, watchdog timers are useful for detecting and handling erroneous system/software behavior.

As shown in Figure 12.1-1, ESP32-C3 contains three digital watchdog timers: one in each of the two timer groups in Chapter 11 *Timer Group (TIMG)*(called Main System Watchdog Timers, or MWDT) and one in the RTC Module (called the RTC Watchdog Timer, or RWDT). Each digital watchdog timer allows for four separately configurable stages and each stage can be programmed to take one action upon expiry, unless the watchdog is fed or disabled. MWDT supports three timeout actions: interrupt, CPU reset, and core reset, while RWDT supports four timeout actions: interrupt, CPU reset, core reset, and system reset (see details in Section 12.2.2.2 *Stages and Timeout Actions*). A timeout value can be set for each stage individually.

During the flash boot process, RWDT and the first MWDT in timergroup 0 are enabled automatically in order to detect and recover from booting errors.

ESP32-C3 also has one analog watchdog timer: Super watchdog (SWD). It is an ultra-low-power circuit in analog domain that helps to prevent the system from operating in a sub-optimal state and resets the system if required.

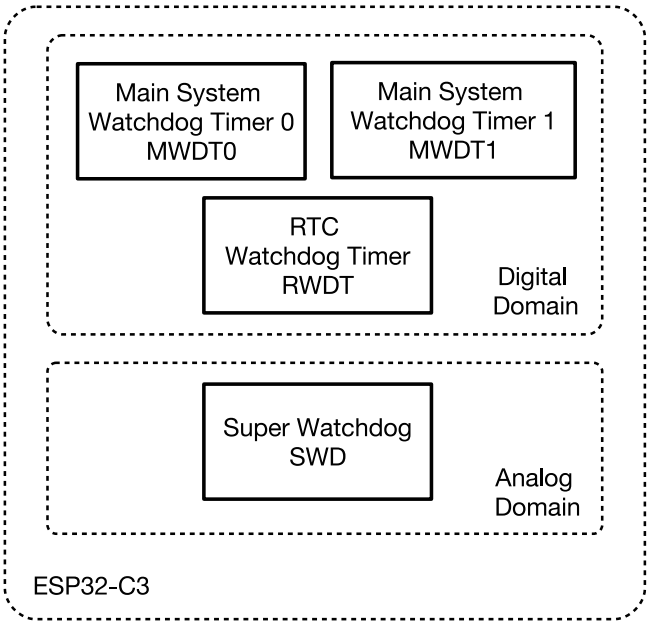



Figure 12.1-1. Watchdog Timers Overview

Note that while this chapter provides the functional descriptions of the watchdog timer's, their register descriptions are provided in Chapter 11 *Timer Group (TIMG)* and Chapter 9 *Low-power Management*.

## 12.2 Digital Watchdog Timers

### 12.2.1 Features

Watchdog timers have the following features:

- Four stages, each with a programmable timeout value. Each stage can be configured and enabled/disabled separately
- Three timeout actions (interrupt, CPU reset, or core reset) for MWDT and four timeout actions (interrupt, CPU reset, core reset, or system reset) for RWDT 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.

## 12.2.2 Functional Description

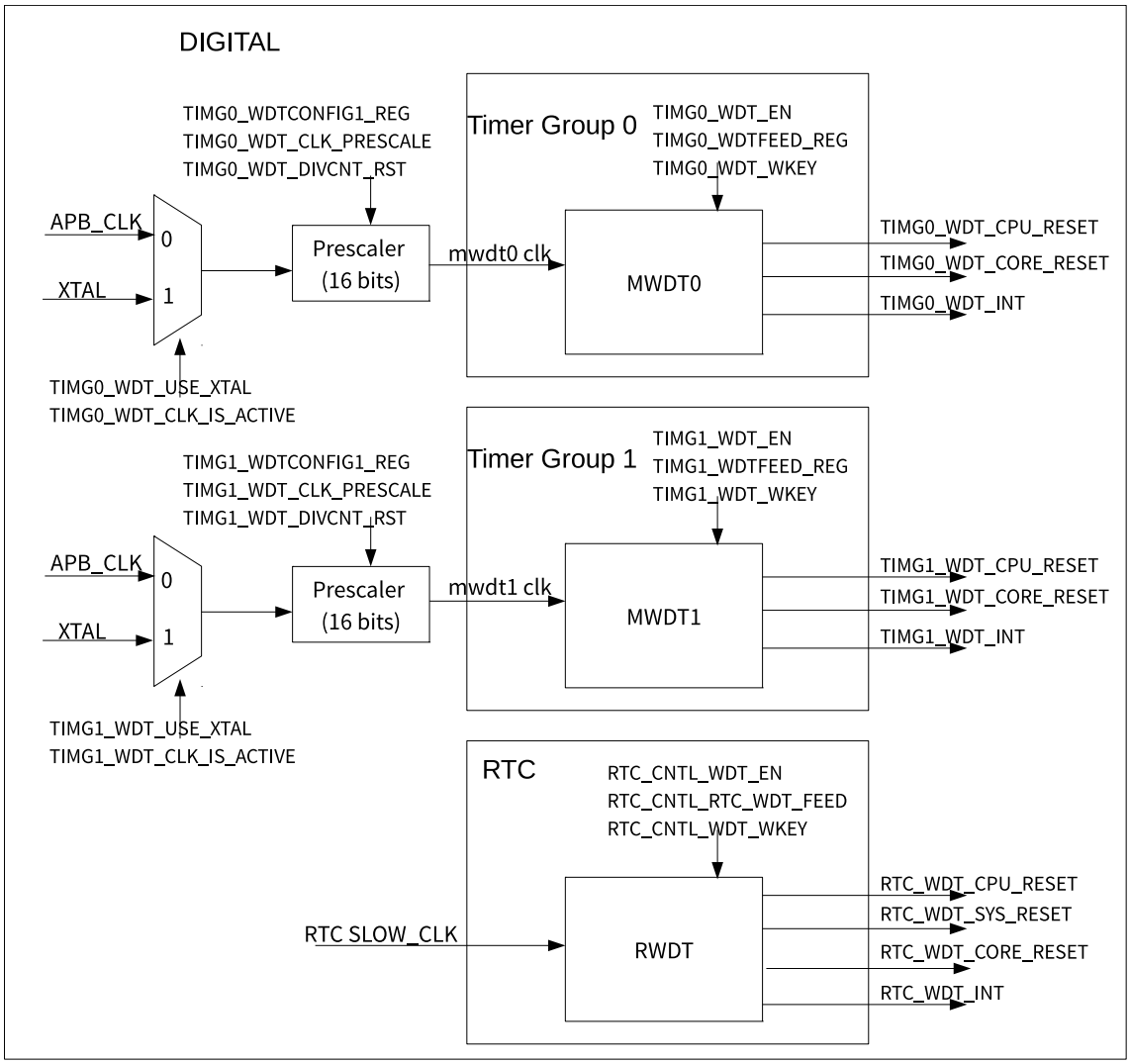



Figure 12.2-1. Watchdog Timers in ESP32-C3

Figure 12.2-1 shows the three watchdog timers in ESP32-C3 digital systems.

### 12.2.2.1 Clock Source and 32-Bit Counter

At the core of each watchdog timer is a 32-bit counter.

MWDTs can select between the APB clock (APB_CLK) or external clock (XTAL_CLK) as its clock source by setting the TIMG_WDT_USE_XTAL field of the TIMG_WDTCONFIGO_REG register. The selected clock is switched on by setting TIMG_WDT_CLK_IS_ACTIVE field of the TIMG_REGCLK_REG register to 1 and switched off by setting it to 0. Then the selected clock is divided by a 16-bit configurable prescaler. The 16-bit prescaler for MWDTs is configured via the TIMG_WDT_CLK_PRESCALE field of TIMG_WDTCONFIG1_REG.When TIMG_WDT_DIVCNT_RST field is set, the prescaler is reset and it can be re-configured at once.

In contrast, the clock source of RWDT is derived directly from an RTC slow clock (the RTC slow clock source shown in Chapter 6 *Reset and Clock*).

MWDTs and RWDT are enabled by setting the TIMG_WDT_EN and RTC_CNTL_WDT_EN fields respectively.

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

#### 12.2.2.2 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 using RTC_CNTL_WDT_STG*j*_HOLD field (where *j* ranges from 0 to 3).

Please note that the timeout value of stage 0 for RWDT ($Thold_0$) is determined by the combination of the
EFUSE_WDT_DELAY_SEL field of eFuse register EFUSE_RD_REPEAT_DATA1_REG and
RTC_CNTL_WDT_STG0_HOLD. The relationship is as follows:

$$T_{hold0} = RTC_CNTL_WDT_STG0_HOLD << (EFUSE_WDT_DELAY_SEL + 1)$$

where $<<$ is a left-shift operator.

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.
- CPU reset – Reset a CPU core
  When the stage expires, the CPU core will be reset.
- Core reset – Reset the main system
  When the stage expires, the main system (which includes MWDTs, CPU, and all peripherals) will be reset. The power management unit and RTC peripheral will not be reset.
- System reset – Reset the main system, power management unit and RTC peripheral
  When the stage expires the main system, power management unit and RTC peripheral (see details in Chapter 9 *Low-power Management*) will all 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_CNTL_WDTCONFIG0_REG.

#### 12.2.2.3 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 field 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 field 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 field itself) whilst the write-key field'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 field.
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 field.

#### 12.2.2.4 Flash Boot Protection

During flash booting process, MWDT in timer group 0 (see Figure 11.1-1 *Timer Units within Groups*), as well as RWDT, are automatically enabled. Stage 0 for the enabled MWDT is automatically configured to reset the system upon expiry, known as core reset. Likewise, stage 0 for RWDT is configured to system reset, which resets 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.

## 12.3 Super Watchdog

Super watchdog (SWD) is an ultra-low-power circuit in analog domain that helps to prevent the system from operating in a sub-optimal state and resets the system if required. SWD contains a watchdog circuit that needs to be fed for at least once during its timeout period, which is slightly less than one second. About 100 ms before watchdog timeout, it will also send out a WD_INTR signal as a request to remind the system to feed the watchdog.

If the system doesn't respond to SWD feed request and watchdog finally times out, SWD will generate a system level signal SWD_RSTB to reset whole digital circuits on the chip.

### 12.3.1 Features

SWD has the following features:

- Ultra-low power
- Interrupt to indicate that the SWD timeout period is close to expiring
- Various dedicated methods for software to feed SWD, which enables SWD to monitor the working state of the whole operating system

### 12.3.2 Super Watchdog Controller

#### 12.3.2.1 Structure

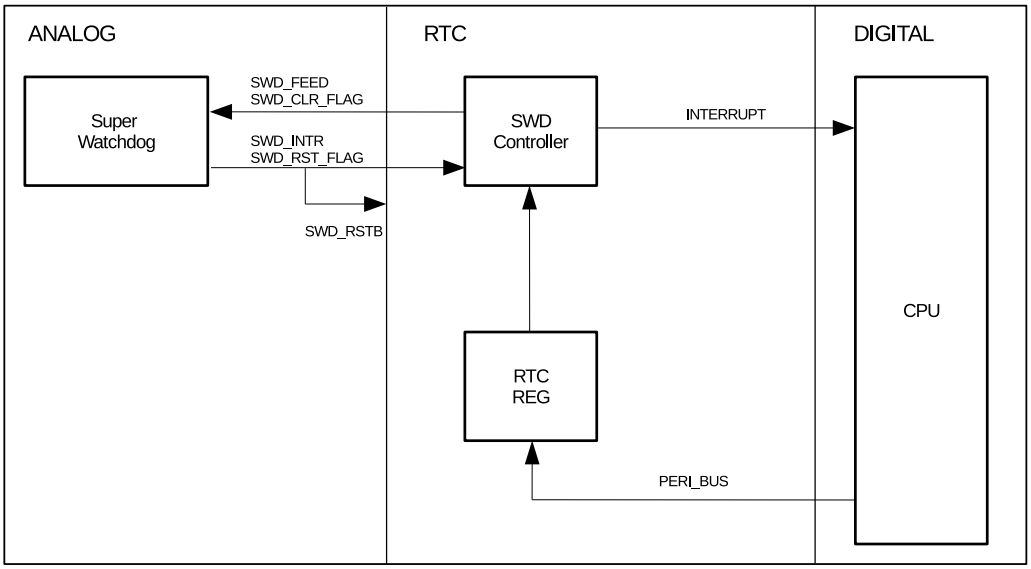



Figure 12.3-1. Super Watchdog Controller Structure

#### 12.3.2.2 Workflow

In normal state:

- SWD controller receives feed request from SWD.
- SWD controller can send an interrupt to main CPU.
- Main CPU can feed SWD directly by setting RTC_CNTL_SWD_FEED.
- When trying to feed SWD, CPU needs to disable SWD controller's write protection by writing 0x8F1D312A to RTC_CNTL_SWD_WKEY. This prevents SWD from being fed by mistake when the system is operating in sub-optimal state.
- If setting RTC_CNTL_SWD_AUTO_FEED_EN to 1, SWD controller can also feed SWD itself without any interaction with CPU.

After reset:

- Check RTC_CNTL_RESET_CAUSE_PROCPU[5:0] for the cause of CPU reset.
  If RTC_CNTL_RESET_CAUSE_PROCPU[5:0] == 0x12, it indicates that the cause is SWD reset.
- Set RTC_CNTL_SWD_RST_FLAG_CLR to clear the SWD reset flag.

## 12.4 Interrupts

For watchdog timer interrupts, please refer to Section 11.2.6 *Interrupts* in Chapter 11 *Timer Group (TIMG)*.

## 12.5 Registers

MWDT registers are part of the timer submodule and are described in Section 11.4 *Register Summary* in Chapter 11 *Timer Group (TIMG)*. RWDT and SWD registers are part of the RTC submodule and are described in Section 9.7 *Register Summary* in Chapter 9 *Low-power Management*.

# Chapter 13

# XTAL32K Watchdog Timers (XTWDT)

## 13.1 Overview

The XTAL32K watchdog timer on ESP32-C3 is used to monitor the status of external crystal XTAL32K_CLK. This watchdog timer can detect the oscillation failure of XTAL32K_CLK, change the clock source of RTC, etc. When XTAL32K_CLK works as the clock source of RTC_SLOW_CLK (for clock description, see Chapter 6 *Reset and Clock*) and stops vibrating, the XTAL32K watchdog timer first switches to BACKUP32K_CLK derived from RC_SLOW_CLK and generates an interrupt (if the chip is in Light-sleep and Deep-sleep mode, the CPU will be woken up), and then switches back to XTAL32K_CLK after it is restarted by software.

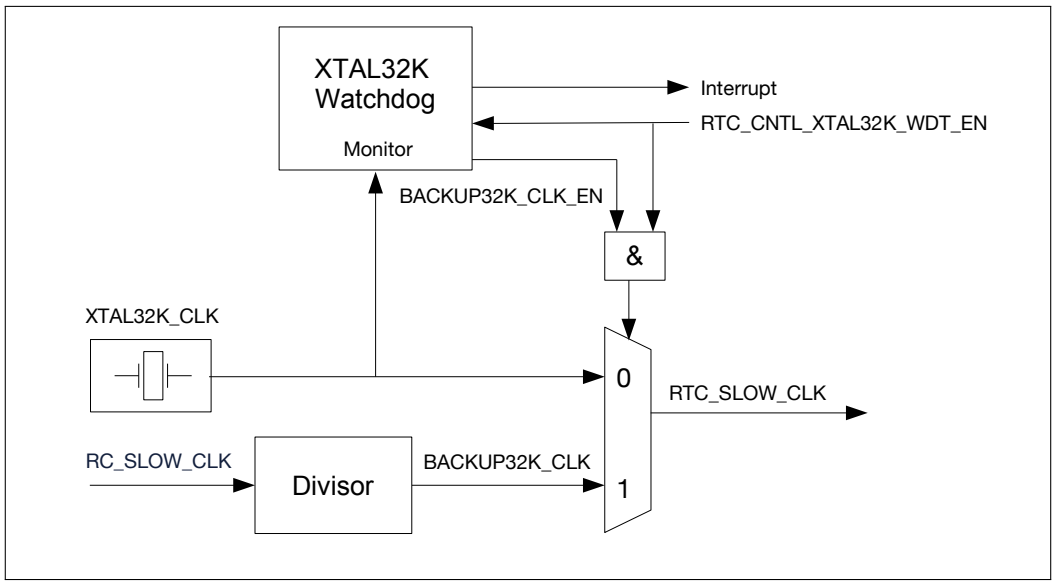



Figure 13.1-1. XTAL32K Watchdog Timer

## 13.2 Features

### 13.2.1 Interrupt and Wake-Up

When the XTAL32K watchdog timer detects the oscillation failure of XTAL32K_CLK, an oscillation failure interrupt RTC_XTAL32K_DEAD_INT (for interrupt description, please refer to Chapter 9 *Low-power Management*) is generated. At this point, the CPU will be woken up if in Light-sleep and Deep-sleep mode.

### 13.2.2 BACKUP32K_CLK

Once the XTAL32K watchdog timer detects the oscillation failure of XTAL32K_CLK, it replaces XTAL32K_CLK with BACKUP32K_CLK (with a frequency of 32 kHz or so) derived from RC_SLOW_CLK as RTC_SLOW_CLK, so as to ensure proper functioning of the system.

## 13.3 Functional Description

### 13.3.1 Workflow

1. The XTAL32K watchdog timer starts counting when RTC_CNTL_XTAL32K_WDT_EN is enabled. The counter based on RC_SLOW_CLK keeps counting until it detects the positive edge of XTAL_32K and is then cleared. When the counter reaches RTC_CNTL_XTAL32K_WDT_TIMEOUT, it generates an interrupt or a wake-up signal and is then reset.
2. If RTC_CNTL_XTAL32K_AUTO_BACKUP is set and step 1 is finished, the XTAL32K watchdog timer will automatically enable BACKUP32K_CLK as the alternative clock source of RTC_SLOW_CLK, to ensure the system's proper functioning and the accuracy of timers running on RTC_SLOW_CLK (e.g. RTC_TIMER). For information about clock frequency configuration, please refer to Section 13.3.2.
3. Software restarts XTAL32K_CLK by turning its XPD (meaning no power-down) signal off and on again via the RTC_CNTL_XPD_XTAL_32K bit. Then, the XTAL32K watchdog timer switches back to XTAL32K_CLK as the clock source of RTC_SLOW_CLK by clearing RTC_CNTL_XTAL32K_WDT_EN (BACKUP32K_CLK_EN is also automatically cleared). If the chip is in Light-sleep and Deep-sleep mode, the XTAL32K watchdog timer will wake up the CPU to finish the above steps.

### 13.3.2 BACKUP32K_CLK Working Principle

Chips have different RC_SLOW_CLK frequencies due to production process variations. To ensure the accuracy of RTC_TIMER and other timers running on RTC_SLOW_CLK when BACKUP32K_CLK is at work, the divisor of BACKUP32K_CLK should be configured according to the actual frequency of RC_SLOW_CLK (see details in Chapter 9 *Low-power Management*) via the RTC_CNTL_XTAL32K_CLK_FACTOR_REG register. Each byte in this register corresponds to a divisor component ($x_0$ ~$x_7$). BACKUP32K_CLK is divided by a fraction where the denominator is always 4, as calculated below.

$$f_back_clk/4 = f_rc_slow_clk/S$$
$$S = x_0 + x_1 + ... + x_7$$

f_back_clk is the desired frequency of BACKUP32K_CLK, i.e. 32.768 kHz; f_rc_slow_clk is the actual frequency of RC_SLOW_CLK; $x_0$ ~$x_7$ correspond to the pulse width in high and low state of four BACKUP32K_CLK clock signals (unit: RC_SLOW_CLK clock cycle).

### 13.3.3 Configuring the Divisor Component of BACKUP32K_CLK

Based on principles described in Section 13.3.2, configure the divisor component as follows:

- Calculate the sum of divisor components S according to the frequency of RC_SLOW_CLK and the desired frequency of BACKUP32K_CLK;
- Calculate the integer part of divisor $N = f_rc_slow_clk/f_back_clk$;
- Calculate the integer part of divisor component $M = N/2$. The integer part of divisor N are separated into two parts because a divisor component corresponds to a pulse width in high or low state;
- Calculate the number of divisor components that equal M ($x_n$ = M) and the number of divisor components that equal M + 1 ($x_n$ = M + 1) according to the value of M and S. (M + 1) is the fractional part of divisor component.

For example, if the frequency of RC_SLOW_CLK is 163 kHz, then $f_rc_slow_clk = 163000$, $f_back_clk = 32768$, $S = 20$, $M = 2$, and $\{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\} = \{2, 3, 2, 3, 2, 3, 2, 3\}$. As a result, the frequency of BACKUP32K_CLK is 32.6 kHz.

# Chapter 14

# Permission Control (PMS)

## 14.1 Overview

ESP32-C3 includes a Permission Controller (PMS), which allocates the hardware resources (memory and peripherals) to two isolated environments, thereby realizing the separation of privileged and unprivileged environments.

- Privileged Environment:
  - Can access all peripherals and memories;
  - Performs all confidential operations, such as user authentication, secure communication, and data encryption and decryption, etc.
- Unprivileged Environment:
  - Can access some peripherals and memories;
  - Performs other operations, such as user operation and different applications, etc.

Besides, ESP32-C3's RISC-V CPU also has a Physical Memory Protection (PMP) unit, which can be used by software to set memory access privileges (read, write, and execute permissions) for required memory regions. However, the PMP unit has some limitations:

- Only supports up to 16 configurable PMP regions, which sometimes are not enough to fully support the access management requirement of ESP32-C3's rich peripherals and different types of memories.
- Can only control CPU, not GDMA.

To this, ESP32-C3 has specially implemented this Permission Controller to complete the Physical Memory Protection unit.

ESP32-C3's completed workflow of permission check can be described below (also see Figure 14.1-1):

1. Check PMP permission
   - Pass: then continue and further check PMS permission
   - Fail: throw an exception and will not further check PMS permission
2. Check PMS permission
   - Pass: access allowed
   - Fail: trigger an interrupt

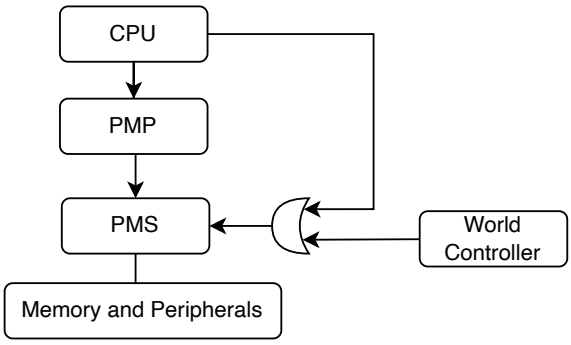



Figure 14.1-1. Permission Control Overview

For details about PMP, please refer to Section 1.8.1 in Chapter 1 *ESP-RISC-V CPU*. For details about World Controller, please refer to Chapter 15 *World Controller (WCL)*. This chapter only describes ESP32-C3's PMS mechanism.

## 14.2 Features

ESP32-C3's extended permission control mechanism supports:

- Independent access management in a privileged environment and unprivileged environment
- Independent access management to internal memory, including
  - CPU access to internal memory
  - GDMA access to internal memory
- Independent access management to external memory, including
  - CPU to external memory via SPI1
  - CPU to external memory via Cache
- Independent access management to peripheral regions, including
  - CPU access to peripheral regions
  - Interrupt upon unsupported access alignment
- Address splitting for more flexible access management
- Register locks to secure the integrity of access management related registers
- Interrupt upon unauthorized access

## 14.3 Privileged Environment and Unprivileged Environment

During PMS check, ESP32-C3 chip:

- When in the privileged environment: check the permission configuration registers for the privileged environment

- When not in the unprivileged environment: check the permission configuration registers for the unprivileged environment

Users can choose either of these two ways below to enter the chip into privileged environment:

- By configuring the world controller
  - Switching to Secure world: entering the privileged environment
  - Switching to Non-secure world: entering the unprivileged environment
- By configuring the privileged level of the 32-bit RISC-V CPU:
  - Switching to Machine mode: entering the privileged environment
  - Switching to User mode: entering the unprivileged environment

Users can configure PMS_PRIVILEGE_MODE_SEL to choose between the above-mentioned two ways to enter the chip into privileged environment:

- 0 (Default): via configuring the world controller. See details in Chapter 15 *World Controller (WCL)*.
- 1: via configuring the CP's privileged level. See details in Chapter 1 *ESP-RISC-V CPU*.

The following sections introduce how to configure the permission to different areas in the privileged environment and the unprivileged environment.

## 14.4 Internal Memory

ESP32-C3 has the following types of internal memory:

- ROM: 384 KB in total, including 256 KB Internal ROM0 and 128 KB Internal ROM1
- SRAM: 400 KB in total, including 16 KB Internal SRAM0 and 384 KB Internal SRAM1
- RTC FAST Memory: 8 KB in total, which can be further split into two regions each with independent permission configuration

This section describes how to configure the permission to each type of ESP32-C3's internal memory.

### 14.4.1 ROM

ESP32-C3's ROM can be accessed by CPU's instruction bus (IBUS) and data bus (DBUS) when configured. The ROM ranges accessible for IBUS and DBUS respectively are listed in Table 14.4-1.

**Table 14.4-1. ROM Address**

| ROM | IBUS Address | | DBUS Address | |
|---|---|---|---|---|
| | Starting Address | Ending Address | Starting Address | Ending Address |
| Internal ROM0 | 0x4000_0000 | 0x4003_FFFF | - | - |
| Internal ROM1 | 0x4004_0000 | 0x4005_FFFF | 0x3FF0_0000 | 0x3FF1_FFFF |

ESP32-C3 uses the registers listed in Table 14.4-2 to configure the instruction execution (X), write (W) and read (R) accesses of CPU's IBUS and DBUS, in User mode and Machine mode. Note that access configuration to ROM0 and ROM1 cannot be configured separately:

Table 14.4-2. Access Configuration to ROM (ROM0 and ROM1)

| Bus | Environment | Configuration Registers[A] | Access |
|---|---|---|---|
| IBUS | Privileged | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG [20:18][B] | X/W/R |
| | Unprivileged | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG [20:18][B] | X/W/R |
| DBUS | Privileged | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG [25:24][C] | W/R |
| | Unprivileged | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG [27:26] | W/R |

[A] 1: with access; 0: without access

[B] For example, configuring this field to 0b101 indicates CPU's IBUS is granted instruction execution and read accesses but not write access to ROM in the unprivileged environment.

[C] For example, configuring this field to 0b01 indicates CPU's DBUS is granted read access but not write access to ROM in privileged environment.

## 14.4.2 SRAM

ESP32-C3's SRAM can be accessed by CPU's instruction bus (IBUS) and data bus (DBUS) when configured. The SRAM address ranges accessible for IBUS and DBUS respectively are listed in Table 14.4-3.

Table 14.4-3. SRAM Address

| SRAM | Block | IBUS Address | | DBUS Address | |
|---|---|---|---|---|---|
| | | Starting Address | Ending Address | Starting Address | Ending Address |
| Internal SRAM0 | - | 0x4037_C000 | 0x4037_FFFF | - | - |
| Internal SRAM1 | Block0 | 0x4038_0000 | 0x4039_FFFF | 0x3FC8_0000 | 0x3FC9_FFFF |
| | Block1 | 0x403A_0000 | 0x403B_FFFF | 0x3FCA_0000 | 0x3FCB_FFFF |
| | Block2 | 0x403C_0000 | 0x403D_FFFF | 0x3FCC_0000 | 0x3FCD_FFFF |

Here, we will first introduce how to configure the permission to Internal SRAM0 and then Internal SRAM1.

### 14.4.2.1 Internal SRAM0 Access Configuration

ESP32-C3's Internal SRAM0 can be allocated to either CPU or ICACHE.

Users can configure PMS_INTERNAL_SRAM_USAGE_CPU_CACHE to allocate ESP32-C3's Internal SRAM0 to either CPU or ICACHE:

- 1: CPU
- 0: ICACHE

When the Internal SRAM0 is allocated to CPU, ESP32-C3 uses the registers listed in Table 14.4-4 to configure the instruction execution (X), write (W) and read (R) accesses of CPU's IBUS, in the privileged environment and the unprivileged environment:

Table 14.4-4. Access Configuration to Internal SRAM0

| Bus[A] | Environment | Configuration Registers[B] | Access |
|---|---|---|---|
| IBUS | Privileged | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_CACHEDATAARRAY_PMS_0[C] | X/W/R |
| | Unprivileged | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_CACHEDATAARRAY_PMS_0 | X/W/R |

[A] To access the Internal SRAM0, CPU must be configured with both the usage permission and respective access permission.

[B] 1: with access; 0: without access

[C] For example, configuring this field to 0b101 indicates CPU's IBUS is granted with instruction execution and read accesses but not write access to SRAM0 in the privileged environment.

### 14.4.2.2 Internal SRAM1 Access Configuration

ESP32-C3's Internal SRAM1 includes Block0 ~ Block2 (see details in Table 14.4-3) and can be:

- Accessed by CPU's DBUS, IBUS and GDMA at the same time
- Further split into up to 6 regions with independent access management for more flexible permission control.

ESP32-C3's Internal SRAM1 can be further split into up to 6 regions with 5 split lines. Users can configure different access to each region independently.

To be more specific, the Internal SRAM1 can be first split into Instruction Region and Data Region by IRam0_DRam0_split_line:

- Instruction Region:
  - Then the Instruction Region should be only configured to be accessed by IBUS;
  - And can be further split into three split regions by IRam0_split_line_0 and IRam0_split_line_1.
- Data Region:
  - The Data Region should be only configured to be accessed by DBUS;
  - And can be further split into three split regions by DRam0_split_line_0 and DRam0_split_line_1.

See illustration in Figure 14.4-1 and Table 14.4-5 below.

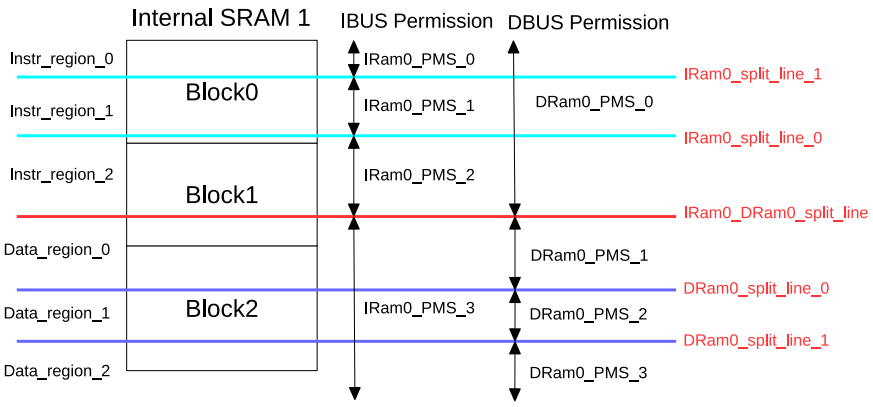



Figure 14.4-1. Split Lines for Internal SRAM1

Table 14.4-5. Internal SRAM1 Split Regions

| Internal Memory [A] | Instruction / Data Regions | Split Regions[B] |
|---|---|---|
| SRAM1 | Instruction Region | Instr_Region_0 |
| | | Instr_Region_1 |
| | | Instr_Region_2 |
| | Data Region | Data_Region_0 |
| | | Data_Region_1 |
| | | Data_Region_2 |

[A] Access to each split region can be configured independently. See details in Table 14.4-6 and 14.4-7.
[B] See the description below on how to configure the split lines.

### Internal SRAM1 Split Regions

ESP32-C3 allows users to configure the split lines to their needs with registers below:

- Split line to split the Instruction and Data regions (IRam0_DRam0_split_line):
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_1_REG
- The first split line to further split the Instruction Region (IRam0_split_line_0):
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_2_REG
- The second split line to further split the Instruction Region (IRam0_split_line_1):
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_3_REG
- The first split line to further split the data Region (DRam0_split_line_0):
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_4_REG
- The second split line to further split the data Region (DRam0_split_line_1):
  - PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_5_REG

When configuring the split lines,

1. First configure the block in which the split line is by:
   - Configuring the Category_*x* field for the block in which the split line is to 0x1 or 0x2 (no difference)
   - Configuring the Category_0 ~ Category_*x-1* fields for all the preceding blocks to 0x0
   - Configuring the Category_*x+1* ~ Category_2 fields for all blocks afterwards to 0x3

   For example, assuming you want to configure the split line in Block1, then first configure the Category_1 field for Block1 to 0x1 or 0x2; configure the Category_0 for Block0 to 0x0; and configure the Category_2 for Block2 to 0x3 (see illustration in Figure 14.4-2). On the other hand, when reading 0x1 or 0x2 from Category_1, then you know the split line is in Block1.
2. Configure the position of the split line inside the configured block by:
   - Writing the [16:9] bits of the actual address at which you want to split the memory to the SPLITADDR field for the block in which the split line is.
   - Note that the split address must be aligned to 512 bytes, meaning you can only write the integral multiples of 0x200 to the SPLITADDR field.

For example, if you want to split the instruction region at 0x3fc88000, then write the [16:9] bits of this address, which is 0b01000000, to SPLITADDR.

3. The split address applies to both IBUS and DBUS address. For example, DBUS address 0x3fc88000 and IBUS address 0x40388000 indicate the same location in SRAM1. The split address for both buses is [16:9].

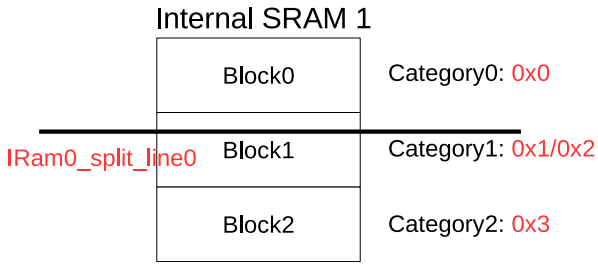



**Figure 14.4-2. An illustration of Configuring the Category fields**

Note the following points when configuring the split lines:

- Position:
  - The split line that splitting the Instruction Region and Data Region can be configured anywhere inside Internal SRAM1.
  - The two split lines further splitting the Instruction Region into 3 split regions must stay inside the Instruction Region.
  - The two split lines further splitting the Data Region into 3 split regions must stay inside the Data Region.
- Spilt lines can overlap with each other. For example,
  - When the two split lines inside the Data Region are not overlapping with each other, then the Data Region is split into 3 split regions
  - When the two split lines inside the Data Region are overlapping with each other, then the Data Region is only split into 2 split regions
  - When the two split lines inside the Data Region are not only overlapping with each other but also with the split line that splits the Data Region and the Instruction Region, then the Data Region is not split at all and only has one region.

#### Access Configuration

After configuring the split lines, users can then use the registers described in the Table 14.4-6 and Table 14.4-7 below to configure the access of CPU's IBUS, DBUS and GDMA peripherals, in the privileged environment and the unprivileged environment, to these split regions independently.

Table 14.4-6. Access Configuration to the Instruction Region of Internal SRAM1

| Buses | Environment | Configuration Registers | Instruction Region | | | Access |
|---|---|---|---|---|---|---|
| | | | instr_region_0 | instr_region_1 | instr_region_2 | |
| IBUS | Privileged | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG | [2:0] | [5:3] | [8:6] | X/W/R |
| | Unprivileged | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG | [2:0] | [5:3] | [8:6] | X/W/R |
| DBUS | Privileged | PMS_Core_X_DRAM0_PMS_CONSTRAIN_1_REG | [13:12][A] | | | W/R |
| | Unprivileged | | [1:0][A] | | | W/R |
| GDMA [C] | *XX* Peripherals [D] | PMS_DMA_APBPERI_*XX*_PMS_CONSTRAIN_1_REG | [1:0][B] | | | W/R |

[A] Configure DBUS' access to the Instruction Region. However, it's recommended to configure these bits to 0.
[B] Configure GDMA's access to the Instruction Region. However, it's recommended to configure these bits to 0.
[C] GMDA doesn't support the privileged environment and unprivileged environment.
[D] ESP32-C3 has 6 peripherals, including SPI2, UCHIO, I2S, AES, SHA, ADC, which can access Internal SRAM1 via GDMA. Each peripherals can be configured with different access to the Internal SRAM1 independently.

Table 14.4-7. Access Configuration to the Data Region of Internal SRAM1

| Buses | Environment | Configuration Registers | Data Region | | | Access |
|---|---|---|---|---|---|---|
| | | | data_region_0 | data_region_1 | data_region_2 | |
| IBUS | Privileged | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG | [11:9][A] | | | X/W/R |
| | Unprivileged | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG | [11:9][A] | | | X/W/R |
| DBUS | Privileged | PMS_Core_X_DRAM0_PMS_CONSTRAIN_1_REG | [3:2] | [5:4] | [7:6] | W/R |
| | Unprivileged | | [15:14] | [17:16] | [19:18] | W/R |
| GDMA [B] | *XX* Peripherals[C] | PMS_DMA_APBPERI_*XX*_PMS_CONSTRAIN_1_REG | [3:2] | [5:4] | [7:6] | W/R |

[A] Configure IBUS' access to the Data Region. However, it's recommended to configure these bits to 0.
[B] GMDA doesn't support the privileged environment and the unprivileged environment.
[C] ESP32-C3 has six peripherals, including SPI2, UCHIO, I2S, AES, SHA, ADC, which can access Internal SRAM1 via GDMA. Each peripheral can be configured with different access to the Internal SRAM1 independently.

For details on how to configure the split lines, see Section 14.4.2.2.

**Note:**
If enabled, the permission control module watches all the memory access and fires the panic handler if a permission violation is detected. This feature automatically splits the SRAM memory into data and instruction segments and sets Read/Execute permissions for the instruction part (below given splitting address) and Read/Write permissions for the data part (above the splitting address). The memory protection is effective on all access through the IRAM0 and DRAM0 buses. See details, see *ESP-IDF api-reference Memory protection*.

### 14.4.3 RTC FAST Memory

ESP32-C3's RTC FAST Memory is 8 KB. See the address of RTC FAST Memory below:

Table 14.4-8. RTC FAST Memory Address

| Memory | Starting Address | Ending Address |
|---|---|---|

| | | |
|---|---|---|
| RTC FAST Memory | 0x5000_0000 | 0x5000_1FFF |

ESP32-C3's RTC FAST Memory can be further split into 2 regions. Each split region can be configured independently with different access.

The Register for configuring the split line is described below:

**Table 14.4-9. Split RTC FAST Memory into the Higher Region and the Lower Region**

| | Privileged Environment | Unprivileged Environment |
|---|---|---|
| Split Lines Configuration Register[1] | PIF_PMS_CONSTRAN_9_REG [10:0] | PIF_PMS_CONSTRAN_9_REG [21:11] |

[1] The word offset from the RTC FAST Memory base address should be used when configuring the split address. For example, if you want to split the RTC FAST Memory at 0x5000_1000, then write 0x400 (0x1000>>2) to this register.

Access configuration for the higher and lower regions of the RTC FAST Memory is described below:

**Table 14.4-10. Access Configuration to the RTC FAST Memory**

| Bus | RTC FAST Memory | Configuration Registers | | Access[A] |
|---|---|---|---|---|
| | | Privileged Environment | Unprivileged Environment | |
| Peri Bus (PIF) | Higher Region | PIF_PMS_CONSTRAN_10_REG [5:3] [B] | PIF_PMS_CONSTRAN_10_REG [11:9] | X/R/W |
| | Lower Region | PIF_PMS_CONSTRAN_10_REG [2:0] | PIF_PMS_CONSTRAN_10_REG [8:6] | |

[A] 1: with access; 0: without access

[B] For example, configuring this field to 0b101 indicates CPU's peripheral (PIF) bus is granted with the instruction execution and read accesses but not the read access in the privileged environment to the higher region of RTC FAST Memory.

## 14.5 Peripherals

### 14.5.1 Access Configuration

ESP32-C3's CPU can be configured with different read (R) and write (W) accesses to most of its modules and peripherals independently, in the privileged environment and in the unprivileged environment, by configuring respective registers
(PMS_CORE_0_PIF_PMS_CONSTRAN_*n*_REG).

Notes on PMS_CORE_0_PIF_PMS_CONSTRAN_*n*_REG:

- *n* can be 1 ~ 8, in which 1 ~ 4 are for the privileged environment and 5 ~ 8 are for the unprivileged environment.

For example, users can configure PMS_CORE_0_PIF_PMS_CONSTRAIN_1_REG [1:0] to 0x2, meaning CPU is granted with read access but not write access in the privileged environment to UART0. In this case, CPU won't be able to modify the UART0's internal registers when in the privileged environment.

Table 14.5-1. Access Configuration of the Peripherals

| Peripherals | Privileged Environment | Unprivileged Environment | Bit[3] |
|---|---|---|---|
| GDMA | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [7:6] |
| eFuse Controller & PMU[1] | **_PMS_CONSTRAN_1_REG | **_PMS_CONSTRAN_5_REG | [15:14] |
| IO_MUX | **_PMS_CONSTRAN_1_REG | **_PMS_CONSTRAN_5_REG | [17:16] |
| GPIO | **_PMS_CONSTRAN_1_REG | **_PMS_CONSTRAN_5_REG | [7:6] |
| Interrupt Matrix | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [21:20] |
| System Timer | **_PMS_CONSTRAN_2_REG | **_PMS_CONSTRAN_6_REG | [31:30] |
| Timer Group 0 | **_PMS_CONSTRAN_2_REG | **_PMS_CONSTRAN_6_REG | [27:26] |
| Timer Group 1 | **_PMS_CONSTRAN_2_REG | **_PMS_CONSTRAN_6_REG | [29:28] |
| System Registers | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [17:16] |
| PMS Registers | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [19:18] |
| Debug Assist | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [27:26] |
| Accelerators[2] | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [5:4] |
| Cache & XTS_AES[1] | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [25:25] |
| UART 0 | **_PMS_CONSTRAN_1_REG | **_PMS_CONSTRAN_5_REG | [1:0] |
| UART 1 | **_PMS_CONSTRAN_1_REG | **_PMS_CONSTRAN_5_REG | [31:30] |
| SPI 0 | **_PMS_CONSTRAN_1_REG | **_PMS_CONSTRAN_5_REG | [5:4] |
| SPI 1 | **_PMS_CONSTRAN_1_REG | **_PMS_CONSTRAN_5_REG | [3:2] |
| SPI 2 | **_PMS_CONSTRAN_3_REG | **_PMS_CONSTRAN_7_REG | [1:0] |
| I2C 0 | **_PMS_CONSTRAN_2_REG | **_PMS_CONSTRAN_6_REG | [5:4] |
| I2S | **_PMS_CONSTRAN_3_REG | **_PMS_CONSTRAN_7_REG | [15:14] |
| USB OTG Core | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [15:14] |
| Two-wire Automotive Interface | **_PMS_CONSTRAN_3_REG | **_PMS_CONSTRAN_7_REG | [11:10] |
| UHCI 0 | **_PMS_CONSTRAN_2_REG | **_PMS_CONSTRAN_6_REG | [7:6] |
| LED PWM Controller | **_PMS_CONSTRAN_2_REG | **_PMS_CONSTRAN_6_REG | [17:16] |
| Remote Control Peripheral | **_PMS_CONSTRAN_2_REG | **_PMS_CONSTRAN_6_REG | [11:10] |
| APB Controller | **_PMS_CONSTRAN_3_REG | **_PMS_CONSTRAN_7_REG | [5:4] |
| ADC Controller | **_PMS_CONSTRAN_4_REG | **_PMS_CONSTRAN_8_REG | [9:8] |

[1] : This is shared by more than one peripherals.

[2] : Accelerators: AES, SHA, RSA, Digital Signatures, HMAC

[3] : Access: R/W

[4] : ** in the table replaces PMS_CORE_0_PIF.

### 14.5.2 Split Peripheral Regions into Split Regions

On top of what described in the previous section, user can select one of ESP32-C3's peripheral region to split them into 7 regions (from Peri Region0 ~ Peri Region7) for more flexible permission control.

For example, the registers for ESP32-C3's GDMA controller are allocated as:

- 3 sets of registers for each of 3 RX channel
- 3 sets of registers for each of 3 TX channel
- 1 set of registers for configuration

As seen above, GDMA's peripheral region is divided into 7 split regions (implemented in hardware), which can be configured with different permission independently, thus achieving independent permission control for each GDMA channel.

Users can configure CPU's read (R) and write (W) accesses to a specific split region (Peri Region*n*) in the privileged environment and in the unprivileged environment by configuring PMS_REGION_PMS_CONSTRAN_*n*_REG.

Notes on PMS_REGION_PMS_CONSTRAN_*n*_REG:

- *n* can be 1~10, in which
  - PMS_REGION_PMS_CONSTRAN_1_REG is for configuring CPU's permission in the privileged environment.
  - PMS_REGION_PMS_CONSTRAN_2_REG is for configuring CPU's permission in the unprivileged environment.
  - PMS_REGION_PMS_CONSTRAN_*n*_REG (*n* = 3~10) are used to configuring the starting addresses for each Peri Regions. Note the starting address of each Peri Region is also the ending address of the previous Peri Region.

**Table 14.5-2. Access Configuration of Peri Regions**

| Peri Regions | Starting Address Configuration | Access Configuration | |
|---|---|---|---|
| | | Privileged Environment | Unprivileged Environment |
| Peri Region0 | PMS_REGION_PMS_CONSTRAN_3_REG | PMS_REGION_PMS_CONSTRAN_1_REG [1:0] | PMS_REGION_PMS_CONSTRAN_2_REG [1:0] |
| Peri Region1 | PMS_REGION_PMS_CONSTRAN_4_REG | PMS_REGION_PMS_CONSTRAN_1_REG [3:2] | PMS_REGION_PMS_CONSTRAN_2_REG [3:2] |
| Peri Region2 | PMS_REGION_PMS_CONSTRAN_5_REG | PMS_REGION_PMS_CONSTRAN_1_REG [5:4] | PMS_REGION_PMS_CONSTRAN_2_REG [5:4] |
| Peri Region3 | PMS_REGION_PMS_CONSTRAN_6_REG | PMS_REGION_PMS_CONSTRAN_1_REG [7:6] | PMS_REGION_PMS_CONSTRAN_2_REG [7:6] |
| Peri Region4 | PMS_REGION_PMS_CONSTRAN_7_REG | PMS_REGION_PMS_CONSTRAN_1_REG [9:8] | PMS_REGION_PMS_CONSTRAN_2_REG [9:8] |
| Peri Region5 | PMS_REGION_PMS_CONSTRAN_8_REG | PMS_REGION_PMS_CONSTRAN_1_REG [11:10] | PMS_REGION_PMS_CONSTRAN_2_REG [11:10] |
| Peri Region6* | PMS_REGION_PMS_CONSTRAN_9_REG | PMS_REGION_PMS_CONSTRAN_1_REG [13:12] | PMS_REGION_PMS_CONSTRAN_2_REG [13:12] |

* The ending address of Peri Region6 is configured by PMS_REGION_PMS_CONSTRAN_10_REG.

## 14.6 External Memory

ESP32-C3 can access the external memory via one of the two ways illustrated in Figure 14.6-1 below.

- CPU via SPI1
- CPU via CACHE

Where,

- Box 0 checks the SPI and Cache's access to external flash.
- Box 1 checks CPU's access to Cache.

### 14.6.1 SPI and Cache's Access to External Flash

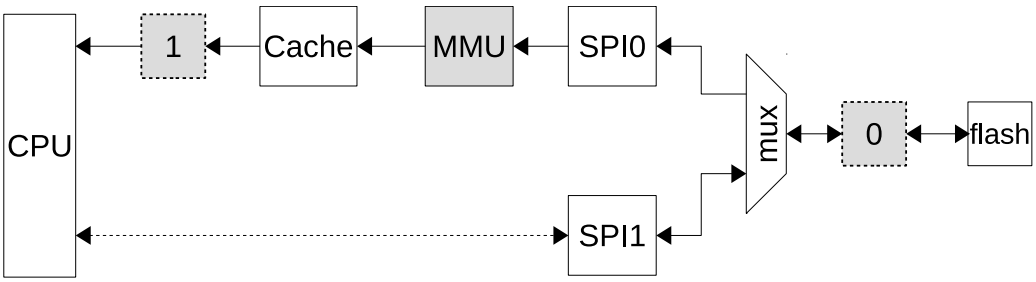



Figure 14.6-1. Two Ways to Access External Memory

### 14.6.1.1 Address

ESP32-C3's flash can be further split to achieve more flexible permission control. Each split region can be configured with different access independently.

- Flash can be split into 4 regions, the length of each should be the integral multiples of 64 KB.
- Also, the starting address of each region should also be aligned to 64 KB.

The following registers can be used to configure how the flash is split.

Table 14.6-1. Split the External Memory into Split Regions

| Split Regions | Split Region Configuration[3] | |
|---|---|---|
| | Starting Address[1] | Length[2] |
| Flash Region*n* (*n*: 0~3) | SYSCON_FLASH_ACE*n*_ADDR_REG | SYSCON_FLASH_ACE*n*_SIZE_REG |

[1] Configuring this field with the actual address, which should be aligned to 64 KB.
[2] When configuring the length of Region*n*, note the total length of all flash regions should be no greater than 16 MB, respectively.
[3] Each region cannot overlap with others.

### 14.6.1.2 Access Configuration

Each split region for flash can be configured with different permission independently via the register described in the table below.

Table 14.6-2. Access Configuration of Flash Regions

| Split Regions | Access Configuration | | |
|---|---|---|---|
| | Configuration Register | Cache | SPI |
| Flash Region *n* (*n*: 0 ~ 3) | SYSCON_FLASH_ACE*n*_ATTR | [1:0][A] | [3:2][B] |

[A] These bits are configured in order R/X. For example, configuring this field to 2'b10 indicates CACHE is granted with the read access but no instruction execution access to the Flash Region *n*.
[B] These bits are configured in order W/R. For example, configuring this field to 2'b01 indicates SPI is granted with the read access but no write access to the Flash Region *n*.

### 14.6.2 CPU's Access to Cache

ESP32-C3's CPU access Cache using a virtual address. The memory space in ESP32-C3 that is accessible for CPU to access Cache is called "Virtual Address Region", which can be seen in Table 14.6-3 below.

Table 14.6-3. Cache Virtual Address Region

| Bus Type | Virtual Address Region | | Size (MB) | Target |
|---|---|---|---|---|
| | Starting Address | Ending Address | | |
| DBus (read-only) | 0x3C00_0000 | 0x3C7F_FFFF | 8 | Uniform Cache |
| IBus | 0x4200_0000 | 0x427F_FFFF | 8 | Uniform Cache |

#### 14.6.2.1 Split Regions

Both ESP32-C3's DBUS and IBUS Cache virtual address regions can be further split into up to 4 regions. Users can configure different access to each region independently.

Table 14.6-4. Split IBUS Cache Virtual Address into 4 Regions

| Split Regions[1] | Split Region Configuration | |
|---|---|---|
| | Starting Address | Ending Address |
| IBUS Region0 | 0x4200_0000 | EXTMEM_IBUS_PMS_TBL_BOUNDARY0_REG[2] |
| IBUS Region1 | EXTMEM_IBUS_PMS_TBL_BOUNDARY0_REG[2] | EXTMEM_IBUS_PMS_TBL_BOUNDARY1_REG[2] |
| IBUS Region2 | EXTMEM_IBUS_PMS_TBL_BOUNDARY1_REG[2] | EXTMEM_IBUS_PMS_TBL_BOUNDARY2_REG[2] |
| IBUS Region3 | EXTMEM_IBUS_PMS_TBL_BOUNDARY2_REG[2] | 0x4280_0000 |

[1] The address range of each split region is [Starting Address, Ending Address).

[2] The address represented is "0x4200_0000 + 0x1000 * EXTMEM_IBUS_PMS_TBL_BOUNDARY*n*_REG". For example, when EXTMEM_IBUS_PMS_TBL_BOUNDARY0_REG is configured to 2, then the address range of IBUS Region0 is [0x4200_0000, 0x4200_2000).

Table 14.6-5. Split DBUS Cache Virtual Address into 4 Regions

| Split Regions[1] | Split Region Configuration | |
|---|---|---|
| | Starting Address | Ending Address |
| DBUS Region0 | 0x3C00_0000 | EXTMEM_DBUS_PMS_TBL_BOUNDARY0_REG[2] |
| DBUS Region1 | EXTMEM_DBUS_PMS_TBL_BOUNDARY0_REG[2] | EXTMEM_DBUS_PMS_TBL_BOUNDARY1_REG[2] |
| DBUS Region2 | EXTMEM_DBUS_PMS_TBL_BOUNDARY1_REG[2] | EXTMEM_DBUS_PMS_TBL_BOUNDARY2_REG[2] |
| DBUS Region3 | EXTMEM_DBUS_PMS_TBL_BOUNDARY2_REG[2] | 0x3C80_0000 |

[1] The address range of each split region is [Starting Address, Ending Address).

[2] The address represented is "0x3C00_0000 + 0x0100 * EXTMEM_DBUS_PMS_TBL_BOUNDARY*n*_REG". For example, when EXTMEM_DBUS_PMS_TBL_BOUNDARY*n*_REG is configured to 2, then the address range of IBUS Split Region0 is [0x3C00_0000, 0x3C00_0200]).

### 14.6.3 Access Configuration

Each Cache split region can be configured with different permission independently via registers described in Table 14.6-6 and Table 14.6-7 below.

Table 14.6-6. Access Configuration of IBUS to Split Regions

| Split Regions | Access Configuration | | |
|---|---|---|---|
| | Configuration Register | Privileged[A] | Unprivileged[A] |
| IBUS Region0[C] | - | - | - |
| IBUS Region1 | EXTMEM_IBUS_PMS_TBL_ATTR_REG | [1:0][B] | [3:2] |
| IBUS Region2 | EXTMEM_IBUS_PMS_TBL_ATTR_REG | [5:4] | [7:6] |
| IBUS Region3 [C] | - | - | - |

[A] These bits are configured in order R/X.

[B] For example, configuring this field to 2'b10 indicates CPU's IBUS is granted read access but no instruction execution access to IBUS region1 in the privileged environment.

[C] IBUS is not allowed to access Region0 and Region3, thus cannot be configured. All attempts will be rejected.

Table 14.6-7. Access Configuration of DBUS to Split Regions

| Split Regions | Access Configuration | | |
|---|---|---|---|
| | Configuration Register | Privileged[A] | Unprivileged[A] |
| DBUS Region0[C] | - | - | - |
| DBUS Region1 | EXTMEM_DBUS_PMS_TBL_ATTR_REG | [0][B] | [1] |
| DBUS Region2 | EXTMEM_DBUS_PMS_TBL_ATTR_REG | [2] | [3] |
| DBUS Region3[C] | - | - | - |

[A] Only the read access can be configured.

[B] For example, configuring this field to 1'b1 indicates CPU's DBUS is granted read access to DBUS region1 in the privileged environment.

[C] DBUS is not allowed to access Region0 and Region3, thus cannot be configured. All attempts will be rejected.

## 14.7 Unauthorized Access and Interrupts

Any attempt to access ESP32-C3's slave device without configured permission is considered an **unauthorized access** and will be handled as described below:

- This attempt will only be responded with default values, in particular,
  - All instruction execution or read attempts will be responded with 0 (for internal memory and peripheral) or 0xdeadbeaf (for external memory)
  - All write attempts will fail
- An interrupt will be triggered (when enabled). See details below.

Note that only the information of the first interrupt is logged. Therefore, it's advised to handle interrupt signals and clear interrupts in-time, so the information of the next interrupt can be logged correctly.

### 14.7.1 Interrupt upon Unauthorized IBUS Access

ESP32-C3 can be configured to trigger interrupts when IBUS attempts to access internal ROM and SRAM without configured permission and log the information about this unauthorized access. Note that, once this interrupt is enabled, it's enabled for all internal ROM and SRAM memory, and cannot be only enabled for a certain address field. This interrupt corresponds to the PMS_IBUS_VIO_INTR interrupt source described in Table 8.3-1 from Chapter 8 *Interrupt Matrix (INTERRUPT)*.

**Table 14.7-1. Interrupt Registers for Unauthorized IBUS Access**

| Registers | Bit | Description |
|---|---|---|
| PMS_CORE_0_IRAM0_PMS_MONITOR_1_REG | [0] | Clears interrupt signal |
| | [1] | Enables interrupt |
| PMS_CORE_0_IRAM0_PMS_MONITOR_2_REG | [0] | Stores interrupt status of unauthorized IBUS access |
| | [1] | Stores the access direction. 1: write; 0: read. |
| | [2] | Stores the instruction direction. 1: load/store; 0: instruction execution. |
| | [4:3] | Stores the privileged mode the CPU was in when the unauthorized IBUS access happened. 0b01: privileged environment; 0b10: unprivileged environment |
| | [28:5] | Stores the address that CPU's IBUS was trying to access unauthorized. |

### 14.7.2 Interrupt upon Unauthorized DBUS Access

ESP32-C3 can be configured to trigger interrupts when DBUS attempts to access internal ROM and SRAM without configured permission and log the information about this unauthorized access. Note that, once this interrupt is enabled, it's enabled for all internal ROM and SRAM memory, and cannot be only enabled for a certain address field. This interrupt corresponds to the PMS_DBUS_VIO_INTR interrupt source described in Table 8.3-1 from Chapter 8 *Interrupt Matrix (INTERRUPT)*.

**Table 14.7-2. Interrupt Registers for Unauthorized DBUS Access**

| Registers | Bit | Description |
|---|---|---|
| PMS_CORE_0_DRAM0_PMS_MONITOR_1_REG | [0] | Clears interrupt signal |
| | [1] | Enables interrupt |
| PMS_CORE_0_DRAM0_PMS_MONITOR_2_REG | [0] | Stores interrupt status of unauthorized DBUS access |
| | [1] | Flags atomic access. 1: atomic access; 0: not atomic access. |
| | [3:2] | Stores the privileged mode the CPU was in when the unauthorized DBUS access happened. 0b01: privileged environment; 0b10: unprivileged environment |
| | [25:4] | Stores the address that CPU's DBUS was trying to access unauthorized. |
| PMS_CORE_0_DRAM0_PMS_MONITOR_3_REG | [0] | Stores the access direction. 1: write; 0: read. |
| | [25:4] | Stores the byte information of the unauthorized DBUS access. |

### 14.7.3 Interrupt upon Unauthorized Access to External Memory

ESP32-C3 can be configured to trigger Interrupt upon unauthorized access to external memory, and log the information about this unauthorized access. This interrupt corresponds to the SPI_MEM_REJECT_INTR interrupt source described in Table 8.3-1 from Chapter 8 *Interrupt Matrix (INTERRUPT)*.

Table 14.7-3. Interrupt Registers for Unauthorized Access to External Memory

| Registers | Bit | Description |
|---|---|---|
| SYSCON_SPI_MEM_PMS_CTRL_REG | [0] | Stores exception signal |
| | [1] | Clears exception signal and logged information |
| | [2] | Indicates unauthorized instruction execution |
| | [3] | Indicates unauthorized read |
| | [4] | Indicates unauthorized write |
| | [5] | Indicates overlapping split regions |
| | [6] | Indicates invalid address |

### 14.7.4 Interrupt upon Unauthorized Access to Internal Memory via GDMA

ESP32-C3 can be configured to trigger Interrupt upon unauthorized access to internal memory via GDMA, and log the information about this unauthorized access. This interrupt corresponds to the PMS_DMA_VIO_INTR interrupt source described in Table 8.3-1 from Chapter 8 *Interrupt Matrix (INTERRUPT)*.

Table 14.7-4. Interrupt Registers for Unauthorized Access to Internal Memory via GDMA

| Registers | Bit | Description |
|---|---|---|
| PMS_DMA_APBPERI_PMS_MONITOR_1_REG | [0] | Clears interrupt signal |
| | [1] | Enables interrupt |
| PMS_DMA_APBPERI_PMS_MONITOR_2_REG | [0] | Stores interrupt signal |
| | [2:1] | Stores the privileged mode the CPU was in when the unauthorized access happened. 0b01: privileged environment; 0b10: unprivileged environment |
| | [24:3] | Stores the address that GDMA was trying to access unauthorized |
| PMS_DMA_APBPERI_PMS_MONITOR_3_REG | [0] | Stores the access direction. 1: write; 0: read |
| | [16:1] | Stores the byte information of unauthorized access |

For information about Interrupt upon unauthorized access to external memory via GDMA, please refer to Chapter 2 *GDMA Controller (GDMA)*.

### 14.7.5 Interrupt upon Unauthorized peripheral bus (PIF) Access

ESP32-C3 can be configured to trigger interrupts when PIF attempts to access RTC FAST memory and peripheral regions without configured permission, and log the information about this unauthorized access. Note that, once this interrupt is enabled, it's enabled for all RTC FAST memory and peripheral regions, and

cannot be only enabled for a certain address field. This interrupt corresponds to the PMS_PERI_VIO_INTR interrupt source described in Table 8.3-1 from Chapter 8 *Interrupt Matrix (INTERRUPT)*.

**Table 14.7-5. Interrupt Registers for Unauthorized PIF Access**

| Registers | Bit | Description |
|---|---|---|
| PMS_CORE_0_PIF_PMS_MONITOR_1_REG | [1] | Enables interrupt |
| | [0] | Clears interrupt signal and logged information |
| PMS_CORE_0_PIF_PMS_MONITOR_2_REG | [7:6] | Stores the privileged mode the CPU was in when the unauthorized PIF access happened. 0b01: privileged environment; 0b10: unprivileged environment |
| | [5] | Stores the access direction. 1: write; 0: read |
| | [4:2] | Stores the data type of unauthorized access. 0: byte; 1: half-word; 2: word |
| | [1] | Stores the access type. 0: instruction; 1: data |
| | [0] | Stores the interrupt signal |
| PMS_CORE_0_PIF_PMS_MONITOR_3_REG | [31:0] | Stores the address of unauthorized access |

In particular, ESP32-C3 can also be configured to check the access alignment when PIF attempts to access the peripheral regions and trigger Interrupt upon unauthorized alignment. See the detailed description in the following section.

### 14.7.6 Interrupt upon Unauthorized PIF Access Alignment

Access to all of ESP32-C3's modules/peripherals is **word aligned**.

ESP32-C3 can be configured to check the access alignment to all modules/peripherals, and trigger Interrupt upon **non-word aligned access**.

This interrupt corresponds to the PMS_PERI_VIO_SIZE_INTR interrupt source described in Table 8.3-1 from Chapter 8 *Interrupt Matrix (INTERRUPT)*.

Note that CPU can convert some non-word aligned access to word aligned access, thus avoiding triggering alignment interrupt.

Table 14.7-6 below lists all the possible access alignments and their results (when the interrupt is enabled), in which:

- **INTR**: interrupt
- √: access succeeds and no interrupt.

**Table 14.7-6. All Possible Access Alignment and their Results**

| Accessed Address | Access Alignment | Read | Write |
|---|---|---|---|
| 0x0 | Byte aligned | √ | INTR |
| | Half-word aligned | √ | INTR |
| | Word aligned | √ | √ |
| 0x1 | Byte aligned | √ | INTR |
| | Half-word aligned | √ | INTR |

| Accessed Address | Access Alignment | Read | Write |
|---|---|---|---|
| | Word aligned | √ | INTR |
| 0x2 | Byte aligned | √ | INTR |
| | Half-word aligned | √ | INTR |
| | Word aligned | √ | INTR |
| 0x3 | Byte aligned | √ | INTR |
| | Half-word aligned | √ | INTR |
| | Word aligned | √ | INTR |

Table 14.7-7. Interrupt Registers for Unauthorized Access Alignment

| Registers | Bit | Description |
|---|---|---|
| PMS_CORE_0_PIF_PMS_MONITOR_4_REG | [1] | Enables interrupt |
| | [0] | Clears interrupt signal and logged information |
| PMS_CORE_0_PIF_PMS_MONITOR_5_REG | [4:3] | Stores the privileged mode the CPU was in when the unauthorized access happened. 0b01: privileged environment; 0b10: unprivileged environment |
| | [2:1] | Stores the unauthorized access type. 0: byte aligned; 1: half-word aligned; 2: word aligned |
| | [0] | Stores the interrupt status. 0: no interrupt; 1: interrupt |
| PMS_CORE_0_PIF_PMS_MONITOR_6_REG | [31:0] | Stores the address of the unauthorized access |

## 14.8 Register Locks

All ESP32-C3's permission control related registers can be locked by respective lock registers. When the lock registers are configured to 1, these registers themselves and their related permission control registers are all protected from modification until the next CPU reset.

Note that there isn't a one-to-one correspondence between the lock registers and permission control registers. See details in Table 14.8-1.

Table 14.8-1. Lock Registers and Related Permission Control Registers

| Lock Registers | Related Permission Control Registers |
|---|---|
| Lock privileged Mode Configuration | |
| PMS_PRIVILEGE_MODE_SEL_LOCK_REG | PMS_PRIVILEGE_MODE_SEL_LOCK_REG |
| | PMS_PRIVILEGE_MODE_SEL_REG |
| Lock Internal SRAM Usuage and Access Configuration | |
| PMS_INTERNAL_SRAM_USAGE_0_REG | PMS_INTERNAL_SRAM_USAGE_0_REG |
| | PMS_INTERNAL_SRAM_USAGE_1_REG |
| | PMS_INTERNAL_SRAM_USAGE_4_REG |
| PMS_CORE_X_IRAM0_PMS_CONSTRAIN_0_REG | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_0_REG |
| | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG |
| | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG |
| PMS_CORE_*m*_IRAM0_PMS_MONITOR_0_REG | PMS_CORE_*m*_IRAM0_PMS_MONITOR_0_REG |

| Lock Registers | Related Permission Control Registers |
|---|---|
| | PMS_CORE_*m*_IRAM0_PMS_MONITOR_1_REG |
| PMS_CORE_X_DRAM0_PMS_CONSTRAIN_0_REG | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_0_REG |
| | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG |
| PMS_CORE_*m*_DRAM0_PMS_MONITOR_0_REG | PMS_CORE_*m*_DRAM0_PMS_MONITOR_0_REG |
| | PMS_CORE_*m*_DRAM0_PMS_MONITOR_1_REG |
| **Lock SRAM Split Lines Configuration** | |
| PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_0_REG | PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_0_REG |
| | PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_*n*_REG (*n*: 1 - 5) |
| **Lock Peripherals Access Configuration** | |
| PMS_CORE_*m*_PIF_PMS_CONSTRAIN_0_REG | PMS_CORE_*m*_PIF_PMS_CONSTRAIN_0_REG |
| | PMS_CORE_*m*_PIF_PMS_CONSTRAIN_*n*_REG (*n*: 1 - 14) |
| PMS_REGION_PMS_CONSTRAIN_0_REG | PMS_REGION_PMS_CONSTRAIN_0_REG |
| | PMS_REGION_PMS_CONSTRAIN_*n*_REG (*n*: 1 - 14) |
| PMS_CORE_*m*_PIF_PMS_MONITOR_0_REG | PMS_CORE_*m*_PIF_PMS_CONSTRAIN_0_REG |
| | PMS_CORE_*m*_PIF_PMS_MONITOR_1_REG (*n*: 1 - 6) |
| **Lock Peripherals Access Configuration to Internal SRAM via GDMA** | |
| PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_0_REG | PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_0_REG |
| | PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_1_REG |
| PMS_DMA_APBPERI_UCHI0_PMS_CONSTRAIN_0_REG | PMS_DMA_APBPERI_UCHI0_PMS_CONSTRAIN_0_REG |
| | PMS_DMA_APBPERI_UCHI0_PMS_CONSTRAIN_1_REG |
| PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_0_REG | PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_0_REG |
| | PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_1_REG |
| PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_0_REG | PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_0_REG |
| | PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_1_REG |
| PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_0_REG | PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_0_REG |
| | PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_1_REG |
| PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_0_REG | PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_0_REG |
| | PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_1_REG |
| PMS_DMA_APBPERI_PMS_MONITOR_0_REG | PMS_DMA_APBPERI_PMS_MONITOR_0_REG |
| | PMS_DMA_APBPERI_PMS_MONITOR_1_REG |
| | PMS_DMA_APBPERI_PMS_MONITOR_2_REG |
| | PMS_DMA_APBPERI_PMS_MONITOR_3_REG |
| **Lock CPU's Access Configuration to Cache** | |
| EXTMEM_DBUS_PMS_TBL_LOCK_REG | EXTMEM_DBUS_PMS_TBL_ATTR_REG |
| | EXTMEM_DBUS_PMS_TBL_BOUNDARY0_REG |
| | EXTMEM_DBUS_PMS_TBL_BOUNDARY1_REG |
| | EXTMEM_DBUS_PMS_TBL_BOUNDARY2_REG |
| EXTMEM_IBUS_PMS_TBL_LOCK_REG | EXTMEM_IBUS_PMS_TBL_ATTR_REG |
| | EXTMEM_IBUS_PMS_TBL_BOUNDARY0_REG |
| | EXTMEM_IBUS_PMS_TBL_BOUNDARY1_REG |

| Lock Registers | Related Permission Control Registers |
|---|---|
| | EXTMEM_IBUS_PMS_TBL_BOUNDARY2_REG |

## 14.9 Register Summary

The addresses in this section are relative to the Permission Control base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

| Name | Description | Address | Access |
|---|---|---|---|
| Configuration Register | | | |
| PMS_PRIVILEGE_MODE_SEL_LOCK_REG | PMS_PRIVILEGE_MODE_SEL_LOCK_REG | 0x0008 | R/WL |
| PMS_PRIVILEGE_MODE_SEL_REG | PMS_PRIVILEGE_MODE_SEL_REG | 0x000C | R/WL |
| PMS_APB_PERIPHERAL_ACCESS_0_REG | APB peripheral configuration register 0 | 0x0010 | R/WL |
| PMS_APB_PERIPHERAL_ACCESS_1_REG | APB peripheral configuration register 1 | 0x0014 | R/WL |
| PMS_INTERNAL_SRAM_USAGE_0_REG | Internal SRAM configuration register 0 | 0x0018 | R/WL |
| PMS_INTERNAL_SRAM_USAGE_1_REG | Internal SRAM configuration register 1 | 0x001C | R/WL |
| PMS_INTERNAL_SRAM_USAGE_4_REG | Internal SRAM configuration register 4 | 0x0024 | R/WL |
| PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_0_REG | SPI2 GDMA Permission Config Register 0 | 0x0038 | R/WL |
| PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_1_REG | SPI2 GDMA Permission Config Register 1 | 0x003C | R/WL |
| PMS_DMA_APBPERI_UCHI0_PMS_CONSTRAIN_0_REG | UCHI0 GDMA Permission Config Register 0 | 0x0040 | R/WL |
| PMS_DMA_APBPERI_UCHI0_PMS_CONSTRAIN_1_REG | UCHI0 GDMA Permission Config Register 1 | 0x0044 | R/WL |
| PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_0_REG | I2S GDMA Permission Config Register 0 | 0x0048 | R/WL |
| PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_1_REG | I2S GDMA Permission Config Register 1 | 0x004C | R/WL |
| PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_0_REG | AES GDMA Permission Config Register 0 | 0x0068 | R/WL |
| PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_1_REG | AES GDMA Permission Config Register 1 | 0x006C | R/WL |
| PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_0_REG | SHA GDMA Permission Config Register 0 | 0x0070 | R/WL |
| PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_1_REG | SHA GDMA Permission Config Register 1 | 0x0074 | R/WL |
| PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_0_REG | ADC_DAC GDMA Permission Config Register 0 | 0x0078 | R/WL |
| PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_1_REG | ADC_DAC GDMA Permission Config Register 1 | 0x007C | R/WL |
| PMS_DMA_APBPERI_PMS_MONITOR_0_REG | GDMA Permission Interrupt Register 0 | 0x0080 | R/WL |
| PMS_DMA_APBPERI_PMS_MONITOR_1_REG | GDMA Permission Interrupt Register 1 | 0x0084 | R/WL |
| PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE _CONSTRAIN_*n*_REG (*n*: 0 - 5) | SRAM split line config register *n* | 0x0090 + 4 * *n* | R/WL |
| PMS_CORE_X_IRAM0_PMS_CONSTRAIN_0_REG | IBUS Permission Config Register 0 | 0x00A8 | R/WL |







GoBack

| Name | Description | Address | Access |
|---|---|---|---|
| PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG | IBUS Permission Config Register 1 | 0x00AC | R/WL |
| PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG | IBUS Permission Config Register 2 | 0x00B0 | R/WL |
| PMS_CORE_0_IRAM0_PMS_MONITOR_0_REG | CPU0 IBUS Permission Interrupt Register 0 | 0x00B4 | R/WL |
| PMS_CORE_0_IRAM0_PMS_MONITOR_1_REG | CPU0 IBUS Permission Interrupt Register 1 | 0x00B8 | R/WL |
| PMS_CORE_X_DRAM0_PMS_CONSTRAIN_0_REG | DBUS Permission Config Register 0 | 0x00C0 | R/WL |
| PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG | DBUS Permission Config Register 1 | 0x00C4 | R/WL |
| PMS_CORE_0_DRAM0_PMS_MONITOR_0_REG | CPU0 dBUS Permission Interrupt Register 0 | 0x00C8 | R/WL |
| PMS_CORE_0_DRAM0_PMS_MONITOR_1_REG | CPU0 dBUS Permission Interrupt Register 1 | 0x00CC | R/WL |
| PMS_CORE_0_PIF_PMS_CONSTRAIN_*n*_REG (*n*: 0 - 10) | Peripheral Permission Configuration Register *n* | 0x00D8 + 4 * *n* | R/WL |
| PMS_REGION_PMS_CONSTRAIN_*n*_REG (*n*: 0 - 10) | CPU Split_Region Permission Register *n* | 0x0104 + 4 * *n* | R/WL |
| PMS_CORE_0_PIF_PMS_MONITOR_0_REG | CPU PIF Permission Interrupt Register 0 | 0x0130 | R/WL |
| PMS_CORE_0_PIF_PMS_MONITOR_1_REG | CPU PIF Permission Interrupt Register 1 | 0x0134 | R/WL |
| PMS_CORE_0_PIF_PMS_MONITOR_4_REG | CPU PIF Permission Interrupt Register 4 | 0x0140 | R/WL |
| **Status Register** | | | |
| PMS_DMA_APBPERI_PMS_MONITOR_2_REG | GDMA Permission Interrupt Register 2 | 0x0088 | RO |
| PMS_DMA_APBPERI_PMS_MONITOR_3_REG | GDMA Permission Interrupt Register 3 | 0x008C | RO |
| PMS_CORE_0_IRAM0_PMS_MONITOR_2_REG | CPU0 IBUS Permission Interrupt Register 2 | 0x00BC | RO |
| PMS_CORE_0_DRAM0_PMS_MONITOR_2_REG | CPU0 dBUS Permission Interrupt Register 2 | 0x00D0 | RO |
| PMS_CORE_0_DRAM0_PMS_MONITOR_3_REG | CPU0 dBUS Permission Interrupt Register 3 | 0x00D4 | RO |
| PMS_CORE_0_PIF_PMS_MONITOR_2_REG | CPU PIF Permission Interrupt Register 2 | 0x0138 | RO |
| PMS_CORE_0_PIF_PMS_MONITOR_3_REG | CPU PIF Permission Interrupt Register 3 | 0x013C | RO |
| PMS_CORE_0_PIF_PMS_MONITOR_5_REG | CPU PIF Permission Interrupt Register 5 | 0x0144 | RO |
| PMS_CORE_0_PIF_PMS_MONITOR_6_REG | CPU PIF Permission Interrupt Register 6 | 0x0148 | RO |
| **Version Register** | | | |
| PMS_CLOCK_GATE_REG_REG | Clock Gate Config Register | 0x0170 | R/W |
| PMS_DATE_REG | Sensitive Version Register | 0x0FFC | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Registers** | | | |
| SYSCON_EXT_MEM_PMS_LOCK_REG | External Memory Permission Lock Register | 0x0020 | R/W |
| SYSCON_FLASH_ACE*n*_ATTR_REG (*n*: 0 - 3) | Flash Area*n* Permission Config Register | 0x0028 + 4 * *n* | R/W |
| SYSCON_SRAM_ACE*n*_ADDR_S (*n*: 0 - 3) | Flash Area*n* Starting Address Config Register | 0x0038 + 4 * *n* | R/W |
| SYSCON_FLASH_ACE*n*_SIZE_REG (*n*: 0 - 3) | Flash Area*n* Length Config Register | 0x0048 + 4 * *n* | R/W |
| SYSCON_SPI_MEM_PMS_CTRL_REG | External Memory Unauthorized Access Interrupt Register | 0x0088 | varies |
| SYSCON_SPI_MEM_REJECT_ADDR_REG | External Memory Unauthorized Access Address Register | 0x008C | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| **Permission Configure register** | | | |
| EXTMEM_IBUS_PMS_TBL_LOCK_REG | Cache IBUS Regions Lock | 0x00D8 | R/W |
| EXTMEM_IBUS_PMS_TBL_BOUNDARY*n*_REG (*n*: 0 - 2) | Cache IBUS Region*(n+1)* Starting Address Config Register | 0x00DC + 4 * *n* | R/W |
| EXTMEM_IBUS_PMS_TBL_ATTR_REG | Cache IBUS Region Permission Register | 0x00E8 | R/W |
| EXTMEM_DBUS_PMS_TBL_LOCK_REG | Cache DBUS Regions Lock | 0x00EC | R/W |
| EXTMEM_DBUS_PMS_TBL_BOUNDARY*n*_REG (*n*: 0 - 2) | Cache DBUS Region*(n+1)* Starting Address Config Register | 0x00F0 + 4 * *n* | R/W |
| EXTMEM_DBUS_PMS_TBL_ATTR_REG | Cache DBUS Region Permission Register | 0x00FC | R/W |

## 14.10 Registers

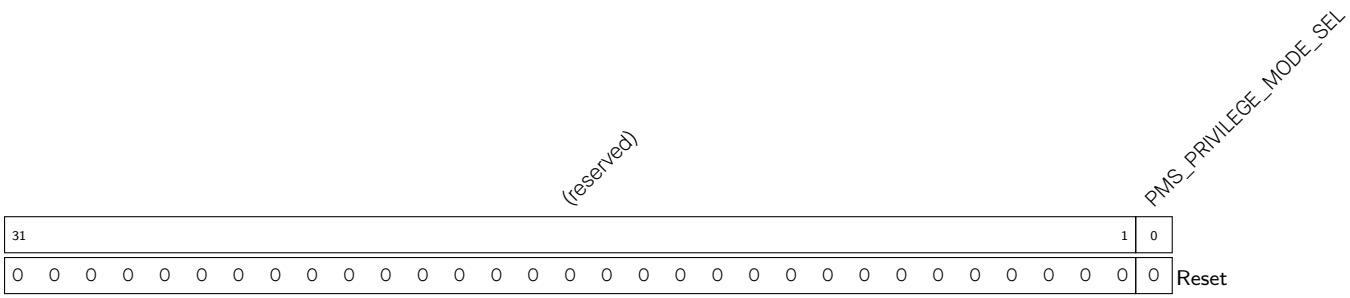
The addresses in this section are relative to the Permission Control base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 14.1. PMS_PRIVILEGE_MODE_SEL_LOCK_REG (0x0008)**

| (reserved) | PMS_PRIVILEGE_MODE_SEL_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_PRIVILEGE_MODE_SEL_LOCK** Set this bit to lock privilege_mode configuration register. (R/WL)

**Register 14.2. PMS_PRIVILEGE_MODE_SEL_REG (0x000C)**



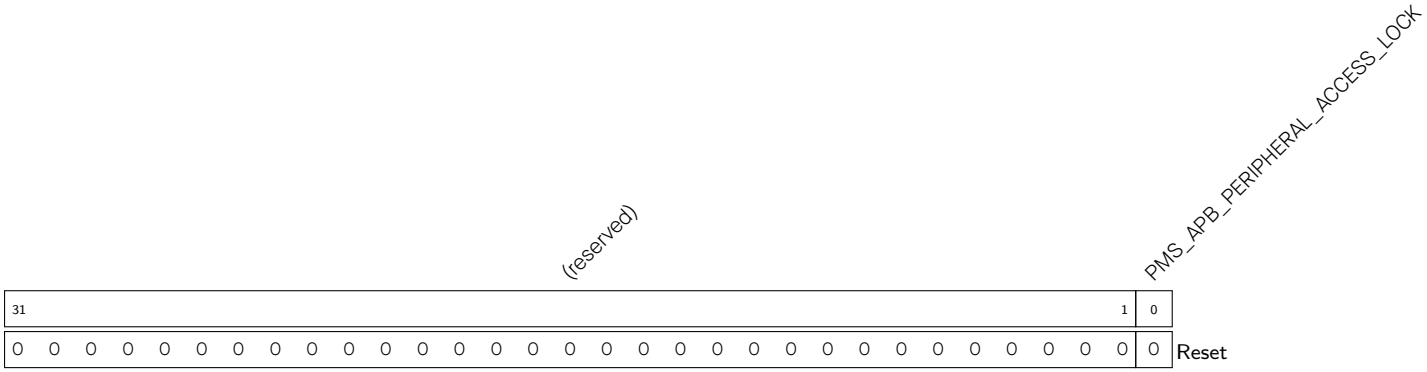

**PMS_PRIVILEGE_MODE_SEL** Configures how to enter privileged environment:

0: By configuring the World Controller

1: By configuring CPU's privileged level

(R/WL)

**Register 14.3. PMS_APB_PERIPHERAL_ACCESS_0_REG (0x0010)**




**PMS_APB_PERIPHERAL_ACCESS_LOCK** Set this bit to lock APB peripheral configuration register. (R/WL)

**Register 14.4. PMS_APB_PERIPHERAL_ACCESS_1_REG (0x0014)**

| (reserved) | PMS_APB_PERIPHERAL_ACCESS_SPLIT_BURST | |
|---|---|---|
| 31 … 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 | 1 | Reset |

**PMS_APB_PERIPHERAL_ACCESS_SPLIT_BURST** Set this bit to support split function for AHB access to APB peripherals. (R/WL)

**Register 14.5. PMS_INTERNAL_SRAM_USAGE_0_REG (0x0018)**

| (reserved) | PMS_INTERNAL_SRAM_USAGE_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_INTERNAL_SRAM_USAGE_LOCK** Set this bit to lock internal SRAM Configuration Register. (R/WL)

Register 14.6. PMS_INTERNAL_SRAM_USAGE_1_REG (0x001C)

| (reserved) | | PMS_INTERNAL_SRAM_USAGE_CPU_CACHE |
|---|---|---|
| 31 … 4 | 3 … 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 | 0x7 | 1 |
| | | Reset |

**PMS_INTERNAL_SRAM_USAGE_CPU_CACHE** Configures SRAM0 is allocated for CPU or ICACHE. (R/WL)

Register 14.7. PMS_INTERNAL_SRAM_USAGE_4_REG (0x0024)

| (reserved) | PMS_INTERNAL_SRAM_USAGE_LOG_SRAM |
|---|---|
| 31 … 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 |
| | Reset |

**PMS_INTERNAL_SRAM_USAGE_LOG_SRAM** Set 1 to enable ASSIST_DEBUG access SRAM1. (R/WL)

Register 14.8. PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_0_REG (0x0038)

| (reserved) | PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_LOCK | |
|---|---|---|
| 31 | 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 | Reset |

**PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_LOCK** Set this bit to lock SPI2's DMA permission configuration register. (R/WL)

## Register 14.9. PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_1_REG (0x003C)

| 31 | 8 | 7 6 | 5 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|
| (reserved) | | PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3 | PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2 | PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1 | PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0** Configure SPI2's permission to the instruction region. (R/WL)

**PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1** Configure SPI2's permission to the data region0 of SRAM. (R/WL)

**PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2** Configure SPI2's permission to the data region1 of SRAM. (R/WL)

**PMS_DMA_APBPERI_SPI2_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3** Configure SPI2's permission to the data region2 of SRAM. (R/WL)

Register 14.10. PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_0_REG (0x0040)

| (reserved) | PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_LOCK** Set this bit to lock UHCIO's DMA permission configuration register. (R/WL)

## Register 14.11. PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_1_REG (0x0044)

| 31 | 8 | 7 6 | 5 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|
| (reserved) | | PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3 | PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2 | PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1 | PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0** Configure UHCIO's permission to the instruction region. (R/WL)

**PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1** Configure UHCIO's permission to the data region0 of SRAM. (R/WL)

**PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2** Configure UHCIO's permission to the data region1 of SRAM. (R/WL)

**PMS_DMA_APBPERI_UCHIO_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3** Configure UHCIO's permission to the data region2 of SRAM. (R/WL)

## Register 14.12. PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_0_REG (0x0048)

| (reserved) | PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_LOCK** Set this bit to lock I2S's DMA permission configuration register. (R/WL)

## Register 14.13. PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_1_REG (0x004C)

| (reserved) | PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3 | PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2 | PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1 | PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0 | |
|---|---|---|---|---|---|
| 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 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0** Configure I2S's permission to the instruction region. (R/WL)

**PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1** Configure I2S's permission to the data region0 of SRAM. (R/WL)

**PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2** Configure I2S's permission to the data region1 of SRAM. (R/WL)

**PMS_DMA_APBPERI_I2S0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3** Configure I2S's permission to the data region2 of SRAM. (R/WL)

Register 14.14. PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_0_REG (0x0068)

| (reserved) | PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_LOCK** Set this bit to lock AES's DMA permission configuration register. (R/WL)

## Register 14.15. PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_1_REG (0x006C)

| (reserved) | PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3 | PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2 | PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1 | PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0 | |
|---|---|---|---|---|---|
| 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 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0** Configure AES's permission to the instruction region. (R/WL)

**PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1** Configure AES's permission to the data region0 of SRAM. (R/WL)

**PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2** Configure AES's permission to the data region1 of SRAM. (R/WL)

**PMS_DMA_APBPERI_AES_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3** Configure AES's permission to the data region2 of SRAM. (R/WL)

Register 14.16. PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_0_REG (0x0070)

| (reserved) | PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_LOCK** Set this bit to lock SHA's DMA permission configuration register. (R/WL)

Register 14.17. PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_1_REG (0x0074)

| (reserved) | PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3 | PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2 | PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1 | PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0 | |
|---|---|---|---|---|---|
| 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 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0** Configure SHA's permission to the instruction region. (R/WL)

**PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1** Configure SHA's permission to the data region0 of SRAM. (R/WL)

**PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2** Configure SHA's permission to the data region1 of SRAM. (R/WL)

**PMS_DMA_APBPERI_SHA_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3** Configure SHA's permission to the data region2 of SRAM. (R/WL)

Register 14.18. PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_0_REG (0x0078)

| (reserved) | PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_LOCK** Set this bit to lock ADC_DAC's DMA permission configuration register. (R/WL)

Register 14.19. PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_1_REG (0x007C)

| (reserved) | PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3 | PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2 | PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1 | PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0 | |
|---|---|---|---|---|---|
| 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 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0** Configure ADC_DAC's permission to the instruction region. (R/WL)

**PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1** Configure ADC_DAC's permission to the data region0 of SRAM. (R/WL)

**PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2** Configure ADC_DAC's permission to the data region1 of SRAM. (R/WL)

**PMS_DMA_APBPERI_ADC_DAC_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3** Configure ADC_DAC's permission to the data region2 of SRAM. (R/WL)

Register 14.20. PMS_DMA_APBPERI_PMS_MONITOR_0_REG (0x0080)

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | PMS_DMA_APBPERI_PMS_MONITOR_LOCK | |
| 0 0 0 0 0 0 0 0 0 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 |

**PMS_DMA_APBPERI_PMS_MONITOR_LOCK** Set this bit to lock DMA access interrupt configuration register. (R/WL)

## Register 14.21. PMS_DMA_APBPERI_PMS_MONITOR_1_REG (0x0084)

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_EN | PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_CLR | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Reset |

**PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_CLR** Set this bit to clear DMA access interrupt status. (R/WL)

**PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_EN** Set this bit to enable interrupt upon illegal DMA access. (R/WL)

Register 14.22. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_0_REG (0x0090)

| (reserved) | PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_LOCK | |
|---|---|---|
| 31 | 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 | Reset |

**PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_LOCK** Set this bit to lock internal SRAM's split lines configuration. (R/WL)

**Register 14.23. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_1_REG (0x0094)**

| (reserved) | PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_SPLITADDR | (reserved) | PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_2 | PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_1 | PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_0 | |
|---|---|---|---|---|---|---|
| 31 22 | 21 14 | 13 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 | Reset |

**PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_0** Configures Block0's category field for the instruction and data split line IRAM0_DRAM0_Split_Line. (R/WL)

**PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_1** Configures Block1's category field for the instruction and data split line IRAM0_DRAM0_Split_Line. (R/WL)

**PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_CATEGORY_2** Configures Block2's category field for the instruction and data split line IRAM0_DRAM0_Split_Line. (R/WL)

**PMS_CORE_X_IRAM0_DRAM0_DMA_SRAM_SPLITADDR** Configures the split address of the instruction and data split line IRAM0_DRAM0_Split_Line. (R/WL)

**Register 14.24. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_2_REG (0x0098)**

| (reserved) | PMS_CORE_X_IRAM0_SRAM_LINE_0_SPLITADDR | (reserved) | PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_2 | PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_1 | PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_0 | |
|---|---|---|---|---|---|---|
| 31 … 22 | 21 … 14 | 13 … 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 | Reset |

**PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_0** Configures Block0's category field for the instruction internal split line IRAM0_Split_Line_0. (R/WL)

**PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_1** Configures Block1's category field for the instruction internal split line IRAM0_Split_Line_0. (R/WL)

**PMS_CORE_X_IRAM0_SRAM_LINE_0_CATEGORY_2** Configures Block2's category field for the instruction internal split line IRAM0_Split_Line_0. (R/WL)

**PMS_CORE_X_IRAM0_SRAM_LINE_0_SPLITADDR** Configures the split address of the instruction internal split line IRAM0_Split_Line_0. (R/WL)

Register 14.25. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_3_REG (0x009C)

| (reserved) | PMS_CORE_X_IRAM0_SRAM_LINE_1_SPLITADDR | (reserved) | PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_2 | PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_1 | PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_0 | |
|---|---|---|---|---|---|---|
| 31 … 22 | 21 … 14 | 13 … 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 | Reset |

**PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_0** Configures Block0's category field for the instruction internal split line IRAM0_Split_Line_1. (R/WL)

**PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_1** Configures Block1's category field for the instruction internal split line IRAM0_Split_Line_1. (R/WL)

**PMS_CORE_X_IRAM0_SRAM_LINE_1_CATEGORY_2** Configures Block2's category field for the instruction internal split line IRAM0_Split_Line_1. (R/WL)

**PMS_CORE_X_IRAM0_SRAM_LINE_1_SPLITADDR** Configures the split address of the instruction internal split line IRAM0_Split_Line_1. (R/WL)

Register 14.26. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_4_REG (0x00A0)

| (reserved) | PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_SPLITADDR | (reserved) | PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_2 | PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_1 | PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_0 | |
|---|---|---|---|---|---|---|
| 31 22 | 21 14 | 13 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 | Reset |

**PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_0** Configures Block0's category field for data internal split line DRAM0_Split_Line_0. (R/WL)

**PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_1** Configures Block1's category field for data internal split line DRAM0_Split_Line_0. (R/WL)

**PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_CATEGORY_2** Configures Block2's category field for data internal split line DRAM0_Split_Line_0. (R/WL)

**PMS_CORE_X_DRAM0_DMA_SRAM_LINE_0_SPLITADDR** Configures the split address of data internal split line DRAM0_Split_Line_0. (R/WL)

**Register 14.27. PMS_CORE_X_IRAM0_DRAM0_DMA_SPLIT_LINE_CONSTRAIN_5_REG (0x00A4)**

| (reserved) | PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_SPLITADDR | (reserved) | PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_2 | PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_1 | PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_0 | |
|---|---|---|---|---|---|---|
| 31 – 22 | 21 – 14 | 13 – 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 | Reset |

**PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_0** Configures Block0's category field for data internal split line DRAM0_Split_Line_1. (R/WL)

**PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_1** Configures Block1's category field for data internal split line DRAM0_Split_Line_1. (R/WL)

**PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_CATEGORY_2** Configures Block2's category field for data internal split line DRAM0_Split_Line_1. (R/WL)

**PMS_CORE_X_DRAM0_DMA_SRAM_LINE_1_SPLITADDR** Configures the split address of data internal split line DRAM0_Split_Line_1. (R/WL)

Register 14.28. PMS_CORE_X_IRAM0_PMS_CONSTRAIN_0_REG (0x00A8)

| (reserved) | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_LOCK** Set this bit to lock the permission of CPU IBUS to internal SRAM. (R/WL)

**Register 14.29. PMS_CORE_X_IRAM0_PMS_CONSTRAIN_1_REG (0x00AC)**

| 31 | 21 | 20 | 18 | 17 | 15 | 14 | 12 | 11 | 9 | 8 | 6 | 5 | 3 | 2 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_ROM_U_MODE_PMS | | (reserved) | | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_CACHEDATAARRAY_PMS_0 | | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_3 | | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_2 | | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_1 | | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_0 | | |
| 0 0 0 0 0 0 0 0 0 0 0 | | 0x7 | | 0 0 0 | | 0x7 | | 0x7 | | 0x7 | | 0x7 | | 0x7 | | Reset |

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_0** Configures the permission of CPU's IBUS to instruction region0 of SRAM from the unpriviledged environment. (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_1** Configures the permission of CPU's IBUS to instruction region1 of SRAM from the unpriviledged environment. (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_2** Configures the permission of CPU's IBUS to instruction region2 of SRAM from the unpriviledged environment. (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_3** Configures the permission of CPU's IBUS to data region of SRAM from the unpriviledged environment. (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_U_MODE_CACHEDATAARRAY_PMS_0** Configure the permission of CPU's IBUS to SRAM0 from the unpriviledged environment. (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_ROM_U_MODE_PMS** Configure the permission of CPU's IBUS to ROM from the unpriviledged environment. (R/WL)

Register 14.30. PMS_CORE_X_IRAM0_PMS_CONSTRAIN_2_REG (0x00B0)

| 31–21 | 20–18 | 17–15 | 14–12 | 11–9 | 8–6 | 5–3 | 2–0 | |
|---|---|---|---|---|---|---|---|---|
| (reserved) | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_ROM_M_MODE_PMS | (reserved) | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_CACHEDATAARRAY_PMS_0 | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3 | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2 | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1 | PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0 | |
| 0 0 0 0 0 0 0 0 0 0 0 | 0x7 | 0 0 0 | 0x7 | 0x7 | 0x7 | 0x7 | 0x7 | Reset |

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0** Configures the permission of CPU's IBUS to instruction region0 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1** Configures the permission of CPU's IBUS to instruction region1 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2** Configures the permission of CPU's IBUS to instruction region2 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3** Configures the permission of CPU's IBUS to data region of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_SRAM_M_MODE_CACHEDATAARRAY_PMS_0** Configures the permission of CPU's IBUS to SRAM0 from the privileged environment (R/WL)

**PMS_CORE_X_IRAM0_PMS_CONSTRAIN_ROM_M_MODE_PMS** Configures the permission of CPU's IBUS to ROM from the privileged environment (R/WL)

Register 14.31. PMS_CORE_0_IRAM0_PMS_MONITOR_0_REG (0x00B4)

| (reserved) | PMS_CORE_0_IRAM0_PMS_MONITOR_LOCK |
|---|---|
| 31 … 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 |
| | Reset |

**PMS_CORE_0_IRAM0_PMS_MONITOR_LOCK** Set this bit to lock CPU0's IBUS interrupt configuration. (R/WL)

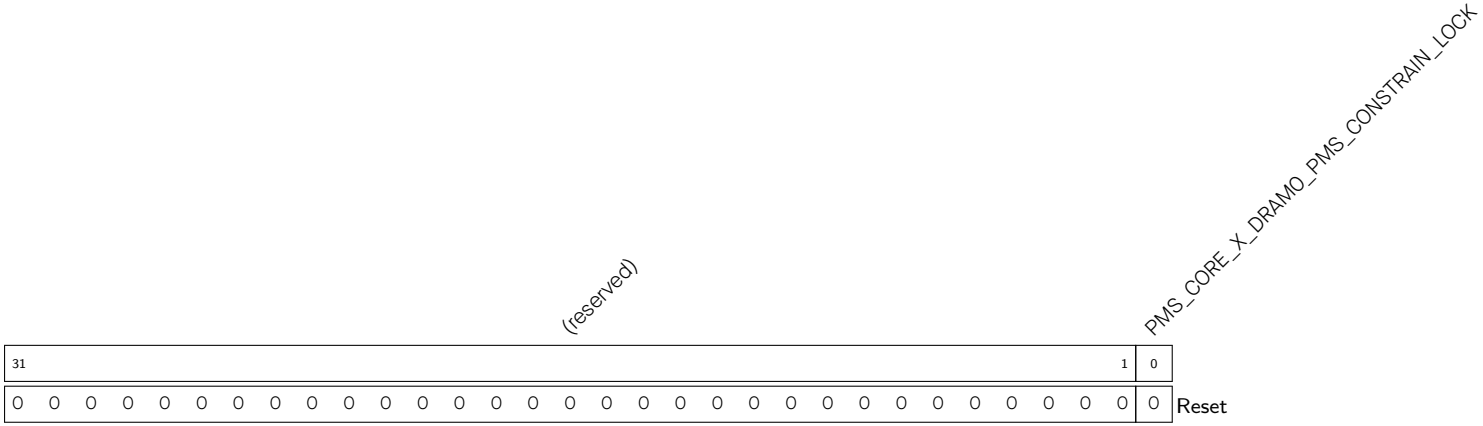
Register 14.32. PMS_CORE_0_IRAM0_PMS_MONITOR_1_REG (0x00B8)

| 31 | 2 | 1 | 0 | |
|---|---|---|---|---|
| (reserved) | | PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_EN | PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_CLR | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 1 | 1 | Reset |

**PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_CLR** Set this bit to clear the interrupt triggered when CPU0's IBUS tries to access SRAM or ROM unauthorized. (R/WL)

**PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_EN** Set this bit to enable interrupt when CPU0's IBUS tries to access SRAM or ROM unauthorized. (R/WL)

Register 14.33. PMS_CORE_X_DRAM0_PMS_CONSTRAIN_0_REG (0x00C0)




**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_LOCK** Set this bit to lock the permission of CPU DBUS to internal SRAM. (R/WL)

**Register 14.34. PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG (0x00C4)**

| (reserved) | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_ROM_U_MODE_PMS | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_ROM_M_MODE_PMS | (reserved) | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_3 | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_2 | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_1 | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_0 | (reserved) | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3 | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2 | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1 | PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 28 | 27 26 | 25 24 | 23 20 | 19 18 | 17 16 | 15 14 | 13 12 | 11 8 | 7 6 | 5 4 | 3 2 | 1 0 | |
| 0 0 0 0 | 0x3 | 0x3 | 0 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | 0 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_0** Configures the permission of CPU's DBUS to instruction region of SRAM from the privileged environment It's advised to configure this field to 0. (R/WL)

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_1** Configures the permission of CPU's DBUS to data region0 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_2** Configures the permission of CPU's DBUS to data region1 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_M_MODE_PMS_3** Configures the permission of CPU's DBUS to data region2 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_0** Configures the permission of CPU's DBUS to instruction region of SRAM from the privileged environment It's advised to configure this field to 0. (R/WL)

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







GoBack

Register 14.34. PMS_CORE_X_DRAM0_PMS_CONSTRAIN_1_REG (0x00C4)

Continued from the previous page...

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_1** Configures the permission of CPU's DBUS to data region0 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_2** Configures the permission of CPU's DBUS to data region1 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_SRAM_U_MODE_PMS_3** Configures the permission of CPU's DBUS to data region2 of SRAM from the privileged environment (R/WL)

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_ROM_M_MODE_PMS** Configures the permission of CPU's DBUS to ROM from the privileged environment (R/WL)

**PMS_CORE_X_DRAM0_PMS_CONSTRAIN_ROM_U_MODE_PMS** Configures the permission of CPU's DBUS to ROM from the unpriviledged environment. (R/WL)







GoBack

Register 14.35. PMS_CORE_0_DRAM0_PMS_MONITOR_0_REG (0x00C8)

| (reserved) | PMS_CORE_0_DRAM0_PMS_MONITOR_LOCK |
|---|---|
| 31 … 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 |
| | Reset |

**PMS_CORE_0_DRAM0_PMS_MONITOR_LOCK** Set this bit to lock CPU's DBUS interrupt configuration. (R/WL)

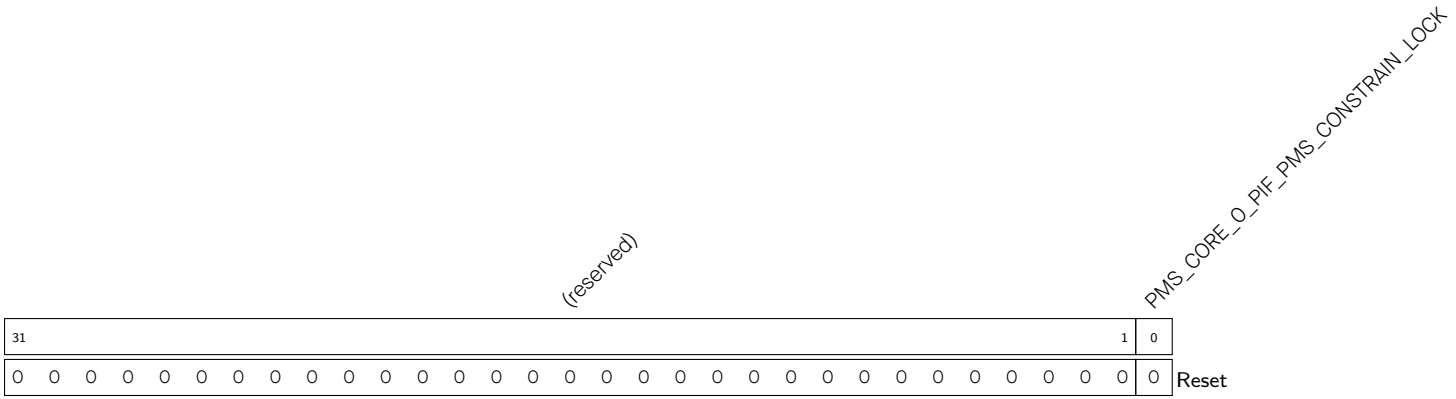
Register 14.36. PMS_CORE_0_DRAM0_PMS_MONITOR_1_REG (0x00CC)

| (reserved) | PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_EN | PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_CLR | |
|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 | 1 | 1 | Reset |

**PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_CLR** Set this bit to clear the interrupt triggered when CPU0's dBUS tries to access SRAM or ROM unauthorized. (R/WL)

**PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_EN** Set this bit to enable interrupt when CPU0's dBUS tries to access SRAM or ROM unauthorized. (R/WL)

Register 14.37. PMS_CORE_0_PIF_PMS_CONSTRAIN_0_REG (0x00D8)




**PMS_CORE_0_PIF_PMS_CONSTRAIN_LOCK** Set this bit to lock CPU permission to different peripherals. (R/WL)

Register 14.38. PMS_CORE_0_PIF_PMS_CONSTRAIN_1_REG (0x00DC)

| PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_UART1 | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_IO_MUX | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_RTC | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_GPIO | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_GOSPI_0 | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_GOSPI_1 | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_UART | |
|---|---|---|---|---|---|---|---|---|---|
| 31 30 | 29 18 | 17 16 | 15 14 | 13 8 | 7 6 | 5 4 | 3 2 | 1 0 | |
| 0x3 | 0 0 0 0 0 0 0 0 0 0 0 0 | 0x3 | 0x3 | 0 0 0 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_UART** Configures CPU's permission to access UART 0 from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_GOSPI_1** Configures CPU's permission to access SPI 1 from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_GOSPI_0** Configures CPU's permission to access SPI 0 from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_GPIO** Configures CPU's permission to access GPIO from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_RTC** Configures CPU's permission to access eFuse Controller & PMU from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_IO_MUX** Configures CPU's permission to access IO_MUX from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_UART1** Configures CPU's permission to access UART 1 from the privileged environment (R/WL)

**Register 14.39. PMS_CORE_0_PIF_PMS_CONSTRAIN_2_REG (0x00E0)**

| 31 30 | 29 28 | 27 26 | 25 18 | 17 16 | 15 12 | 11 10 | 9 8 | 7 6 | 5 4 | 3 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_SYSTIMER | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_TIMERGROUP1 | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_TIMERGROUP | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_LEDC | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_RMT | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_UHCI0 | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_I2C_EXT0 | (reserved) | |
| 0x3 | 0x3 | 0x3 | 0 0 0 0 0 0 0 0 | 0x3 | 0 0 0 0 | 0x3 | 0 0 | 0x3 | 0x3 | 0 0 0 0 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_I2C_EXT0** Configures CPU's permission to access I2C 0 from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_UHCI0** Configures CPU's permission to access UHCI 0 from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_RMT** Configures CPU's permission to access Remote Control Peripheral from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_LEDC** Configures CPU's permission to access LED PWM Controller from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_TIMERGROUP** Configures CPU's permission to access Timer Group 0 from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_TIMERGROUP1** Configures CPU's permission to access Timer Group 1 from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_SYSTIMER** Configures CPU's permission to access System Timer from the privileged environment (R/WL)

### Register 14.40. PMS_CORE_0_PIF_PMS_CONSTRAIN_3_REG (0x00E4)

| 31–16 | 15–14 | 13–12 | 11–10 | 9–6 | 5–4 | 3–2 | 1–0 | |
|---|---|---|---|---|---|---|---|---|
| (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_I2S1 | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_CAN | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_APB_CTRL | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_SPI_2 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x3 | 0 0 | 0x3 | 0 0 0 0 | 0x3 | 0 0 | 0x3 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_SPI_2** Configures CPU's permission to access SPI 2 from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_APB_CTRL** Configures CPU's permission to access APB Controller from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_CAN** Configures CPU's permission to access Two-wire Automotive Interface from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_I2S1** Configures CPU's permission to access I2S 1 from the privileged environment (R/WL)

Register 14.41. PMS_CORE_0_PIF_PMS_CONSTRAIN_4_REG (0x00E8)

| 31–28 | 27–26 | 25–24 | 23–22 | 21–20 | 19–18 | 17–16 | 15–14 | 13–10 | 9–8 | 7–6 | 5–4 | 3–2 | 1–0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_AD | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_CACHE_CONFIG | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_INTERRUPT | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_PMS | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_SYSTEM | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_USB_DEVICE | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_APB_ADC | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_CRYPTO_DMA | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_CRYPTO_PERI | PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_USB_WRAP | (reserved) | |
| 0 0 0 0 | 0x3 | 0x3 | 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | 0 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | 0 0 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_USB_WRAP** Configures CPU's permission to access USB OTG External from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_CRYPTO_PERI** Configures CPU's permission to access Accelerators from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_CRYPTO_DMA** Configures CPU's permission to access GDMA from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_APB_ADC** Configures CPU's permission to access ADC Controller from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_USB_DEVICE** Configures CPU's permission to access USB OTG Core from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_SYSTEM** Configures CPU's permission to access System Registers from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_PMS** Configures CPU's permission to access PMS Registers from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_INTERRUPT** Configures CPU's permission to access Interrupt Matrix from the privileged environment (R/WL)

Register 14.41. PMS_CORE_0_PIF_PMS_CONSTRAIN_4_REG (0x00E8)

Continued from the previous page...

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_CACHE_CONFIG** Configures CPU's permission to access Cache & XTS_AES from the privileged environment (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_M_MODE_AD** Configures CPU's permission to access Debug Assist from the privileged environment (R/WL)

**Register 14.42. PMS_CORE_0_PIF_PMS_CONSTRAIN_5_REG (0x00EC)**

| 31 30 | 29 18 | 17 16 | 15 14 | 13 8 | 7 6 | 5 4 | 3 2 | 1 0 | |
|---|---|---|---|---|---|---|---|---|---|
| PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_UART1 | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_IO_MUX | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_RTC | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_GPIO | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_GOSPI_0 | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_GOSPI_1 | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_UART | |
| 0x3 | 0 0 0 0 0 0 0 0 0 0 0 0 | 0x3 | 0x3 | 0 0 0 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_UART** Configures CPU's permission to access UART 0 from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_GOSPI_1** Configures CPU's permission to access SPI 1 from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_GOSPI_0** Configures CPU's permission to access SPI 0 from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_GPIO** Configures CPU's permission to access GPIO from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_RTC** Configures CPU's permission to access eFuse Controller & PMU from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_IO_MUX** Configures CPU's permission to access IO_MUX from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_UART1** Configures CPU's permission to access UART 1 from the unpriviledged environment. (R/WL)

### Register 14.43. PMS_CORE_0_PIF_PMS_CONSTRAIN_6_REG (0x00F0)

| 31 30 | 29 28 | 27 26 | 25 18 | 17 16 | 15 12 | 11 10 | 9 8 | 7 6 | 5 4 | 3 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_SYSTIMER | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_TIMERGROUP1 | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_TIMERGROUP | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_LEDC | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_RMT | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_UHCI0 | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_I2C_EXT0 | (reserved) | |
| 0x3 | 0x3 | 0x3 | 0 0 0 0 0 0 0 0 | 0x3 | 0 0 0 0 | 0x3 | 0 0 | 0x3 | 0x3 | 0 0 0 0 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_I2C_EXT0** Configures CPU's permission to access I2C 0 from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_UHCI0** Configures CPU's permission to access UHCI 0 from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_RMT** Configures CPU's permission to access Remote Control Peripheral from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_LEDC** Configures CPU's permission to access LED PWM Controller from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_TIMERGROUP** Configures CPU's permission to access Timer Group 0 from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_TIMERGROUP1** Configures CPU's permission to access Timer Group 1 from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_SYSTIMER** Configures CPU's permission to access System Timer from the unpriviledged environment. (R/WL)

### Register 14.44. PMS_CORE_0_PIF_PMS_CONSTRAIN_7_REG (0x00F4)

| 31–16 | 15–14 | 13–12 | 11–10 | 9–6 | 5–4 | 3–2 | 1–0 | |
|---|---|---|---|---|---|---|---|---|
| (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_I2S1 | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_CAN | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_APB_CTRL | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_SPI_2 | |
| 0000000000000000 | 0x3 | 00 | 0x3 | 0000 | 0x3 | 00 | 0x3 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_SPI_2** Configures CPU's permission to access SPI 2 from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_APB_CTRL** Configures CPU's permission to access APB Controller from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_CAN** Configures CPU's permission to access Two-wire Automotive Interface from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_I2S1** Configures CPU's permission to access I2S 1 from the unpriviledged environment. (R/WL)

**Register 14.45. PMS_CORE_0_PIF_PMS_CONSTRAIN_8_REG (0x00F8)**

| 31–28 | 27–26 | 25–24 | 23–22 | 21–20 | 19–18 | 17–16 | 15–14 | 13–10 | 9–8 | 7–6 | 5–4 | 3–2 | 1–0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_AD | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_CACHE_CONFIG | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_INTERRUPT | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_PMS | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_SYSTEM | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_USB_DEVICE | (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_APB_ADC | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_CRYPTO_DMA | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_CRYPTO_PERI | PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_USB_WRAP | (reserved) | |
| 0 0 0 0 | 0x3 | 0x3 | 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | 0 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | 0 0 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_USB_WRAP** Configures CPU's permission to access USB OTG External from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_CRYPTO_PERI** Configures CPU's permission to access Accelerators from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_CRYPTO_DMA** Configures CPU's permission to access GDMA from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_APB_ADC** Configures CPU's permission to access ADC Controller from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_USB_DEVICE** Configures CPU's permission to access USB OTG Core from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_SYSTEM** Configures CPU's permission to access System Registers from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_PMS** Configures CPU's permission to access PMS Registers from the unpriviledged environment. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_U_MODE_INTERRUPT** Configures CPU's permission to access Interrupt Matrix from the unpriviledged environment. (R/WL)

Register 14.46. PMS_CORE_0_PIF_PMS_CONSTRAIN_9_REG (0x00FC)

| (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_SPLTADDR_U_MODE | PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_SPLTADDR_M_MODE | |
|---|---|---|---|
| 31 22 | 21 11 | 10 0 | |
| 0 0 0 0 0 0 0 0 0 0 | 0x7ff | 0x7ff | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_SPLTADDR_M_MODE** Configures the address to split RTC Fast Memory into two regions in unpriviledgeddenvironment for CPU. Note you should use address offset, instead of absolute address. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_SPLTADDR_U_MODE** Configures the address to split RTC Fast Memory into two regions in privilegedenvironment for CPU. Note you should use address offset, instead of absolute address. (R/WL)

Register 14.47. PMS_CORE_0_PIF_PMS_CONSTRAIN_10_REG (0x0100)

| (reserved) | PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_U_MODE_H | PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_U_MODE_L | PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_M_MODE_H | PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_M_MODE_L | |
|---|---|---|---|---|---|
| 31 … 12 | 11 … 9 | 8 … 6 | 5 … 3 | 2 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x7 | 0x7 | 0x7 | 0x7 | Reset |

**PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_M_MODE_L** Configures the permission of CPU from unpriviledgeddenvironment to the lower region of RTC Fast Memory. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_M_MODE_H** Configures the permission of CPU from unpriviledgeddenvironment to the higher region of RTC Fast Memory. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_U_MODE_L** Configures the permission of CPU from privilegedenvironment to the lower region of RTC Fast Memory. (R/WL)

**PMS_CORE_0_PIF_PMS_CONSTRAIN_RTCFAST_U_MODE_H** Configures the permission of CPU from privilegedenvironment to the higher region of RTC Fast Memory. (R/WL)

Register 14.48. PMS_REGION_PMS_CONSTRAIN_0_REG (0x0104)

| (reserved) | PMS_REGION_PMS_CONSTRAIN_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**PMS_REGION_PMS_CONSTRAIN_LOCK** Set this bit to lock Core0's permission to peripheral regions. (R/WL)

Register 14.49. PMS_REGION_PMS_CONSTRAIN_1_REG (0x0108)

| 31 … 14 | 13 … 12 | 11 … 10 | 9 … 8 | 7 … 6 | 5 … 4 | 3 … 2 | 1 … 0 | |
|---|---|---|---|---|---|---|---|---|
| (reserved) | PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_6 | PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_5 | PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_4 | PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_3 | PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_2 | PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_1 | PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_0** Configures CPU's permission to Peri Region0 from the privileged environment (R/WL)

**PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_1** Configures CPU's permission to Peri Region1 from the privileged environment (R/WL)

**PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_2** Configures CPU's permission to Peri Region2 from the privileged environment (R/WL)

**PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_3** Configures CPU's permission to Peri Region3 from the privileged environment (R/WL)

**PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_4** Configures CPU's permission to Peri Region4 from the privilegedenvironment (R/WL)

**PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_5** Configures CPU's permission to Peri Region5 from the privileged environment (R/WL)

**PMS_REGION_PMS_CONSTRAIN_M_MODE_AREA_6** Configures CPU's permission to Peri Region6 from the privileged environment (R/WL)







GoBack

Register 14.50. PMS_REGION_PMS_CONSTRAIN_2_REG (0x010C)

| 31 … 14 | 13 … 12 | 11 … 10 | 9 … 8 | 7 … 6 | 5 … 4 | 3 … 2 | 1 … 0 | |
|---|---|---|---|---|---|---|---|---|
| (reserved) | PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_6 | PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_5 | PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_4 | PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_3 | PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_2 | PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_1 | PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x3 | 0x3 | 0x3 | 0x3 | 0x3 | 0x3 | 0x3 | Reset |

**PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_0** Configures CPU's permission to Peri Region0 from the unpriviledged environment. (R/WL)

**PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_1** Configures CPU's permission to Peri Region1 from the unpriviledged environment. (R/WL)

**PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_2** Configures CPU's permission to Peri Region2 from the unpriviledged environment. (R/WL)

**PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_3** Configures CPU's permission to Peri Region3 from the unpriviledged environment. (R/WL)

**PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_4** Configures CPU's permission to Peri Region4 from the unpriviledged environment. (R/WL)

**PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_5** Configures CPU's permission to Peri Region5 from the unpriviledged environment. (R/WL)

**PMS_REGION_PMS_CONSTRAIN_U_MODE_AREA_6** Configures CPU's permission to Peri Region6 from the unpriviledged environment. (R/WL)







GoBack

**Register 14.51. PMS_REGION_PMS_CONSTRAIN_3_REG (0x0110)**

| (reserved) | PMS_REGION_PMS_CONSTRAIN_ADDR_0 | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0 | Reset |

**PMS_REGION_PMS_CONSTRAIN_ADDR_0** Configures the starting address of Region0 for CPU0. (R/WL)

**Register 14.52. PMS_REGION_PMS_CONSTRAIN_4_REG (0x0114)**

| (reserved) | PMS_REGION_PMS_CONSTRAIN_ADDR_1 | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0 | Reset |

**PMS_REGION_PMS_CONSTRAIN_ADDR_1** Configures the starting address of Region1 for CPU0. (R/WL)

Register 14.53. PMS_REGION_PMS_CONSTRAIN_5_REG (0x0118)

| (reserved) | PMS_REGION_PMS_CONSTRAIN_ADDR_2 | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0 | Reset |

**PMS_REGION_PMS_CONSTRAIN_ADDR_2** Configures the starting address of Region2 for CPU0. (R/WL)

Register 14.54. PMS_REGION_PMS_CONSTRAIN_6_REG (0x011C)

| (reserved) | PMS_REGION_PMS_CONSTRAIN_ADDR_3 | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0 | Reset |

**PMS_REGION_PMS_CONSTRAIN_ADDR_3** Configures the starting address of Region3 for CPU0. (R/WL)

Register 14.55. PMS_REGION_PMS_CONSTRAIN_7_REG (0x0120)

| (reserved) | PMS_REGION_PMS_CONSTRAIN_ADDR_4 | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0 | Reset |

**PMS_REGION_PMS_CONSTRAIN_ADDR_4** Configures the starting address of Region4 for CPU0. (R/WL)

Register 14.56. PMS_REGION_PMS_CONSTRAIN_8_REG (0x0124)

| (reserved) | PMS_REGION_PMS_CONSTRAIN_ADDR_5 | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0 | Reset |

**PMS_REGION_PMS_CONSTRAIN_ADDR_5** Configures the starting address of Region5 for CPU0. (R/WL)

Register 14.57. PMS_REGION_PMS_CONSTRAIN_9_REG (0x0128)

| (reserved) | PMS_REGION_PMS_CONSTRAIN_ADDR_6 | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0 | Reset |

**PMS_REGION_PMS_CONSTRAIN_ADDR_6** Configures the starting address of Region6 for CPU0. (R/WL)

Register 14.58. PMS_REGION_PMS_CONSTRAIN_10_REG (0x012C)

| (reserved) | PMS_REGION_PMS_CONSTRAIN_ADDR_7 | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0 | Reset |

**PMS_REGION_PMS_CONSTRAIN_ADDR_7** Configures the starting address of Region7 for CPU0. (R/WL)

Register 14.59. PMS_CORE_0_PIF_PMS_MONITOR_0_REG (0x0130)

| 31 | 1 | 0 |
|---|---|---|
| (reserved) | | PMS_CORE_0_PIF_PMS_MONITOR_LOCK |
| 0 0 0 0 0 0 0 0 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

**PMS_CORE_0_PIF_PMS_MONITOR_LOCK** Set this bit to lock CPU's PIF interrupt configuration. (R/WL)

Register 14.60. PMS_CORE_0_PIF_PMS_MONITOR_1_REG (0x0134)

| | (reserved) | PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_EN | PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_CLR | |
|---|---|---|---|---|
| | 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 0 0 0 0 0 0 0 0 0 | 1 | 1 | Reset |

**PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_CLR** Set this bit to clear the interrupt triggered when CPU's PIF bus tries to access RTC memory or peripherals unauthorized. (R/WL)

**PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_EN** Set this bit to enable interrupt when CPU's PIF bus tries to access RTC memory or peripherals unauthorized. (R/WL)

Register 14.61. PMS_CORE_0_PIF_PMS_MONITOR_4_REG (0x0140)

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_EN | PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_CLR | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Reset |

**PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_CLR** Set this bit to clear the interrupt triggered when CPU's PIF bus tries to access RTC memory or peripherals using unsupported data type. (R/WL)

**PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_EN** Set this bit to enable interrupt when CPU's PIF bus tries to access RTC memory or peripherals using unsupported data type. (R/WL)

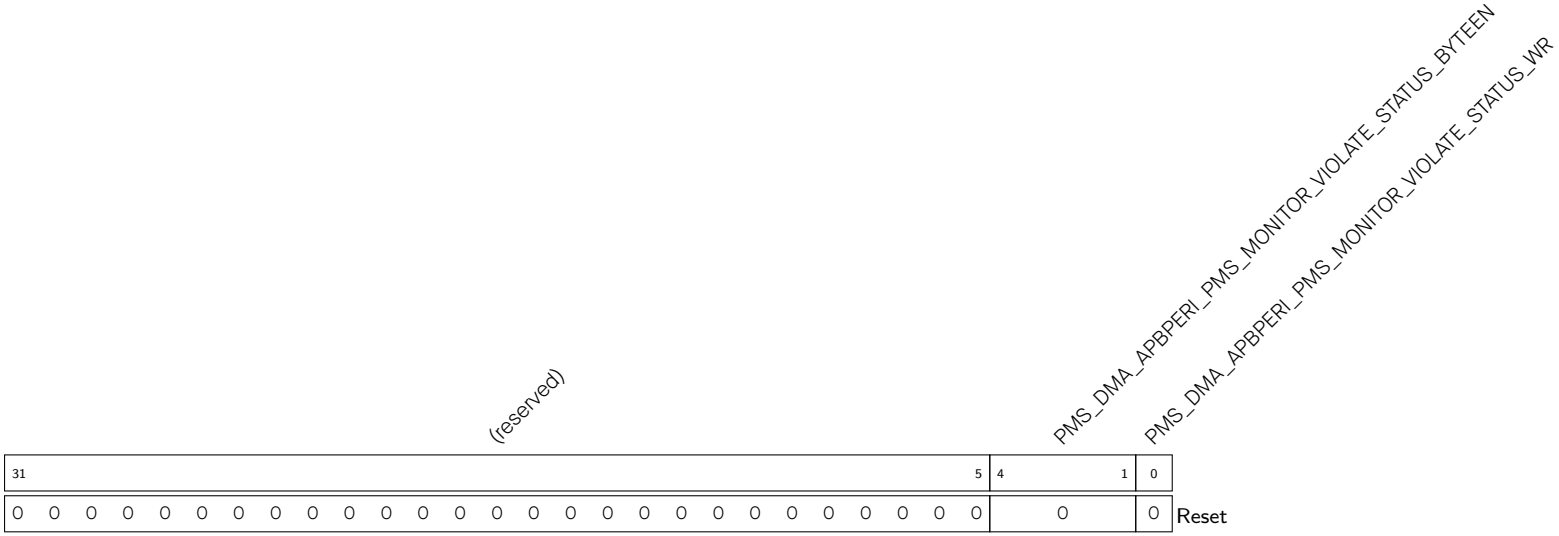
Register 14.62. PMS_DMA_APBPERI_PMS_MONITOR_2_REG (0x0088)

| (reserved) | PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_ADDR | (reserved) | PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR | |
|---|---|---|---|---|
| 31 27 | 26 3 | 2 1 | 0 | |
| 0 0 0 0 0 | 0 | 0 0 | 0 | Reset |

**PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_INTR** Stores unauthorized DMA access interrupt status. (RO)

**PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_ADDR** Stores the address that triggered the unauthorized DMA address. Note that this is an offset to 0x3c000000 and the unit is 16, which means the actual address should be 0x3c000000 + PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_ADDR * 16. (RO)

Register 14.63. PMS_DMA_APBPERI_PMS_MONITOR_3_REG (0x008C)




**PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_WR** Store the direction of unauthorized GDMA access. 1: write, 0: read. (RO)

**PMS_DMA_APBPERI_PMS_MONITOR_VIOLATE_STATUS_BYTEEN** Stores the byte information of unauthorized GDMA access. (RO)

Register 14.64. PMS_CORE_0_IRAM0_PMS_MONITOR_2_REG (0x00BC)

| (reserved) | PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR | PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_WORLD | PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_LOADSTORE | PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_WR | PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR | |
|---|---|---|---|---|---|---|
| 31 29 | 28 5 | 4 3 | 2 | 1 | 0 | |
| 0 0 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_INTR** Stores the interrupt status of CPU's unauthorized IBUS access. (RO)

**PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_WR** Indicates the access direction. 1: write, 0: read. Note that this field is only valid when PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_LOADSTORE is 1. (RO)

**PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_LOADSTORE** Indicates the instruction direction. 1: load/store, 0: instruction execution. (RO)

**PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_WORLD** Stores the privileged mode the CPU was in when the illegal access happened. 0x01: privilegedenvironment, 0x10: unpriviledged environment. (RO)

**PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR** Stores the address that CPU's IBUS was trying to access unauthorized. Note that this is an offset to 0x40000000 and the unit is 4, which means the actual address should be 0x40000000 + PMS_CORE_0_IRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR * 4. (RO)

Register 14.65. PMS_CORE_0_DRAM0_PMS_MONITOR_2_REG (0x00D0)

| (reserved) | PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR | PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_WORLD | (reserved) | PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR | |
|---|---|---|---|---|---|
| 31 28 | 27 4 | 3 2 | 1 | 0 | |
| 0 0 0 0 | 0 | 0 | 0 | 0 | Reset |

**PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_INTR** Stores the interrupt status of dBUS unauthorized access. (RO)

**PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_WORLD** Stores the privileged mode the CPU was in when the illegal access happened. 0x01: privilegedenvironment, 0x10: unpriviledged environment. (RO)

**PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR** Stores the address that CPU0's dBUS was trying to access unauthorized. Note that this is an offset to 0x3c000000 and the unit is 16, which means the actual address should be 0x3c000000 + PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_ADDR * 4. (RO)

Register 14.66. PMS_CORE_0_DRAM0_PMS_MONITOR_3_REG (0x00D4)

| (reserved) | PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_BYTEEN | PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_WR | |
|---|---|---|---|
| 31 … 5 | 4 … 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 |

**PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_WR** Stores the direction of unauthorized access. 0: read, 1: write. (RO)

**PMS_CORE_0_DRAM0_PMS_MONITOR_VIOLATE_STATUS_BYTEEN** Stores the byte information of illegal access. (RO)

**Register 14.67. PMS_CORE_0_PIF_PMS_MONITOR_2_REG (0x0138)**

| 31 – 8 | 7 – 6 | 5 | 4 – 2 | 1 | 0 | |
|---|---|---|---|---|---|---|
| (reserved) | | PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HWORLD | PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HWRITE | PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HSIZE | PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HPORT_0 | PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR |
| 0 0 0 0 0 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 |

**PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_INTR** Stores the interrupt status of PIF bus unauthorized access. (RO)

**PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HPORT_0** Stores the type of unauthorized access. 0: instruction. 1: data. (RO)

**PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HSIZE** Stores the data type of unauthorized access. 0: byte. 1: half-word. 2: word. (RO)

**PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HWRITE** Stores the direction of unauthorized access. 0: read. 1: write. (RO)

**PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HWORLD** Stores the privileged mode the CPU was in when the unauthorized access happened. 01: privileged environment 10: unpriviledged environment. (RO)

Register 14.68. PMS_CORE_0_PIF_PMS_MONITOR_3_REG (0x013C)

| PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HADDR | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**PMS_CORE_0_PIF_PMS_MONITOR_VIOLATE_STATUS_HADDR** Stores the address that CPU's PIF bus was trying to access unauthorized. (RO)

**Register 14.69. PMS_CORE_0_PIF_PMS_MONITOR_5_REG (0x0144)**

| 31 … 5 | 4 … 3 | 2 … 1 | 0 | |
|---|---|---|---|---|
| (reserved) | PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HWORLD | PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HSIZE | PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_INTR | |
| 0 0 0 0 0 0 0 0 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 |

**PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_INTR** Stores the interrupt status of PIF upsupported data type. (RO)

**PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HSIZE** Stores the data type when the unauthorized access happened. (RO)

**PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HWORLD** Stores the privileged mode the CPU was in when the unauthorized access happened. 01: privileged environment 10: unpriviledged environment. (RO)

Register 14.70. PMS_CORE_0_PIF_PMS_MONITOR_6_REG (0x0148)

| PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HADDR | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**PMS_CORE_0_PIF_PMS_MONITOR_NONWORD_VIOLATE_STATUS_HADDR** Stores the address that CPU's PIF bus was trying to access using unsupported data type. (RO)

Register 14.71. PMS_CLOCK_GATE_REG_REG (0x0170)

| (reserved) | PMS_CLK_EN | |
|---|---|---|
| 31 … 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 | 1 | Reset |

**PMS_CLK_EN** Set this bit to force the clock gating always on. (R/W)

### Register 14.72. PMS_DATE_REG (0x0FFC)

| (reserved) | PMS_DATE | |
|---|---|---|
| 31 … 28 | 27 … 0 | |
| 0 0 0 0 | 0x2010200 | Reset |

**PMS_DATE** Date register. (R/W)

### Register 14.73. SYSCON_EXT_MEM_PMS_LOCK_REG (0x0020)

| (reserved) | SYSCON_EXT_MEM_PMS_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**SYSCON_EXT_MEM_PMS_LOCK** Set this bit to lock the permission configuration related to external memory. (R/W)

**Register 14.74. SYSCON_FLASH_ACE*n*_ATTR_REG (*n*: 0 - 3) (0x0028 + 4**n*)**

| (reserved) | SYSCON_FLASH_ACE*n*_ATTR | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0xff | Reset |

**SYSCON_FLASH_ACE*n*_ATTR** Configures the permission to Region *n* of Flash. (R/W)

**Register 14.75. SYSCON_FLASH_ACE*n*_ADDR_REG (*n*: 0-3) (0x0038 + 4**n*)**

| SYSCON_FLASH_ACE*n*_ADDR_S | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**SYSCON_FLASH_ACE0_ADDR_S** Configure the starting address of Flash Region *n*. The size of each region should be aligned to 64 KB. (R/W)

**Register 14.76. SYSCON_FLASH_ACE*n*_SIZE_REG (*n*: 0-3) (0x0048 + 4**n* )**

| (reserved) | SYSCON_FLASH_ACE*n*_SIZE | |
|---|---|---|
| 31 … 16 | 15 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x1000 | Reset |

**SYSCON_FLASH_ACE*n*_SIZE** Configure the length of Flash Region *n*. The size of each region should be aligned to 64 KB. (R/W)

**Register 14.77. SYSCON_SPI_MEM_PMS_CTRL_REG (0x0088)**

| (reserved) | SYSCON_SPI_MEM_REJECT_CDE | SYSCON_SPI_MEM_REJECT_CLR | SYSCON_SPI_MEM_REJECT_INT | |
|---|---|---|---|---|
| 31 … 7 | 6 … 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 | 0x0 | 0 | 0 | Reset |

**SYSCON_SPI_MEM_REJECT_INT** Indicates exception accessing external memory and triggers an interrupt. (RO)

**SYSCON_SPI_MEM_REJECT_CLR** Set this bit to clear the exception status. (WT)

**SYSCON_SPI_MEM_REJECT_CDE** Stores the exception cause: invalid region, overlapping regions, illegal write, illegal read and illegal instruction execution. (RO)

### Register 14.78. SYSCON_SPI_MEM_REJECT_ADDR_REG (0x008C)

| SYSCON_SPI_MEM_REJECT_ADDR | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**SYSCON_SPI_MEM_REJECT_ADDR** Store the execption address.(RO)

### Register 14.79. EXTMEM_IBUS_PMS_TBL_LOCK_REG (0x00D8)

| (reserved) | EXTMEM_IBUS_PMS_LOCK | |
|---|---|---|
| 31 … 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 | Reset |

**EXTMEM_IBUS_PMS_LOCK** Set this bit to lock IBUS' access to Cache IBUS regions. (R/W)

### Register 14.80. EXTMEM_IBUS_PMS_TBL_BOUNDARY0_REG (0x00DC)

| (reserved) | EXTMEM_IBUS_PMS_BOUNDARY0 | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**EXTMEM_IBUS_PMS_BOUNDARY0** Configures the starting address of Cache IBUS Region1. (R/W)

### Register 14.81. EXTMEM_IBUS_PMS_TBL_BOUNDARY1_REG (0x00E0)

| (reserved) | EXTMEM_IBUS_PMS_BOUNDARY1 | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x800 | Reset |

**EXTMEM_IBUS_PMS_BOUNDARY1** Configures the starting address of Cache IBUS Region2.(R/W)

### Register 14.82. EXTMEM_IBUS_PMS_TBL_BOUNDARY2_REG (0x00E4)

| (reserved) | EXTMEM_IBUS_PMS_BOUNDARY2 | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x800 | Reset |

**EXTMEM_IBUS_PMS_BOUNDARY2** Configures the starting address of Cache IBUS Region3. (R/W)

**Register 14.83. EXTMEM_IBUS_PMS_TBL_ATTR_REG (0x00E8)**

| (reserved) | EXTMEM_IBUS_PMS_SCT2_ATTR | EXTMEM_IBUS_PMS_SCT1_ATTR | |
|---|---|---|---|
| 31 … 8 | 7 … 4 | 3 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0xf | 0xf | Reset |

**EXTMEM_IBUS_PMS_SCT1_ATTR** Configures IBUS' access to Cache IBUS Region1.

Bit 0: Instruction execution access in the privileged environment

Bit 1: Read access in the privileged environment

Bit 2: Instruction execution access in the unprivileged environment

Bit 3: Read access in the unprivileged environment

(R/W)

**EXTMEM_IBUS_PMS_SCT2_ATTR** Configures IBUS' access to Cache IBUS Region2.

Bit 0: Instruction execution access in the privileged environment

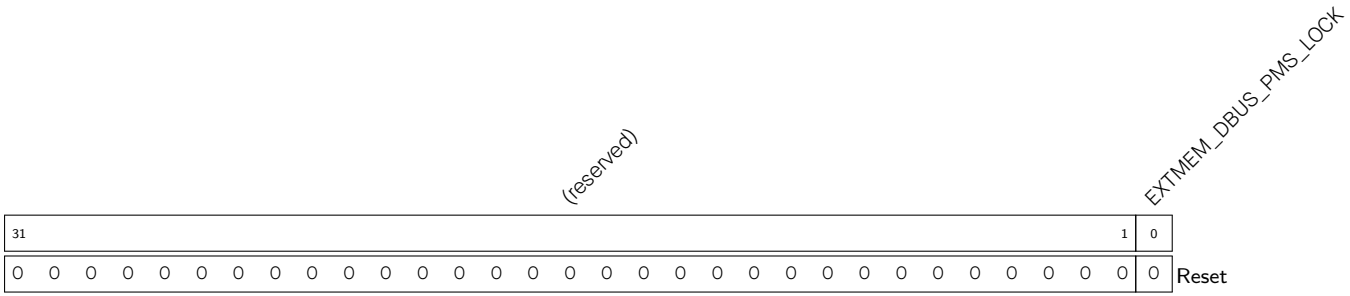
Bit 1: Read access in the privileged environment

Bit 2: Instruction execution access in the unprivileged environment

Bit 3: Read access in the unprivileged environment

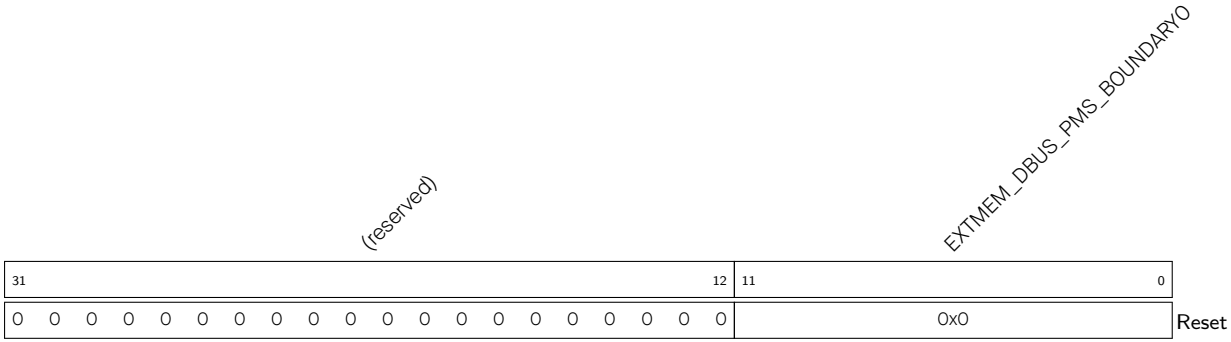
(R/W)

### Register 14.84. EXTMEM_DBUS_PMS_TBL_LOCK_REG (0x00EC)




**EXTMEM_DBUS_PMS_LOCK** Set this bit to lock DBUS' access to Cache DBUS regions. (R/W)

### Register 14.85. EXTMEM_DBUS_PMS_TBL_BOUNDARY0_REG (0x00F0)




**EXTMEM_DBUS_PMS_BOUNDARY0** Configures the starting address of Cache DBUS Region1. (R/W)

**Register 14.86. EXTMEM_DBUS_PMS_TBL_BOUNDARY1_REG (0x00F4)**

| (reserved) | EXTMEM_DBUS_PMS_BOUNDARY1 | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x800 | Reset |

**EXTMEM_DBUS_PMS_BOUNDARY1** Configures the starting address of Cache DBUS Region2. (R/W)

**Register 14.87. EXTMEM_DBUS_PMS_TBL_BOUNDARY2_REG (0x00F8)**

| (reserved) | EXTMEM_DBUS_PMS_BOUNDARY2 | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x800 | Reset |

**EXTMEM_DBUS_PMS_BOUNDARY2** Configures the starting address of Cache DBUS Region3. (R/W)

**Register 14.88. EXTMEM_DBUS_PMS_TBL_ATTR_REG (0x00FC)**

| (reserved) | EXTMEM_DBUS_PMS_SCT2_ATTR | EXTMEM_DBUS_PMS_SCT1_ATTR | |
|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 | 3 | 3 | Reset |

**EXTMEM_DBUS_PMS_SCT1_ATTR** Configures DBUS' access to Cache DBUS Region1.

Bit 0: Read access in the privileged environment

Bit 1: Read access in the unprivileged environment

(R/W)

**EXTMEM_DBUS_PMS_SCT2_ATTR** Configures DBUS' access to Cache DBUS Region2.

Bit 0: Read access in the privileged environment

Bit 1: Read access in the unprivileged environment

(R/W)







GoBack

# Chapter 15

# World Controller (WCL)

## 15.1 Introduction

ESP32-C3 allows users to allocate its hardware and software resources into Secure World (World0) and Non-secure World (World1), thus protecting resources from unauthorized access (read or write), and from malicious attacks such as malware, hardware-based monitoring, hardware-level intervention, and so on. CPUs can switch between Secure World and Non-secure World with the help of the World Controller.

By default, all resources in ESP32-C3 are shareable. Users can allocate the resources into two worlds by managing respective permission (For details, please refer to Chapter 14 *Permission Control (PMS)*). This chapter only introduces the World Controller and how CPUs can switch between worlds with the help of World Controller.

## 15.2 Features

ESP32-C3's World Controller:

- Controls the CPUs to switch between the Secure World and Non-secure World
- Logs CPU's world switches

## 15.3 Functional Description

With the help of World Controller, we can allocate different resources to the Secure World and the Non-secure World:

- Secure World (World0):
  - Can access all peripherals and memories;
  - Performs all security related operations, such user authentication, secure communication, and data encryption and decryption, etc.
- Non-secure World (World1):
  - Can access some peripherals and memories;
  - Performs other operations, such as user operation and different applications, etc.

ESP32-C3's CPU and slave devices are both configurable with permission to either Secure World and/or Non-Secure World:

- CPU can be in either world at a particular time:

  - In Secure World: performs confidential operations;
  - In Non-secure World: performs non-confidential operations;
  - By default, CPU runs in Secure World after power-up, then can be programmed to switch between two worlds.

- All slave devices (including peripherals* and memories) can be configured to be accessible from the Secure World and/or the Non-secure World:
  - Secure World Access: this slave can be called from Secure World only, meaning it can be accessed only when CPU is in Secure World;
  - Non-secure World Access: this slave can be called from Non-secure World only, meaning it can be accessed only when CPU is in Non-secure World.
  - Note that a slave can be configured to be accessible from both Secure World and Non-secure World simultaneously.

  For details, please refer to Chapter 14 *Permission Control (PMS)*.

**Note:**
* World Controller itself is a peripheral, meaning it also can be granted with Secure World access and/or Non-secure World access, just like all other peripherals. However, to secure the world switch mechanism, World Controller should not be accessible from Non-secure world. Therefore, world controller **should not be granted with** Non-secure World access, preventing any modification to world controller from the Non-secure World.

**When CPU accesses any slaves:**

1. First, CPU notifies the slave about its own world information;
2. Second, slave checks if it can be accessed by CPU based on the CPU's world information and its own world permission configuration.
   - if allowed, then this slave responds to CPU;
   - if not allowed, then this slave will not respond to CPU and trigger an interrupt.

In this way, the resources in the Secure World will not be illegally accessible by the Non-secure World in an unauthorized way.

Note that the following CPU interrupt-related CSR registers can only be written to in the Secure World, and can only be read but not written to in the Non-secure World, thus ensuring that interrupts can only be controlled by the Secure World.

| Name | Description | Address | Access |
|---|---|---|---|
| **Machine Trap Setup CSRs** | | | |
| mstatus | Machine Mode Status | 0x300 | R/W |
| mtvec | Machine Trap Vector | 0x305 | R/W |
| **Machine Trap Handling CSRs** | | | |
| mscratch | Machine Scratch | 0x340 | R/W |
| mepc | Machine Trap Program Counter | 0x341 | R/W |
| mcause | Machine Trap Cause | 0x342 | R/W |
| mtval | Machine Trap Value | 0x343 | R/W |

## 15.4 CPU's World Switch

CPU can switch from Secure World to Non-secure World, and from Non-secure World to Secure World.

### 15.4.1 From Secure World to Non-secure World

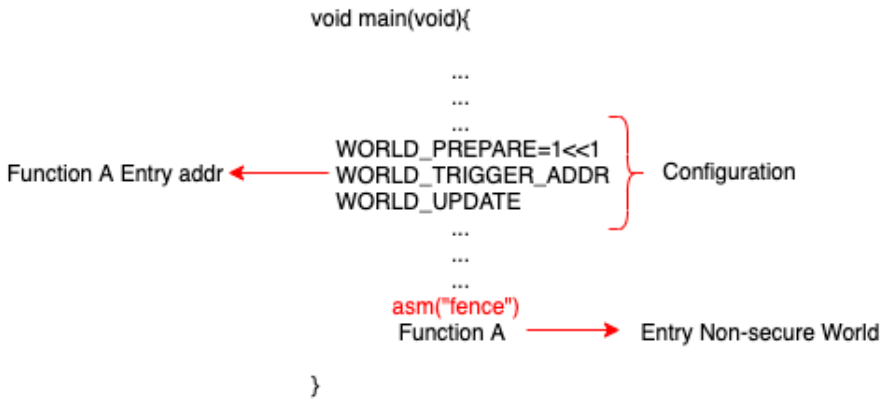



Figure 15.4-1. Switching From Secure World to Non-secure World

ESP32-C3's CPU only needs to complete the following steps to switch from Secure World to Non-secure World:

1. Write 0x2 to Register WCL_CORE_0_WORLD_PERPARE_REG, indicating the CPU needs to switch to the Non-secure World.
2. Configure Register WCL_CORE_0_World_TRIGGER_ADDR_REG as the entry address to the Non-secure World, i.e., the address of the application in the Non-secure World that needs to be executed.
3. Write any value to Register WCL_CORE_0_World_UPDATE_REG, indicating the configuration is done.

**Note:**

- Registers WCL_CORE*m*_WORLD_PERPARE_REG and WCL_CORE_0_World_TRIGGER_ADDR_REG can be configured in any order. Register WCL_CORE_0_World_UPDATE_REG must be configured at last.

Afterwards, the World Controller keeps monitoring if CPU is executing the configured address of the application in Non-secure World. CPU switches to the Non-secure World once it executes the configured address, and executes the applications in the Non-secure World.

After configuration, the World Controller:

- Keeps monitoring until the CPU executes the configured address and switches to the Non-secure World.
  - Write any value to Register WCL_CORE_0_World_Cancel_REG to cancel the World Controller configuration. After the cancellation, CPU will not switch to the Non-secure World even it executes to the configured address.
- The World Controller can only switch from the Secure World to Non-secure World once per configuration. Therefore, the World Controller needs to be configured again after each world switch to

prepare it for the next world switch.

However, it's worth noting that you cannot call the application in Non-secure world immediately after configuring the World Controller. For reasons such as CPU pre-indexed addressing and pipeline, it is possible that the CPU has already executed the application in Non-secure World before the World Controller configuration is effective, meaning the CPU runs unsecured application in the Secure World.

Therefore, you need to make sure the CPU only calls applications in the Non-secure world after the World Controller configuration takes effect. This can be guaranteed by **declaring the applications in the Non-secure World as "noinline"**.

## 15.4.2 From Non-secure World to Secure World

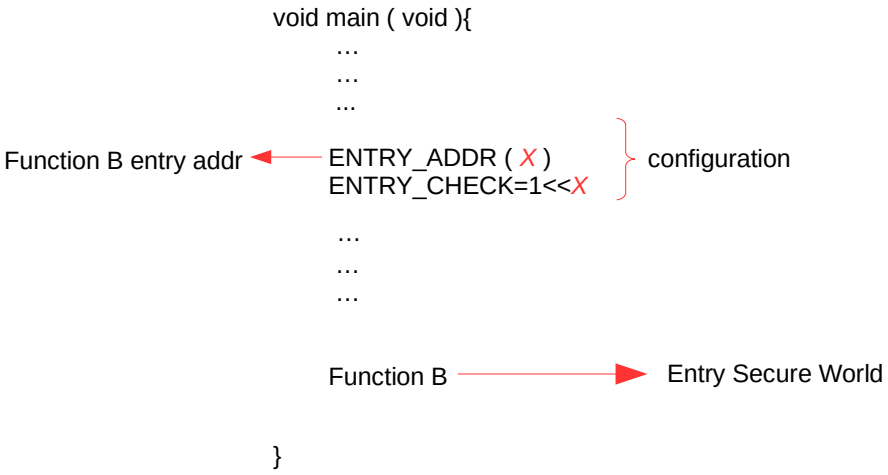



**Figure 15.4-2. Switching From Non-secure World to Secure World**

CPU can only switch from Non-secure World to Secure World via **Interrupts (or Exceptions)**. After configuring the World Controller, the CPU can switch back from Non-secure World to Secure World upon the configured Interrupt trigger.

#### Configuring the World Controller

The detailed steps to configure the World Controller to switch the CPU from Non-secure World to Secure World are described below:

1. Configure the entry base address of interrupts or exception WCL_CORE_0_MTVEC_BASE_REG. After that, the World controller populates the monitored addresses for each entry as follows:
   - Exception entry: WCL_CORE_0_MTVEC_BASE_REG + 0x00
   - Interrupt entries: WCL_CORE_0_MTVEC_BASE_REG + 4* *i* (*i* = 1~31)

   Note that this register must be configured to the mtvec CSR register of the CPU. When modifying the CPU's mtvec CSR registers, this register also must be updated. For details, please refer to Chapter 1 *ESP-RISC-V CPU*.

2. Configure Register WCL_CORE_0_ENTRY_CHECK_REG to enable the monitoring of one or more certain entries (0: disable; 1: enable).
   - Bit 0 controls the entry monitoring of exception
   - Bit *x* controls the entry monitoring of interrupt Entry *x* (*x* = 1~31), respectively

   Note that, once configured, register WCL_CORE_0_ENTRY_CHECK_REG is always effective till it's disabled again, meaning you don't need to configure this register every time after each world switch.
3. Configure WCL_CORE_0_MSTATUS_MIE_REG to enable updating the World Switch Log. Otherwise, this log will not be updated for world switches. For detailed information about the World Switch Log, see Section 15.5.

## 15.5 World Switch Log

In actual use cases, CPU is switching between two worlds quite frequently and has to deal with nested interrupts. To be able to restore to the previous world, World Controller keeps a world switching log in a series of registers, which is called "World Switch Log Table".

### 15.5.1 Structure of World Switch Log Register

ESP32-C3's World Switch Log Table consists of 32 WCL_CORE_0_STATUSTABLE*n*_REG(*n*: 0-31) registers (see Figure 15.5-1). The Entry *x*, is logged in WCL_CORE_0_STATUSTABLE*x*_REG.

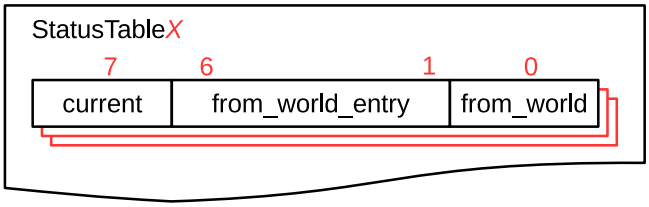



**Figure 15.5-1. World Switch Log Register**

- WCL_CORE_0_FROM_WORLD_*n*: logs the world information before the world switch.
  - 0: CPU was in Secure World
  - 1: CPU was in Non-secure World
- WCL_CORE_0_FROM_ENTRY_*n*: logs the entry information before the world switch, in total of 6 bits.
  - 0~31: CPU is currently jumping from another interrupt/exception entry 0~31
  - 32: CPU was not at any interrupts monitored at any entry
- WCL_CORE_0_CURRENT_*n*: indicates if CPU is at the interrupt monitored at the current entry. When CPU is at the interrupt monitored at Entry *x*,
  - WCL_CORE_0_CURRENT_*x* is updated to 1;
  - and the same field of all other entries are updated to 0.

### 15.5.2 How World Switch Log Registers are Updated

To explain this process, assuming:

1. At the beginning:
   - CPU is running in the Non-secure World;
   - Registers WCL_CORE_0_STATUSTABLE*n*_REG(*n*: 0-31) are all empty.
2. Then an interrupt occurs at Entry 9;
3. Then another interrupt with higher priority occurs at Entry 1;
4. Then the last interrupt with highest priority occurs at Entry 4.

The World Switch Log Table is updated as described below:

1. First, an interrupt occurs at Entry 9. At this time, CPU executes to the entry address of this interrupt. The World Switch Log Table is updated as described in Figure 15.5-2:

| entry | current | from_entry | from_world |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 |
| 9 | 1 | 32 | 1 |
| ... | 0 | 0 | 0 |
| 30 | 0 | 0 | 0 |
| 31 | 0 | 0 | 0 |

Figure 15.5-2. Nested Interrupts Handling - Entry 9

   At this time:

   - WCL_CORE_0_STATUSTABLE9_REG
     - Field WCL_CORE_0_FROM_WORLD_9 is updated to 1, indicating CPU was in Non-secure World before the interrupt;
     - Field WCL_CORE_0_FROM_ENTRY_9 is updated to 32, indicating there was not any interrupt before this one;
     - Field WCL_CORE_0_CURRENT_9 is updated to 1, indicating the CPU is currently at the interrupt monitored at Entry 9.
   - Other WCL_CORE_0_STATUSTABLE*n*_REG registers are not updated.

2. Then another interrupt with higher priority occurs at Entry 1. At this time, CPU executes to the entry address of this interrupt. The World Switch Log Table is updated again as described in Figure 15.5-3:

| entry | current | from_entry | from_world |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 9 | 0 |
| 2 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 |
| 9 | 0 | 32 | 1 |
| ... | 0 | 0 | 0 |
| 30 | 0 | 0 | 0 |
| 31 | 0 | 0 | 0 |

Figure 15.5-3. Nested Interrupts Handling - Entry 1

At this time:

- WCL_CORE_0_STATUSTABLE1_REG
  - Field WCL_CORE_0_FROM_WORLD_1 is updated to 0, indicating the CPU was in Secure World before this interrupt.
  - Field WCL_CORE_0_FROM_ENTRY_1 is updated to 9, indicating the CPU was executing the interrupt at Entry 9.
  - Field WCL_CORE_0_CURRENT_1 is updated to 1, indicating CPU is currently at the interrupt monitored at Entry 1.
- WCL_CORE_0_STATUSTABLE9_REG
  - Field WCL_CORE_0_CURRENT_9 is updated to 0, indicating CPU is no longer at the interrupt monitored at Entry 9 (Instead, CPU is at the interrupt monitored at Entry 1 already).
  - Fields WCL_CORE_0_FROM_WORLD_9 and WCL_CORE_0_FROM_ENTRY_9 stay the same.
- Other WCL_CORE_0_STATUSTABLE*n*_REG registers are not updated.

3. Then the last interrupt with highest priority occurs at Entry 4. At this time, CPU executes to the entry address of interrupt 4. The World Switch Log Table is updated again as described in Figure 15.5-4:

| entry | current | from_entry | from_world |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 9 | 0 |
| 2 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 |
| 4 | 1 | 1 | 0 |
| 5 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 |
| 9 | 0 | 32 | 1 |
| ... | 0 | 0 | 0 |
| 30 | 0 | 0 | 0 |
| 31 | 0 | 0 | 0 |

Figure 15.5-4. Nested Interrupts Handling - Entry 4

At this time:

- WCL_CORE_0_STATUSTABLE4_REG

  - Field WCL_CORE_0_FROM_WORLD_4 is updated to 0, indicating the CPU was in Secure World before this interrupt.
  - Field WCL_CORE_0_FROM_ENTRY_4 is updated to 1, indicating the CPU was the interrupt at Entry 1.
  - Field WCL_CORE_0_CURRENT_4 is updated to 1, indicating the CPU is currently at the interrupt monitored an Entry 4.
- WCL_CORE_0_STATUSTABLE1_REG
  - Field WCL_CORE_0_CURRENT_1 is updated to 0, indicating the CPU is no longer at the interrupt monitored at Entry 1 (Instead CPU is at the interrupt monitored at Entry 4 already).
  - Fields WCL_CORE_0_FROM_WORLD_1 and WCL_CORE_0_FROM_ENTRY_1 are not updated.
- Other WCL_CORE_0_STATUSTABLE*n*_REG registers are not updated.

### 15.5.3 How to Read World Switch Log Registers

By reading World Switch Log Registers, we get to understand the information of previous world switches and nested interrupts, thus being able to restore to previous world.

Steps are described below: (See Figure 15.5-4 as an example):

1. Read Register WCL_CORE_0_STATUSTABLE_CURRENT_REG, and understand CPU is now at the interrupt monitored at Entry 4.
2. Read 1 from Field WCL_CORE_0_FROM_ENTRY_4, and understand the CPU was at an interrupt monitored at Entry 1.
3. Read 9 from Field WCL_CORE_0_FROM_ENTRY_1, and understand the CPU was at an interrupt monitored at Entry 9.
4. Read 32 from WCL_CORE_0_FROM_ENTRY_9, and understand CPU wasn't at any interrupt. Then read 1 from WCL_CORE_0_FROM_WORLD_9, and understand CPU was in Non-secure World at the beginning.

### 15.5.4 Nested Interrupts

To support interrupt nesting, World controller provides additional configuration to update World Switch Log. See details in Section Programming Procedure below.

#### 15.5.4.1 Programming Procedure

**Handling the interrupt at Entry *A*:**

1. Save context.
2. Configure WCL_CORE_0_MSTATUS_MIE_REG register to enable updating the World Switch Log table.
   - After entering the interrupt and exception vector, CPU will automatically turn off the global interrupt enable to avoid interrupt nesting. After saving the context, the global interrupt enable can be turned on again to respond to higher-level interrupts.
   - The World Controller WCL_CORE_0_MSTATUS_MIE_REG register also supports a similar feature of global interrupt enable. When any entry trigger is detected, WCL_CORE_0_MSTATUS_MIE_REG will

be automatically cleared to 0, and software needs to be enabled again in the interrupt/exception service routine. This register should be configured before turning on the global interrupt enable.

3. Enable the global interrupt enable.
4. Execute the interrupt programs.
5. Disable CPU global interrupt enable.
6. Read Field WCL_CORE_0_FROM_ENTRY_*A* for Entry *A*:
   - 32: indicates all interrupts are handled, and return to a normal program,
     (a) Update Field WCL_CORE_0_CURRENT_*A* of Entry *A* to 0, indicating the CPU is no longer at the interrupt monitored at Entry *A*.
     (b) Go to Step 7.
   - 0~31: indicates the CPU returns to another interrupt monitored at Entry *B*,
     - Update the world switch register of Entry *A*:
       * Update Field WCL_CORE_0_CURRENT_*A* to 0, indicating the CPU is no longer at the interrupt monitored at Entry *A*.
       * Fields WCL_CORE_0_FROM_WORLD_*A* and WCL_CORE_0_FROM_ENTRY_*A* stay the same.
     - Update the world switch register of Entry *B*:
       * Update Field WCL_CORE_0_CURRENT_*B* to 1, indicating the CPU will return to Entry *B*.
7. Prepare to exit interrupt.
   (a) Check if CPU needs to switch to the other world:
   - If world switch not required, then go to Step 8.
   - If world switch required, then switch the CPU to the other world following instructions described in Section 15.4, then go to Step 8.
8. Enable interrupts, restore context and exit.

**Note:**
Steps 6 and 7 should not be interrupted by any interrupts. Therefore, users need to disable all the interrupts before these steps, and enable interrupts once done.

## 15.6 Register Summary

The addresses in this section are relative to the World Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

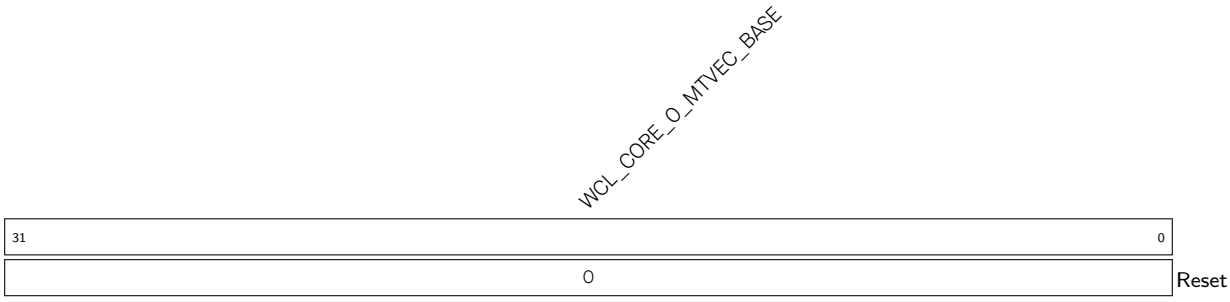
The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **WORLD1 to WORLD0 Configuration Registers** | | | |
| WCL_Core_0_MTVEC_BASE_REG | MTVEC configuration | 0x0000 | R/W |
| WCL_Core_0_MSTATUS_MIE_REG | MSTATUS_MIE configuration | 0x0004 | R/W |
| WCL_Core_0_ENTRY_CHECK_REG | CPU entry check configuration | 0x0008 | R/W |
| **StatusTable Registers** | | | |
| WCL_Core_0_STATUSTABLE*n*_REG (*n*: 0-31) | Entry *n* world switching status | 0x0040 | R/W |
| WCL_Core_0_STATUSTABLE_CURRENT_REG | Represetns the entry where the interrupt is currently at | 0x00E0 | R/W |
| **WORLD0 to WORLD1 Configuration Registers** | | | |
| WCL_Core_0_World_TRIGGER_ADDR_REG | CPU trigger address configuration | 0x0140 | RW |
| WCL_Core_0_World_PREPARE_REG | CPU world switching preparation configuration | 0x0144 | R/W |
| WCL_Core_0_World_UPDATE_REG | CPU world switching update configuration | 0x0148 | WO |
| WCL_Core_0_World_Cancel_REG | CPU world switching cancel configuration | 0x014C | WO |
| WCL_Core_0_World_IRam0_REG | CPU IBUS world info | 0x0150 | R/W |
| WCL_Core_0_World_DRam0_PIF_REG | CPU DBUS and PIF bus world info | 0x0154 | R/W |
| WCL_Core_0_World_Phase_REG | CPU world switching readiness | 0x0158 | RO |

## 15.7 Registers

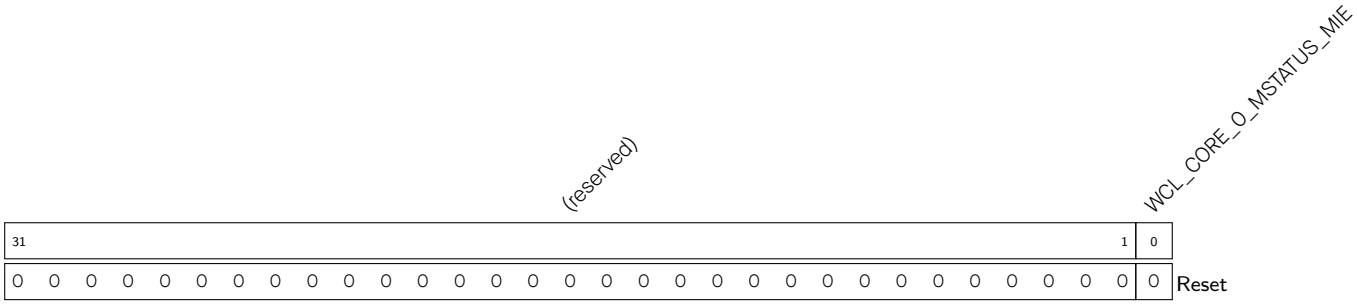
The addresses in this section are relative to the World Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 15.1. WCL_Core_0_MTVEC_BASE_REG (0x0000)**



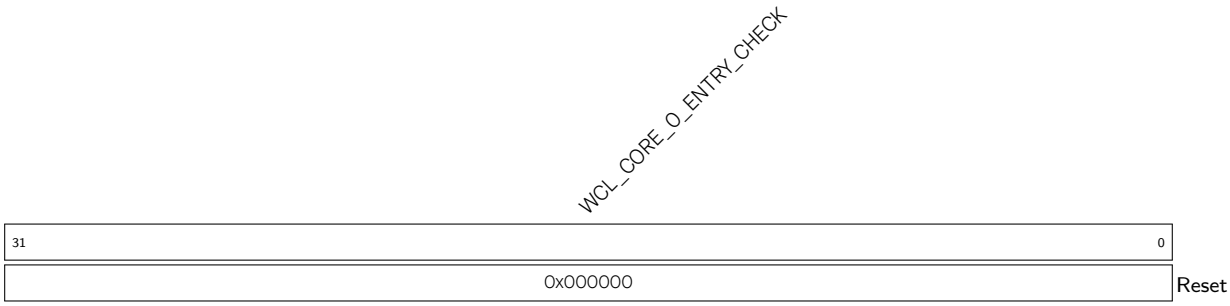

**WCL_CORE_0_MTVEC_BASE** Configures the MTVEC base address, which should be kept consistent with the MTVEC in RISC-V. (R/W)

**Register 15.2. WCL_Core_0_MSTATUS_MIE_REG (0x0004)**




**WCL_CORE_0_MSTATUS_MIE** Write 1 to enable World Switch Log Table. Only when the bit is set, the world switching is recorded in the World Switch Log Table. This bit is cleared once CPU switches from the Non-secure World to Secure World. (R/W)

**Register 15.3. WCL_Core_0_ENTRY_CHECK_REG (0x0008)**




**WCL_CORE_0_ENTRY_CHECK** Write 1 to enable CPU switching from Non-secure World to Secure world upon the monitored addresses. (R/W)

**Register 15.4. WCL_Core_0_STATUSTABLE*n*_REG(*n*: 0-31) (0x0x0040+4*n)**

| (reserved) | WCL_CORE_0_CURRENT_*n* | WCL_CORE_0_FROM_ENTRY_*n* | WCL_CORE_0_FROM_WORLD_*n* | |
|---|---|---|---|---|
| 31 … 8 | 7 | 6 … 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 | 0x0 | 0 | Reset |

**WCL_CORE_0_FROM_WORLD_*n*** Stores the world info before CPU entering entry *n*. (R/W)

**WCL_CORE_0_FROM_ENTRY_*n*** Stores the previous entry info before CPU entering entry *n*.(R/W)

**WCL_CORE_0_CURRENT_*n*** Represents if the interrupt is at entry *n*. (R/W)

**Register 15.5. WCL_Core_0_STATUSTABLE_CURRENT_REG (0x00E0)**

| WCL_CORE_0_STATUSTABLE_CURRENT | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

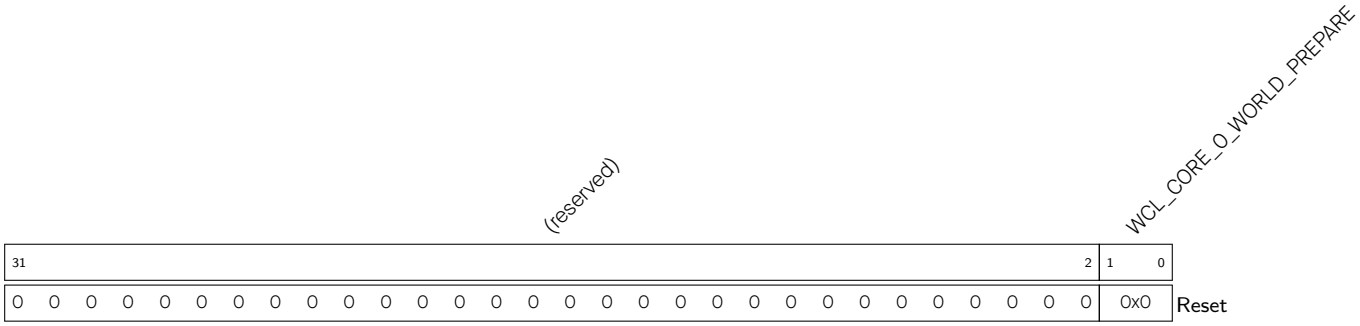
**WCL_CORE_0_STATUSTABLE_CURRENT** Represents the entry where the interrupt is currently at. (R/W)

**Register 15.6. WCL_Core_0_World_TRIGGER_ADDR_REG (0x0140)**

| WCL_CORE_0_WORLD_TRIGGER_ADDR | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**WCL_CORE_0_WORLD_TRIGGER_ADDR** Configures the entry address at which CPU switches from Secure World to Non-secure World. (RW)

**Register 15.7. WCL_Core_0_World_PREPARE_REG (0x0144)**



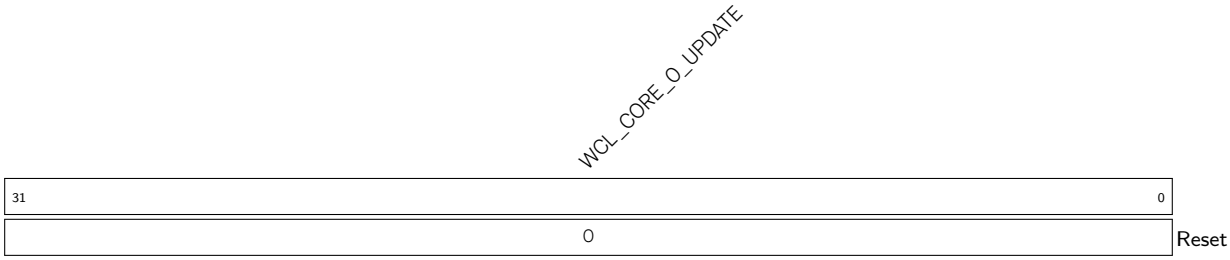

**WCL_CORE_0_WORLD_PREPARE** Configures the world to switch to.

0x1: reserved

0x2: Non-secure World

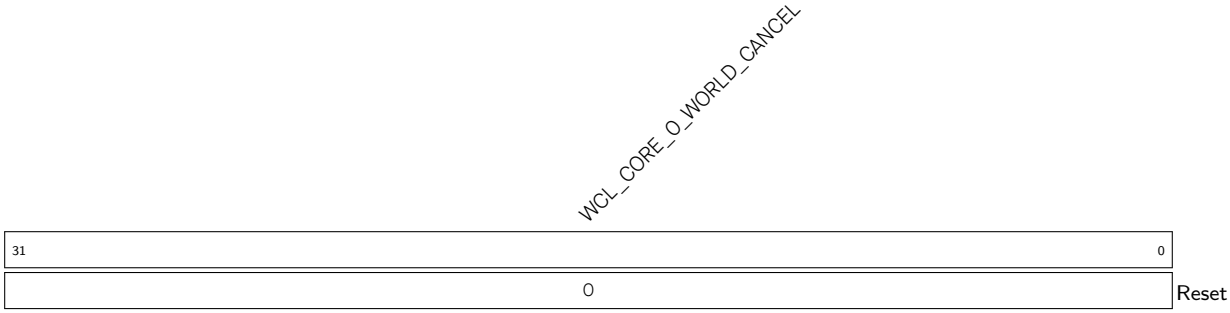
(R/W)

**Register 15.8. WCL_Core_0_World_UPDATE_REG (0x0148)**




**WCL_CORE_0_UPDATE** Write any value to this field to indicate the completion of CPU configuration for switching from Secure World to Non-Secure World. (WO)

**Register 15.9. WCL_Core_0_World_Cancel_REG (0x014C)**




**WCL_CORE_0_WORLD_CANCEL** Write any value to this filed to cancel the CPU configuration for switching from Secure World to Non-Secure World. (WO)

### Register 15.10. WCL_Core_0_World_IRam0_REG (0x0150)

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | WCL_CORE_0_WORLD_IRAM0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**WCL_CORE_0_WORLD_IRAM0** Stores the world info of CPU's instruction bus. Only used for debugging. (R/W)

### Register 15.11. WCL_Core_0_World_DRam0_PIF_REG (0x0154)

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | WCL_CORE_0_WORLD_DRAM0_PIF | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**WCL_CORE_0_WORLD_DRAM0_PIF** Stores the world info of CPU's data bus and peripheral bus. Only used for debugging. (R/W)

### Register 15.12. WCL_Core_0_World_Phase_REG (0x0158)

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | WCL_CORE_0_WORLD_PHASE |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**WCL_CORE_0_WORLD_PHASE** Represents if the CPU is ready to switch from Non-secure World to Secure World.

1: ready

2: not ready

(RO)

# Chapter 16

# System Registers (SYSREG)

## 16.1 Overview

The ESP32-C3 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-C3 has various 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.

## 16.2 Features

ESP32-C3 system registers can be used to control the following peripheral blocks and core modules:

- System and memory
- Clock
- Software Interrupt
- Low-power management
- Peripheral clock gating and reset

## 16.3 Function Description

### 16.3.1 System and Memory Registers

#### 16.3.1.1 Internal Memory

The following registers can be used to control ESP32-C3's internal memory:

- In register SYSCON_CLKGATE_FORCE_ON_REG:
  - Setting different bits of the SYSCON_ROM_CLKGATE_FORCE_ON field forces on the clock gates of different blocks of Internal ROM 0 and Internal ROM 1.
  - Setting different bits of the SYSCON_SRAM_CLKGATE_FORCE_ON field forces on the clock gates of different blocks of Internal SRAM.
  - This means when the respective bits of this register are set to 1, the clock gate of the corresponding ROM or SRAM blocks will always be on. Otherwise, the clock gate will turn on automatically when the corresponding ROM or SRAM blocks are accessed and turn off

automatically when the corresponding ROM or SRAM blocks are not accessed. Therefore, it's recommended to configure these bits to 0 to lower power consumption.

- In register SYSCON_MEM_POWER_DOWN_REG:
  - Setting different bits of the SYSCON_ROM_POWER_DOWN field sends different blocks of Internal ROM 0 and Internal ROM 1 into retention state.
  - Setting different bits of the SYSCON_SRAM_POWER_DOWN field sends different blocks of Internal SRAM into retention state.
  - The "Retention" state is a low-power state of a memory block. In this state, the memory block still holds all the data stored but cannot be accessed, thus reducing the power consumption. Therefore, you can send a certain block of memory into the retention state to reduce power consumption if you know you are not going to use such memory block for some time.
- In register SYSCON_MEM_POWER_UP_REG:
  - By default, all memory enters low-power state when the chip enters the Light-sleep mode.
  - Setting different bits of the SYSCON_ROM_POWER_UP field forces different blocks of Internal ROM 0 and Internal ROM 1 to work as normal (do not enter the retention state) when the chip enters Light-sleep.
  - Setting different bits of the SYSCON_SRAM_POWER_UP field forces different blocks of Internal SRAM to work as normal (do not enter the retention state) when the chip enters Light-sleep.

For detailed information about the controlling bits of different blocks, please see Table 16.3-1 below.

**Table 16.3-1. Memory Controlling Bit**

| Memory | Lowest Address1 | Highest Address1 | Lowest Address2 | Highest Address2 | Controlling Bit |
|---|---|---|---|---|---|
| ROM 0 | 0x4000_0000 | 0x4003_FFFF | - | - | Bit0 |
| ROM 1 | 0x4004_0000 | 0x4005_FFFF | 0x3FF0_0000 | 0x3FF1_FFFF | Bit1 |
| SRAM Block 0 | 0x4037_C000 | 0x4037_FFFF | - | - | Bit0 |
| SRAM Block 1 | 0x4038_0000 | 0x4039_FFFF | 0x3FC8_0000 | 0x3FC9_FFFF | Bit1 |
| SRAM Block 2 | 0x403A_0000 | 0x403B_FFFF | 0x3FCA_0000 | 0x3FCB_FFFF | Bit2 |
| SRAM Block 3 | 0x403C_0000 | 0x403D_FFFF | 0x3FCC_0000 | 0x3FCD_FFFF | Bit3 |

For more information, please refer to Chapter 3 *System and Memory*.

#### 16.3.1.2 External Memory

SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG configures encryption and decryption options of the external memory. For details, please refer to Chapter 23 *External Memory Encryption and Decryption (XTS_AES)*.

#### 16.3.1.3 RSA Memory

SYSTEM_RSA_PD_CTRL_REG controls the SRAM memory in the RSA accelerator.

- Setting the SYSTEM_RSA_MEM_PD bit to send the RSA memory into retention state. This bit has the lowest priority, meaning it can be masked by the SYSTEM_RSA_MEM_FORCE_PU field. This bit is invalid when the *Digital Signature (DS)* occupies the RSA.

- Setting the SYSTEM_RSA_MEM_FORCE_PU bit to force the RSA memory to work as normal when the chip enters light sleep. This bit has the second highest priority, meaning it overrides the SYSTEM_RSA_MEM_PD field.
- Setting the SYSTEM_RSA_MEM_FORCE_PD bit to send the RSA memory into retention state. This bit has the highest priority, meaning it sends the RSA memory into retention state regardless of the SYSTEM_RSA_MEM_FORCE_PU field.

### 16.3.2 Clock Registers

The following registers are used to set clock sources and frequency. For more information, please refer to Chapter 6 *Reset and Clock*.

- SYSTEM_CPU_PER_CONF_REG
- SYSTEM_SYSCLK_CONF_REG
- SYSTEM_BT_LPCK_DIV_FRAC_REG

### 16.3.3 Interrupt Signal Registers

The following registers are used for generating the interrupt signals (software interrupt), which then can be routed to the CPU peripheral interrupts via the interrupt matrix. To be more specific, writing 1 to any of the following registers generates an interrupt signal. Therefore, these registers can be used by software to control interrupts. The following registers correspond to the interrupt source SW_INTR_0/1/2/3. For more information, please refer to Chapter 8 *Interrupt Matrix (INTERRUPT)*.

- 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

### 16.3.4 Low-power Management Registers

The following registers are used for low-power management. For more information, please refer to Chapter 9 *Low-power Management*.

- SYSTEM_RTC_FASTMEM_CONFIG_REG: configures the RTC CRC check.
- SYSTEM_RTC_FASTMEM_CRC_REG: configures the CRC check value.

### 16.3.5 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 16.3-2.

- SYSTEM_CACHE_CONTROL_REG
- SYSTEM_PERIP_CLK_EN0_REG
- SYSTEM_PERIP_RST_EN0_REG

- SYSTEM_PERIP_CLK_EN1_REG
- SYSTEM_PERIP_RST_EN1_REG

ESP32-C3 features low power consumption. This is why some peripheral clocks are gated (disabled) by default. Before using any of these peripherals, it is mandatory to enable the clock for the given peripheral and release the peripheral from reset state. For details, see the table below:

Table 16.3-2. Clock Gating and Reset Bits

| Component | Clock Enabling Bit [1] | Reset Controlling Bit [2] [3] |
|---|---|---|
| **CACHE Control** | **SYSTEM_CACHE_CONTROL_REG** | |
| DCACHE | SYSTEM_DCACHE_CLK_ON | SYSTEM_DCACHE_RESET |
| ICACHE | SYSTEM_ICACHE_CLK_ON | SYSTEM_ICACHE_RESET |
| **CPU** | **SYSTEM_CPU_PERI_CLK_EN_REG** | **SYSTEM_CPU_PERI_RST_EN_REG** |
| DEBUG_ASSIST | SYSTEM_CLK_EN_ASSIST_DEBUG | SYSTEM_RST_EN_ASSIST_DEBUG |
| **Peripherals** | **SYSTEM_PERIP_CLK_EN0_REG** | **SYSTEM_PERIP_RST_EN0_REG** |
| TIMER | SYSTEM_TIMERS_CLK_EN | SYSTEM_TIMERS_RST |
| SPI0 / SPI1 | SYSTEM_SPI01_CLK_EN | SYSTEM_SPI01_RST |
| UART0 | SYSTEM_UART_CLK_EN | SYSTEM_UART_RST |
| UART1 | SYSTEM_UART1_CLK_EN | SYSTEM_UART1_RST |
| I2S | SYSTEM_I2S0_CLK_EN | SYSTEM_I2S0_RST |
| SPI2 | SYSTEM_SPI2_CLK_EN | SYSTEM_SPI2_RST |
| I2C0 | SYSTEM_EXT0_CLK_EN | SYSTEM_EXT0_RST |
| UHCI0 | SYSTEM_UHCI0_CLK_EN | SYSTEM_UHCI0_RST |
| RMT | SYSTEM_RMT_CLK_EN | SYSTEM_RMT_RST |
| LED PWM Controller | SYSTEM_LEDC_CLK_EN | SYSTEM_LEDC_RST |
| Timer Group0 | SYSTEM_TIMERGROUP_CLK_EN | SYSTEM_TIMERGROUP_RST |
| Timer Group1 | SYSTEM_TIMERGROUP1_CLK_EN | SYSTEM_TIMERGROUP1_RST |
| TWAI Controller | SYSTEM_CAN_CLK_EN | SYSTEM_CAN_RST |
| USB_DEVICE | SYSTEM_USB_DEVICE_CLK_EN | SYSTEM_USB_DEVICE_RST |
| UART MEM | SYSTEM_UART_MEM_CLK_EN [4] | SYSTEM_UART_MEM_RST |
| APB SARADC | SYSTEM_APB_SARADC_CLK_EN | SYSTEM_APB_SARADC_RST |
| ADC Controller | SYSTEM_ADC2_ARB_CLK_EN | SYSTEM_ADC2_ARB_RST |
| System Timer | SYSTEM_SYSTIMER_CLK_EN | SYSTEM_SYSTIMER_RST |
| **Accelerators** | **SYSTEM_PERIP_CLK_EN1_REG** | **SYSTEM_PERIP_RST_EN1_REG** |
| TSENS | SYSTEM_TSENS_CLK_EN | SYSTEM_TSENS_RST |
| DMA | SYSTEM_DMA_CLK_EN | SYSTEM_DMA_RST[5] |
| HMAC | SYSTEM_CRYPTO_HMAC_CLK_EN | SYSTEM_CRYPTO_HMAC_RST [6] |
| Digital Signature | SYSTEM_CRYPTO_DS_CLK_EN | SYSTEM_CRYPTO_DS_RST [7] |
| RSA Accelerator | SYSTEM_CRYPTO_RSA_CLK_EN | SYSTEM_CRYPTO_RSA_RST |
| SHA Accelerator | SYSTEM_CRYPTO_SHA_CLK_EN | SYSTEM_CRYPTO_SHA_RST |
| AES Accelerator | SYSTEM_CRYPTO_AES_CLK_EN | SYSTEM_CRYPTO_AES_RST |

Cont'd on next page

Table 16.3-2 – cont'd from previous page

| Component | Clock Enabling Bit [1] | Reset Controlling Bit [2] [3] |
|---|---|---|

[1] Set the clock enable bit to 1 to enable the clock, and to 0 to disable the clock;

[2] Set the reset enabling bit to 1 to reset a peripheral, and to 0 to disable the reset.

[3] Reset registers cannot be cleared by hardware. Therefore, SW reset clear is required after setting the reset registers.

[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] When DMA is required for periphral communications, for example, UCHI0, SPI2, I2S, AES, SHA, and ADC, DMA clock should also be enabled.

[6] Resetting this bit also resets the SHA accelerator.

[7] Resetting this bit also resets the AES, SHA, and RSA accelerators.

## 16.4 Register Summary

The addresses in this section are relative to the base address of system registers provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Peripheral Clock Control Registers** | | | |
| SYSTEM_CPU_PERI_CLK_EN_REG | CPU peripheral clock enable register | 0x0000 | R/W |
| SYSTEM_CPU_PERI_RST_EN_REG | CPU peripheral clock reset register | 0x0004 | R/W |
| SYSTEM_PERIP_CLK_EN0_REG | System peripheral clock enable register 0 | 0x0010 | R/W |
| SYSTEM_PERIP_CLK_EN1_REG | System peripheral clock enable register 1 | 0x0014 | R/W |
| SYSTEM_PERIP_RST_EN0_REG | System peripheral clock reset register 0 | 0x0018 | R/W |
| SYSTEM_PERIP_RST_EN1_REG | System peripheral clock reset register 1 | 0x001C | R/W |
| SYSTEM_CACHE_CONTROL_REG | Cache clock control register | 0x0040 | R/W |
| **Clock Configuration Registers** | | | |
| SYSTEM_CPU_PER_CONF_REG | CPU clock configuration register | 0x0008 | R/W |
| SYSTEM_SYSCLK_CONF_REG | System clock configuration register | 0x0058 | varies |
| **Low-power Management Registers** | | | |
| SYSTEM_BT_LPCK_DIV_FRAC_REG | Low-power clock configuration register 1 | 0x0024 | R/W |
| SYSTEM_RTC_FASTMEM_CONFIG_REG | Fast memory CRC configuration register | 0x0048 | varies |
| SYSTEM_RTC_FASTMEM_CRC_REG | Fast memory CRC result register | 0x004C | RO |
| **CPU Interrupt Control Registers** | | | |
| SYSTEM_CPU_INTR_FROM_CPU_0_REG | CPU interrupt control register 0 | 0x0028 | R/W |
| SYSTEM_CPU_INTR_FROM_CPU_1_REG | CPU interrupt control register 1 | 0x002C | R/W |
| SYSTEM_CPU_INTR_FROM_CPU_2_REG | CPU interrupt control register 2 | 0x0030 | R/W |
| SYSTEM_CPU_INTR_FROM_CPU_3_REG | CPU interrupt control register 3 | 0x0034 | R/W |
| **System and Memory Control Registers** | | | |
| SYSTEM_RSA_PD_CTRL_REG | RSA memory power control register | 0x0038 | R/W |
| SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG | External memory encryption and decryption control register | 0x0044 | R/W |
| **Clock Gate Control Register** | | | |
| SYSTEM_CLOCK_GATE_REG | Clock gate control register | 0x0054 | R/W |
| **Date Register** | | | |
| SYSTEM_DATE_REG | Version register | 0x0FFC | R/W |

The addresses below are relative to the base address of apb control provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Register** | | | |
| SYSCON_CLKGATE_FORCE_ON_REG | Internal memory clock gate enable register | 0x00A4 | R/W |
| SYSCON_MEM_POWER_DOWN_REG | Internal memory control register | 0x00A8 | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| SYSCON_MEM_POWER_UP_REG | Internal memory control register | 0x00AC | R/W |

## 16.5 Registers

The addresses below are relative to the base address of system register provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 16.1. SYSTEM_CPU_PERI_CLK_EN_REG (0x0000)**

| (reserved) | SYSTEM_CLK_EN_ASSIST_DEBUG | (reserved) | |
|---|---|---|---|
| 30 … 7 | 6 | 5 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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_CLK_EN_ASSIST_DEBUG** Set this bit to enable the ASSIST_DEBUG clock. Please see Chapter 17 *Debug Assistant (ASSIST_DEBUG)* for more information about ASSIST_DEBUG. (R/W)

**Register 16.2. SYSTEM_CPU_PERI_RST_EN_REG (0x0004)**

| (reserved) | SYSTEM_RST_EN_ASSIST_DEBUG | (reserved) | |
|---|---|---|---|
| 31 … 8 | 6 | 5 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 | 0 0 0 0 0 0 | Reset |

**SYSTEM_RST_EN_ASSIST_DEBUG** Set this bit to reset the ASSIST_DEBUG clock. Please see Chapter 17 *Debug Assistant (ASSIST_DEBUG)* for more information about ASSIST_DEBUG. (R/W)

**Register 16.3. SYSTEM_PERIP_CLK_EN0_REG (0x0010)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | SYSTEM_ADC2_ARB_CLK_EN | SYSTEM_SYSTIMER_CLK_EN | SYSTEM_APB_SARADC_CLK_EN | SYSTEM_SPI3_DMA_CLK_EN | (reserved) | | SYSTEM_UART_MEM_CLK_EN | SYSTEM_USB_DEVICE_CLK_EN | (reserved) | SYSTEM_I2S0_CLK_EN | (reserved) | SYSTEM_CAN_CLK_EN | (reserved) | | | SYSTEM_TIMERGROUP1_CLK_EN | (reserved) | SYSTEM_TIMERGROUP_CLK_EN | (reserved) | SYSTEM_LEDC_CLK_EN | (reserved) | SYSTEM_RMT_CLK_EN | SYSTEM_UHCI0_CLK_EN | SYSTEM_EXT0_CLK_EN | SYSTEM_SPI2_CLK_EN | SYSTEM_UART1_CLK_EN | (reserved) | (reserved) | SYSTEM_UART_CLK_EN | SYSTEM_SPI01_CLK_EN | SYSTEM_TIMERS_CLK_EN | |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | Reset |

**SYSTEM_TIMERS_CLK_EN** Set this bit to enable TIMERS clock. (R/W)

**SYSTEM_SPI01_CLK_EN** Set this bit to enable SPI0 / SPI1 clock. (R/W)

**SYSTEM_UART_CLK_EN** Set this bit to enable UART clock. (R/W)

**SYSTEM_UART1_CLK_EN** Set this bit to enable UART1 clock. (R/W)

**SYSTEM_SPI2_CLK_EN** Set this bit to enable SPI2 clock. (R/W)

**SYSTEM_EXT0_CLK_EN** Set this bit to enable I2C_EXT0 clock. (R/W)

**SYSTEM_UHCI0_CLK_EN** Set this bit to enable UHCI0 clock. (R/W)

**SYSTEM_RMT_CLK_EN** Set this bit to enable RMT clock. (R/W)

**SYSTEM_LEDC_CLK_EN** Set this bit to enable LEDC clock. (R/W)

**SYSTEM_TIMERGROUP_CLK_EN** Set this bit to enable TIMER GROUP clock. (R/W)

**SYSTEM_TIMERGROUP1_CLK_EN** Set this bit to enable TIMERGROUP1 clock. (R/W)

**SYSTEM_CAN_CLK_EN** Set this bit to enable TWAI clock. (R/W)

**SYSTEM_I2S0_CLK_EN** Set this bit to enable I2S clock. (R/W)

**SYSTEM_USB_DEVICE_CLK_EN** Set this bit to enable USB DEVICE clock. (R/W)

**SYSTEM_UART_MEM_CLK_EN** Set this bit to enable UART_MEM clock. (R/W)

**SYSTEM_SPI3_DMA_CLK_EN** Set this bit to enable SPI3 DMA clock. (R/W)

**SYSTEM_APB_SARADC_CLK_EN** Set this bit to enable APB_SARADC clock. (R/W)

**SYSTEM_SYSTIMER_CLK_EN** Set this bit to enable SYSTEMTIMER clock. (R/W)

**SYSTEM_ADC2_ARB_CLK_EN** Set this bit to enable ADC2_ARB clock. (R/W)

**Register 16.4. SYSTEM_PERIP_CLK_EN1_REG (0x0014)**

| 31 | 11 | 10 | 9 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | SYSTEM_TSENS_CLK_EN | (reserved) | | SYSTEM_DMA_CLK_EN | SYSTEM_CRYPTO_HMAC_CLK_EN | SYSTEM_CRYPTO_DS_CLK_EN | SYSTEM_CRYPTO_RSA_CLK_EN | SYSTEM_CRYPTO_SHA_CLK_EN | SYSTEM_CRYPTO_AES_CLK_EN | (reserved) | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | | 0 | 1 0 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**SYSTEM_CRYPTO_AES_CLK_EN** Set this bit to enable AES clock. (R/W)

**SYSTEM_CRYPTO_SHA_CLK_EN** Set this bit to enable SHA clock. (R/W)

**SYSTEM_CRYPTO_RSA_CLK_EN** Set this bit to enable RSA clock. (R/W)

**SYSTEM_CRYPTO_DS_CLK_EN** Set this bit to enable DS clock. (R/W)

**SYSTEM_CRYPTO_HMAC_CLK_EN** Set this bit to enable HMAC clock. (R/W)

**SYSTEM_DMA_CLK_EN** Set this bit to enable DMA clock. (R/W)

**SYSTEM_TSENS_CLK_EN** Set this bit to enable TSENS clock. (R/W)

### Register 16.5. SYSTEM_PERIP_RST_EN0_REG (0x0018)

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | SYSTEM_ADC2_ARB_RST | SYSTEM_SYSTIMER_RST | SYSTEM_APB_SARADC_RST | SYSTEM_SPI3_DMA_RST | (reserved) | | SYSTEM_UART_MEM_RST | SYSTEM_USB_DEVICE_RST | (reserved) | SYSTEM_I2S0_RST | (reserved) | SYSTEM_CAN_RST | (reserved) | | | SYSTEM_TIMERGROUP1_RST | (reserved) | SYSTEM_TIMERGROUP_RST | (reserved) | SYSTEM_LEDC_RST | (reserved) | SYSTEM_RMT_RST | SYSTEM_UHCI0_RST | SYSTEM_EXT0_RST | SYSTEM_SPI2_RST | SYSTEM_UART1_RST | (reserved) | (reserved) | SYSTEM_UART_RST | SYSTEM_SPI01_RST | SYSTEM_TIMERS_RST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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_TIMERS_RST** Set this bit to reset TIMERS. (R/W)

**SYSTEM_SPI01_RST** Set this bit to reset SPI0 / SPI1. (R/W)

**SYSTEM_UART_RST** Set this bit to reset UART. (R/W)

**SYSTEM_UART1_RST** Set this bit to reset UART1. (R/W)

**SYSTEM_SPI2_RST** Set this bit to reset SPI2. (R/W)

**SYSTEM_EXT0_RST** Set this bit to reset I2C_EXT0. (R/W)

**SYSTEM_UHCI0_RST** Set this bit to reset UHCI0. (R/W)

**SYSTEM_RMT_RST** Set this bit to reset RMT. (R/W)

**SYSTEM_LEDC_RST** Set this bit to reset LEDC. (R/W)

**SYSTEM_TIMERGROUP_RST** Set this bit to reset TIMERGROUP. (R/W)

**SYSTEM_TIMERGROUP1_RST** Set this bit to reset TIMERGROUP1. (R/W)

**SYSTEM_CAN_RST** Set this bit to reset CAN. (R/W)

**SYSTEM_I2S0_RST** Set this bit to reset I2S. (R/W)

**SYSTEM_USB_DEVICE_RST** Set this bit to reset USB DEVICE. (R/W)

**SYSTEM_UART_MEM_RST** Set this bit to reset UART_MEM. (R/W)

**SYSTEM_SPI3_DMA_RST** Set this bit to reset SPI3. (R/W)

**SYSTEM_APB_SARADC_RST** Set this bit to reset APB_SARADC. (R/W)

**SYSTEM_SYSTIMER_RST** Set this bit to reset SYSTIMER. (R/W)

**SYSTEM_ADC2_ARB_RST** Set this bit to reset ADC2_ARB. (R/W)

**Register 16.6. SYSTEM_PERIP_RST_EN1_REG (0x001C)**

| (reserved) | SYSTEM_TSENS_RST | (reserved) | SYSTEM_DMA_RST | SYSTEM_CRYPTO_HMAC_RST | SYSTEM_CRYPTO_DS_RST | SYSTEM_CRYPTO_RSA_RST | SYSTEM_CRYPTO_SHA_RST | SYSTEM_CRYPTO_AES_RST | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 11 | 10 | 9 … 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 1 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | Reset |

**SYSTEM_CRYPTO_AES_RST** Set this bit to reset CRYPTO_AES. (R/W)

**SYSTEM_CRYPTO_SHA_RST** Set this bit to reset CRYPTO_SHA. (R/W)

**SYSTEM_CRYPTO_RSA_RST** Set this bit to reset CRYPTO_RSA. (R/W)

**SYSTEM_CRYPTO_DS_RST** Set this bit to reset CRYPTO_DS. (R/W)

**SYSTEM_CRYPTO_HMAC_RST** Set this bit to reset CRYPTO_HMAC. (R/W)

**SYSTEM_DMA_RST** Set this bit to reset DMA. (R/W)

**SYSTEM_TSENS_RST** Set this bit to reset TSENS. (R/W)

**Register 16.7. SYSTEM_CACHE_CONTROL_REG (0x0040)**

| (reserved) | SYSTEM_DCACHE_RESET | SYSTEM_DCACHE_CLK_ON | SYSTEM_ICACHE_RESET | SYSTEM_ICACHE_CLK_ON | |
|---|---|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 1 | 0 | 1 | Reset |

**SYSTEM_ICACHE_CLK_ON** Set this bit to enable i-cache clock. (R/W)

**SYSTEM_ICACHE_RESET** Set this bit to reset i-cache. (R/W)

**SYSTEM_DCACHE_CLK_ON** Set this bit to enable d-cache clock. (R/W)

**SYSTEM_DCACHE_RESET** Set this bit to reset d-cache. (R/W)

**Register 16.8. SYSTEM_CPU_PER_CONF_REG (0x0008)**

| (reserved) | SYSTEM_CPU_WAITI_DELAY_NUM | SYSTEM_CPU_WAIT_MODE_FORCE_ON | SYSTEM_PLL_FREQ_SEL | SYSTEM_CPUPERIOD_SEL | |
|---|---|---|---|---|---|
| 31 … 8 | 7 … 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 | 0x0 | 1 | 1 | 0 | Reset |

**SYSTEM_CPUPERIOD_SEL** Set this field to select the CPU clock frequency. For details, please refer to Table 6.2-3 in Chapter 6 *Reset and Clock*.(R/W)

**SYSTEM_PLL_FREQ_SEL** Set this bit to select the PLL clock frequency. For details, please refer to Table 6.2-3 in Chapter 6 *Reset and Clock*. (R/W)

**SYSTEM_CPU_WAIT_MODE_FORCE_ON** Set this bit to force on the clock gate of CPU wait mode. Usually, after executing the WFI instruction, CPU enters the wait mode, during which the clock gate of CPU is turned off until any interrupts occur. In this way, power consumption is saved. However, if this bit is set, the clock gate of CPU is always on and will not be turned off by the WFI instruction. (R/W)

**SYSTEM_CPU_WAITI_DELAY_NUM** Sets the number of delay cycles to turn off the CPU clock gate after the CPU enters the wait mode because of a WFI instruction. (R/W)

**Register 16.9. SYSTEM_BT_LPCK_DIV_FRAC_REG (0x0024)**

| (reserved) | SYSTEM_LPCLK_RTC_EN | SYSTEM_LPCLK_SEL_XTAL32K | SYSTEM_LPCLK_SEL_XTAL | SYSTEM_LPCLK_SEL_8M | SYSTEM_LPCLK_SEL_RTC_SLOW | (reserved) | |
|---|---|---|---|---|---|---|---|
| 31 … 29 | 28 | 27 | 26 | 25 | 24 | 23 … 0 | |
| 0 0 0 | 0 | 0 | 0 | 1 | 0 | | Reset |

**SYSTEM_LPCLK_SEL_RTC_SLOW** Set this bit to select RTC_SLOW_CLK as the low-power clock. (R/W)

**SYSTEM_LPCLK_SEL_8M** Set this bit to select RC_FAST_CLK div n 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 LOW_POWER_CLK clock. (R/W)

**Register 16.10. SYSTEM_SYSCLK_CONF_REG (0x0058)**

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

**SYSTEM_PRE_DIV_CNT** This field is used to set the count of prescaler of XTAL_CLK. For details, please refer to Table 6.2-3 in Chapter 6 *Reset and Clock*. (R/W)

**SYSTEM_SOC_CLK_SEL** This field is used to select SOC clock. For details, please refer to Table 6.2-1 in Chapter 6 *Reset and Clock*. (R/W)

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

**Register 16.11. SYSTEM_RTC_FASTMEM_CONFIG_REG (0x0048)**

| SYSTEM_RTC_MEM_CRC_FINISH | SYSTEM_RTC_MEM_CRC_LEN | SYSTEM_RTC_MEM_CRC_ADDR | SYSTEM_RTC_MEM_CRC_START | (reserved) | |
|---|---|---|---|---|---|
| 31 | 30 – 20 | 19 – 9 | 8 | 7 – 0 | |
| 0 | 0x7ff | 0x0 | 0 | 0 0 0 0 0 0 0 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 16.12. SYSTEM_RTC_FASTMEM_CRC_REG (0x004C)

| 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 16.13. SYSTEM_CPU_INTR_FROM_CPU_0_REG (0x0028)

| (reserved) (31:1) | SYSTEM_CPU_INTR_FROM_CPU_0 (0) | |
|---|---|---|
| 0 0 0 0 0 0 0 0 0 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_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 16.14. SYSTEM_CPU_INTR_FROM_CPU_1_REG (0x002C)

| (reserved) (31:1) | SYSTEM_CPU_INTR_FROM_CPU_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 | Reset |

**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 16.15. SYSTEM_CPU_INTR_FROM_CPU_2_REG (0x0030)**

| (reserved) | SYSTEM_CPU_INTR_FROM_CPU_2 | |
|---|---|---|
| 31 … 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 | Reset |

**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 16.16. SYSTEM_CPU_INTR_FROM_CPU_3_REG (0x0034)**

| (reserved) | SYSTEM_CPU_INTR_FROM_CPU_3 | |
|---|---|---|
| 31 … 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 | Reset |

**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 16.17. SYSTEM_RSA_PD_CTRL_REG (0x0038)**

| (reserved) | SYSTEM_RSA_MEM_FORCE_PD | SYSTEM_RSA_MEM_FORCE_PU | SYSTEM_RSA_MEM_PD | |
|---|---|---|---|---|
| 31 … 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 | 1 | Reset |

**SYSTEM_RSA_MEM_PD** Set this bit to send the RSA memory into retention state. This bit has the lowest priority, meaning it can be masked by the SYSTEM_RSA_MEM_FORCE_PU field. When Digital Signature occupies the RSA, this bit is invalid. (R/W)

**SYSTEM_RSA_MEM_FORCE_PU** Set this bit to force the RSA memory to work as normal when the chip enters light sleep. This bit has the second highest priority, meaning it overrides the SYSTEM_RSA_MEM_PD field. (R/W)

**SYSTEM_RSA_MEM_FORCE_PD** Set this bit to send the RSA memory into retention state. This bit has the highest priority, meaning it sends the RSA memory into retention state regardless of the SYSTEM_RSA_MEM_FORCE_PU field. (R/W)

**Register 16.18. SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG (0x0044)**

| (reserved) | SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT | SYSTEM_ENABLE_DOWNLOAD_GOCB_DECRYPT | SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT | SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT | |
|---|---|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 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_GOCB_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 16.19. SYSTEM_CLOCK_GATE_REG (0x0054)**

| (reserved) | SYSTEM_CLK_EN | |
|---|---|---|
| 31 … 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 | 1 | Reset |

**SYSTEM_CLK_EN** Set this bit to enable the system clock. (R/W)

**Register 16.20. SYSTEM_DATE_REG (0x0FFC)**

| (reserved) | SYSTEM_DATE | |
|---|---|---|
| 31 … 28 | 27 … 0 | |
| 0 0 0 0 | 0x2007150 | Reset |

**SYSTEM_DATE** Version control register. (R/W)

The addresses below are relative to the base address of apb control provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 16.21. SYSCON_CLKGATE_FORCE_ON_REG (0x00A4)**

| (reserved) | SYSCON_SRAM_CLKGATE_FORCE_ON | SYSCON_ROM_CLKGATE_FORCE_ON | |
|---|---|---|---|
| 31 … 6 | 5 … 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 | 0xf | 3 | Reset |

**SYSCON_ROM_CLKGATE_FORCE_ON** Set 1 to configure the ROM clock gate to be always on; Set 0 to configure the clock gate to turn on automatically when ROM is accessed and turn off automatically when ROM is not accessed. (R/W)

**SYSCON_SRAM_CLKGATE_FORCE_ON** Set 1 to configure the SRAM clock gate to be always on; Set 0 to configure the clock gate to turn on automatically when SRAM is accessed and turn off automatically when SRAM is not accessed. (R/W)

**Register 16.22. SYSCON_MEM_POWER_DOWN_REG (0x00A8)**

| (reserved) | SYSCON_SRAM_POWER_DOWN | SYSCON_ROM_POWER_DOWN | |
|---|---|---|---|
| 31 … 6 | 5 … 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 | Reset |

**SYSCON_ROM_POWER_DOWN** Set this field to send the internal ROM into retention state. (R/W)

**SYSCON_SRAM_POWER_DOWN** Set this field to send the internal SRAM into retention state. (R/W)

**Register 16.23. SYSCON_MEM_POWER_UP_REG (0x00AC)**

| (reserved) | SYSCON_SRAM_POWER_UP | SYSCON_ROM_POWER_UP | |
|---|---|---|---|
| 31 … 6 | 5 … 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 | 0xf | 3 | Reset |

**SYSCON_ROM_POWER_UP** Set this field to force the internal ROM to work as normal (do not enter the retention state) when the chip enters light sleep. (R/W)

**SYSCON_SRAM_POWER_UP** Set this field to force the internal SRAM to work as normal (do not enter the retention state) when the chip enters light sleep. (R/W)

# Chapter 17

# Debug Assistant (ASSIST_DEBUG)

## 17.1 Overview

Debug Assistant is an auxiliary module that features a set of functions to help locate bugs and issues during software debugging.

## 17.2 Features

- **Read/write monitoring**: Monitors whether the CPU bus has read from or written to a specified address space. A detected read or write will trigger an interrupt.
- **Stack pointer (SP) monitoring**: Monitors whether the SP exceeds the specified address space. A bounds violation will trigger an interrupt.
- **Program counter (PC) logging**: Records PC value. The developer can get the last PC value at the most recent CPU reset.
- **Bus access logging**: Records the information about bus access. When the CPU or DMA writes a specified value, the Debug Assistant module will record the address and PC value of this write operation, and push the data to the SRAM.

## 17.3 Functional Description

### 17.3.1 Region Read/Write Monitoring

The Debug Assistant module can monitor reads/writes performed by the CPU's Data bus and Peripheral bus in a certain address space, i.e., memory region. Whenever the Data bus reads or writes in the specified address space, an interrupt will be triggered. The Data bus can monitor two memory regions (assuming they are region 0 and region 1, defined by developer's needs) at the same time, so can the Peripheral bus.

### 17.3.2 SP Monitoring

The Debug Assistant module can monitor the SP so as to prevent stack overflow or erroneous push/pop. When the stack pointer exceeds the minimum or maximum threshold, Debug Assistant will record the PC pointer and generate an interrupt. The threshold is configured by software.

### 17.3.3 PC Logging

In some cases, software developers want to know the PC at the last CPU reset. For instance, when the program is stuck and can only be reset, the developer may want to know where the program got stuck in order to debug. The Debug Assistant module can record the PC at the last CPU reset, which can be then read for software debugging.

### 17.3.4 CPU/DMA Bus Access Logging

The Debug Assistant module can record the information about the CPU Data bus's and DMA bus's write behaviors in real time. When a write operation occurs in or a specific value is written to a specified address space, the Debug Assistant will record the bus type, PC, and the address, and then store the data in the SRAM in a certain format.

## 17.4 Recommended Operation

### 17.4.1 Region Monitoring and SP Monitoring Configuration Process

The Debug Assistant module can monitor reads and writes performed by the CPU's Data bus and Peripheral bus. Two memory regions on each bus can be monitored at the same time. All the monitoring modes supported by the Debug Assistant module are listed below:

- Monitoring of the read/write operations on Data bus
  - Data bus reads in region 0
  - Data bus writes in region 0
  - Data bus reads in region 1
  - Data bus writes in region 1
- Monitoring of the read/write operations on Peripheral bus
  - Peripheral bus reads in region 0
  - Peripheral bus writes in region 0
  - Peripheral bus reads in region 1
  - Peripheral bus writes in region 1
- Monitoring of exceeding the SP bounds
  - SP exceeds the upper bound address
  - SP exceeds the lower bound address

The configuration process for region monitoring and SP monitoring is as follows:

1. Configure monitored region and SP threshold.
   - Configure Data bus region 0 with ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MIN_REG and ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MAX_REG.

- Configure Data bus region 1 with ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_MIN_REG and ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_MAX_REG.
- Configure Peripheral bus region 0 with ASSIST_DEBUG_CORE_0_AREA_PIF_0_MIN_REG and ASSIST_DEBUG_CORE_0_AREA_PIF_0_MAX_REG.
- Configure Peripheral bus region 1 with ASSIST_DEBUG_CORE_0_AREA_PIF_1_MIN_REG and ASSIST_DEBUG_CORE_0_AREA_PIF_1_MAX_REG.
- Configure SP threshold with ASSIST_DEBUG_CORE_0_SP_MIN_REG and ASSIST_DEBUG_CORE_0_SP_MAX_REG.

2. Configure interrupts.
   - Configure ASSIST_DEBUG_CORE_0_INTR_ENA_REG to enable the interrupt of a monitoring mode.
   - Configure ASSIST_DEBUG_CORE_0_INTR_RAW_REG to get the interrupt status of a monitoring mode.
   - Configure ASSIST_DEBUG_CORE_0_INTR_CLR_REG to clear the interrupt of a monitoring mode.
3. Configure ASSIST_DEBUG_CORE_0_MONTR_ENA_REG to enable the monitoring mode(s). Various monitoring modes can be enabled at the same time.

Assuming that Debug Assistant needs to monitor whether Data bus has written to [A ~ B] address space, the user can enable monitoring in either Data bus region 0 or region 1. The following configuration process is based on region 0:

1. Configure ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MIN_REG to A.
2. Configure ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MAX_REG to B.
3. Configure ASSIST_DEBUG_CORE_0_INTR_ENA_REG bit[1] to enable the interrupt for write operations by Data bus in region 0.
4. Configure ASSIST_DEBUG_CORE_0_MONTR_ENA_REG bit[1] to enable monitoring write operations by Data bus in region 0.
5. Configure interrupt matrix to map ASSIST_DEBUG_INT into CPU interrupt (please refer to Chapter 8 *Interrupt Matrix (INTERRUPT)*).
6. After the interrupt is triggered:
   - Read ASSIST_DEBUG_CORE_0_INTR_RAW_REG to learn which operation triggered interrupt.
   - If the interrupt is triggered by region monitoring, read ASSIST_DEBUG_CORE_0_AREA_PC_REG for the PC value, and ASSIST_DEBUG_CORE_0_AREA_SP_REG for the SP.
   - If the interrupt is triggered by stack monitoring, read ASSIST_DEBUG_CORE_0_SP_PC_REG for the PC value.
   - Write '1' to the corresponding bits of ASSIST_DEBUG_CORE_0_INTR_RAW_REG to clear the interrupts.

### 17.4.2 PC Logging Configuration Process

The CPU sends PC signals to Debug Assistant. Only when ASSIST_DEBUG_CORE_0_RCD_PDEBUGEN is 1, the PC signal is valid, otherwise, it is always 0.

Only when ASSIST_DEBUG_CORE_0_RCD_RECORDEN is 1, ASSIST_DEBUG_CORE_0_RCD_PDEBUGPC_REG samples the CPU's PC signals, otherwise, it keeps the original value.

The description of ASSIST_DEBUG_CORE_0_RCD_EN_REG and ASSIST_DEBUG_CORE_0_RCD_PDEBUGPC_REG can be found in section 17.18 and 17.19.

When the CPU resets, ASSIST_DEBUG_CORE_0_RCD_EN_REG will reset, while ASSIST_DEBUG_CORE_0_RCD_PDEBUGPC_REG will not. Therefore, the latter will keep the PC value at the CPU reset.

### 17.4.3 CPU/DMA Bus Access Logging Configuration Process

The configuration process for CPU/DMA bus access logging is described below.

1. Configure monitored address space.
   - Configure ASSIST_DEBUG_LOG_MIN_REG and ASSIST_DEBUG_LOG_MAX_REG to specify monitored address space.
2. Configure monitoring mode with ASSIST_DEBUG_LOG_MODE:
   - write monitoring (whether the bus has write operations)
   - word monitoring (whether the bus writes a specific word)
   - halfword monitoring (whether the bus writes a specific halfword)
   - byte monitoring (whether the bus writes a specific byte)
3. Configure the specific values to be monitored.
   - In word monitoring mode, ASSIST_DEBUG_LOG_DATA_0_REG specifies the monitored word.
   - In halfword monitoring mode, ASSIST_DEBUG_LOG_DATA_0_REG[15:0] specifies the monitored halfword.
   - In byte monitoring mode, ASSIST_DEBUG_LOG_DATA_0_REG[7:0] specifies the monitored byte.
   - ASSIST_DEBUG_LOG_DATA_MASK_REG is used to mask the byte specified in ASSIST_DEBUG_LOG_DATA_0_REG. A masked byte can be any value. For example, in word monitoring, ASSIST_DEBUG_LOG_DATA_0_REG is configured to 0x01020304, and ASSIST_DEBUG_LOG_DATA_MASK_REG is configured to 0x1, then bus writes with data matching to 0x010203XX pattern will be recorded.
4. Configure the storage space for recorded data.
   - ASSIST_DEBUG_LOG_MEM_START_REG and ASSIST_DEBUG_LOG_MEM_END_REG specify the storage space for recorded data. The storage space must be in the range of 0x3FCC_0000 ~ 0x3FCD_FFFF.
   - Configure the permission for the Debug Assistant module to access the internal SRAM. Only if the access permission is enabled, the Debug Assistant module is able to access the internal SRAM. For more information please refer to Chapter 14 *Permission Control (PMS)*).
5. Configure the writing mode for recorded data: loop mode and non-loop mode.

- In loop mode, writing to specified address space is performed in loops. When writing reaches the end address, it will return to the starting address and continue, overwriting the previously recorded data.
  For example, 10 writes (1 ~ 10) write to address space 0 ~ 4. After the 5th write writes to address 4, the 6th write will start writing from address 0. The 6th to 10th writes will overwrite the previous data written by 0 ~ 4 writes.
- In non-loop mode, when writing reaches the end address, it will stop at the end address, not overwriting the previously recorded data.
  For example, 10 writes (1 ~ 10) write to address space 0 ~ 4. After the 5th write writes to address 4, the 6th to 10th writes will write at address 4. Only the data written by the last (10th) write will be retained at address 4.

6. Configure bus enable registers.
   - Enable CPU or DMA bus access logging with ASSIST_DEBUG_LOG_ENA. CPU and DMA bus access logging can be enabled at the same time.

When bus access logging is finished, the recorded data can be read from memory for decoding. The recorded data is in two packet formats, namely CPU packet (corresponding to CPU bus) and DMA packet (corresponding to DMA bus). The packet formats are shown in Table 17.4-1 and 17.4-2:

**Table 17.4-1. CPU Packet Format**

| Bit[49:29] | Bit[28:2] | Bit[1:0] |
|---|---|---|
| addr_offset | pc_offset | format |

**Table 17.4-2. DMA Packet Format**

| Bit[24:6] | Bit[5:2] | Bit[1:0] |
|---|---|---|
| addr_offset | dma_source | format |

It can be seen from the data packet formats that the CPU packet size is 50 bits and DMA packet size 25 bits. The packet formats contain the following fields:

- **format** – the packet type. 1: CPU packet; 3: DMA packet; other values: reserved.
- **pc_offset** – the offset of the PC register at time of access. Actual PC = pc_offset + 0x4000_0000.
- **addr_offset** – the address offset of a write operation. Actual adddress = addr_offset + ASSIST_DEBUG_LOG_MIN_REG.
- **dma_source** – the source of DMA access. Refer to Table 17.4-3.

**Table 17.4-3. DMA Source**

| Value | Source |
|---|---|
| 1 | SPI2 |
| 2 | reserved |
| 3 | reserved |
| 4 | AES |

| Value | Source |
|---|---|
| 5 | SHA |
| 6 | ADC |
| 7 | I2S |
| 8 | reserved |
| 9 | reserved |
| 10 | reserved |
| 11 | UHCI0 |
| 12 | reserved |
| 13 | reserved |
| 14 | reserved |
| 15 | reserved |

The packets are stored in the internal buffer first. When the buffered data reaches 125 bits, it will be expanded to 128 bits and written to the internal SRAM. The written data format is shown in Table 17.4-4.

**Table 17.4-4. Written Data Format**

| Bit[127:3] | Bit[2:0] |
|---|---|
| Valid packets | START_FLAG |

Since the CPU packet size is 50 bits and the DMA packet size 25 bits, the recorded data in each record is at least 25 bits and at most 75 bits. When the data stored in the internal buffer reaches 125 bits, it will be popped into memory. There are cases where a packet is divided into two portions: the first portion is written to memory, and the second portion is left in the buffer and will be popped into memory in the next write. The data left in the buffer is called residual data. The value of START_FLAG records the number of residual bits left from the last write to memory. The number of residual bits is START_FLAG * 25. START_FLAG also indicates the starting bit of the first valid packet in the current write. As an example: Assume that four DMA writes have generated four DMA packets to be stored in the buffer with a total of 100-bit data. Then, one CPU write occurs and generates one 50-bit CPU packet. The buffer will pop the previously-recorded 100-bit data plus the first 25 bits in the CPU packet into SRAM. The remaining 25 bits in the CPU packet is left in the buffer, waiting for the next write. START_FLAG in the next write will indicate that 25 bits in this write is from the last write.

In loop writing mode, if data is looped several times in the storage memory, the residual data will interfere with packet parsing. Therefore, users need to filter out the residual data in order to determine the starting position of the first valid packet with START_FLAG and ASSIST_DEBUG_LOG_MEM_CURRENT_ADDR_REG. Once the starting position of the packet is identified, the subsequent data is continuous and users do not need to care about the value of START_FLAG.

Note that if data in the buffer does not reach 125 bits, it will not be written to memory. All data should be written to memory for packet parsing. This can be done by disabling bus access logging. When ASSIST_DEBUG_LOG_ENA is set to 0, if there is data in the buffer, it will be padded with zeros from the left until it becomes 128 bits long and written to the memory.

The process of packet parsing is described below:

- Determine whether there is a data overflow with ASSIST_DEBUG_LOG_MEM_FULL_FLAG. If there is no overflow, ASSIST_DEBUG_LOG_MEM_START_REG is the starting address of the first packet. If there is an

overflow and loop mode is enabled, ASSIST_DEBUG_LOG_MEM_CURRENT_ADDR_REG is the starting address of the first packet.

- Read and parse data from the starting address. Read 128 bits each time.
- Use START_FLAG to determine the starting bit of the first packet. Starting bit = START_FLAG * 25 + 3.

Note that START_FLAG is only used to locate the starting bit of the first packet. Once the starting bit is located, START_FLAG should be filtered out in the subsequent data.

After packet parsing is completed, clear the ASSIST_DEBUG_LOG_MEM_FULL_FLAG flag bit by setting ASSIST_DEBUG_CLR_LOG_MEM_FULL_FLAG.

## 17.5 Register Summary

The addresses in this section are relative to Debug Assistant base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Monitor configuration registers** | | | |
| ASSIST_DEBUG_CORE_0_MONTR_ENA_REG | Monitoring enable register | 0x0000 | R/W |
| ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MIN_REG | Configures boundary address of region 0 monitored on Data bus | 0x0010 | R/W |
| ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MAX_REG | Configures boundary address of region 0 monitored on Data bus | 0x0014 | R/W |
| ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_MIN_REG | Configures boundary address of region 1 monitored on Data bus | 0x0018 | R/W |
| ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_MAX_REG | Configures boundary address of region 1 monitored on Data bus | 0x001C | R/W |
| ASSIST_DEBUG_CORE_0_AREA_PIF_0_MIN_REG | Configures boundary address of region 0 monitored on Peripheral bus | 0x0020 | R/W |
| ASSIST_DEBUG_CORE_0_AREA_PIF_0_MAX_REG | Configures boundary address of region 0 monitored on Peripheral bus | 0x0024 | R/W |
| ASSIST_DEBUG_CORE_0_AREA_PIF_1_MIN_REG | Configures boundary address of region 1 monitored on Peripheral bus | 0x0028 | R/W |
| ASSIST_DEBUG_CORE_0_AREA_PIF_1_MAX_REG | Configures boundary address of region 1 monitored on Peripheral bus | 0x002C | R/W |
| ASSIST_DEBUG_CORE_0_AREA_PC_REG | Region monitoring PC status register | 0x0030 | RO |
| ASSIST_DEBUG_CORE_0_AREA_SP_REG | Region monitoring SP status register | 0x0034 | RO |
| ASSIST_DEBUG_CORE_0_SP_MIN_REG | Configures stack monitoring boundary address | 0x0038 | R/W |
| ASSIST_DEBUG_CORE_0_SP_MAX_REG | Configures stack monitoring boundary address | 0x003C | R/W |
| ASSIST_DEBUG_CORE_0_SP_PC_REG | Stack monitoring PC status register | 0x0040 | RO |
| **Interrupt configuration registers** | | | |
| ASSIST_DEBUG_CORE_0_INTR_RAW_REG | Interrupt status register | 0x0004 | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| ASSIST_DEBUG_CORE_0_INTR_ENA_REG | Interrupt enable register | 0x0008 | R/W |
| ASSIST_DEBUG_CORE_0_INTR_CLR_REG | Interrupt clear register | 0x000C | R/W |
| **PC logging configuration register** | | | |
| ASSIST_DEBUG_CORE_0_RCD_EN_REG | PC logging enable register | 0x0044 | R/W |
| **PC logging status registers** | | | |
| ASSIST_DEBUG_CORE_0_RCD_PDEBUGPC_REG | PC logging register | 0x0048 | RO |
| ASSIST_DEBUG_CORE_0_RCD_PDEBUGSP_REG | PC logging register | 0x004C | RO |
| **Bus access logging configuration registers** | | | |
| ASSIST_DEBUG_LOG_SETTING_REG | Bus access logging configuration register | 0x0070 | R/W |
| ASSIST_DEBUG_LOG_DATA_0_REG | Configures monitored data in Bus access logging | 0x0074 | R/W |
| ASSIST_DEBUG_LOG_DATA_MASK_REG | Configures masked data in Bus access logging | 0x0078 | R/W |
| ASSIST_DEBUG_LOG_MIN_REG | Configures monitored address space in Bus access logging | 0x007C | R/W |
| ASSIST_DEBUG_LOG_MAX_REG | Configures monitored address space in Bus access logging | 0x0080 | R/W |
| ASSIST_DEBUG_LOG_MEM_START_REG | Configures the starting address of the storage memory for recorded data | 0x0084 | R/W |
| ASSIST_DEBUG_LOG_MEM_END_REG | Configures the end address of the storage memory for recorded data | 0x0088 | R/W |
| ASSIST_DEBUG_LOG_MEM_CURRENT_ADDR_REG | The current address of the storage memory for recorded data | 0x008C | RO |
| ASSIST_DEBUG_LOG_MEM_FULL_FLAG_REG | Logging overflow status register | 0x0090 | varies |
| **CPU status registers** | | | |
| ASSIST_DEBUG_CORE_0_LASTPC_BEFORE_EXCEPTION_REG | PC of the last command before CPU enters exception | 0x0094 | RO |
| ASSIST_DEBUG_CORE_0_DEBUG_MODE_REG | CPU debug mode status register | 0x0098 | RO |
| **Version register** | | | |
| ASSIST_DEBUG_DATE_REG | Version control register | 0x01FC | R/W |

## 17.6 Registers

The addresses in this section are relative to Debug Assistant base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 17.1. ASSIST_DEBUG_CORE_0_MONTR_ENA_REG (0x0000)**

| 31 | | | | | | | | | | | | | | | | | | | | | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | ASSIST_DEBUG_CORE_0_SP_SPILL_MAX_ENA | ASSIST_DEBUG_CORE_0_SP_SPILL_MIN_ENA | ASSIST_DEBUG_CORE_0_AREA_PIF_1_WR_ENA | ASSIST_DEBUG_CORE_0_AREA_PIF_1_RD_ENA | ASSIST_DEBUG_CORE_0_AREA_PIF_0_WR_ENA | ASSIST_DEBUG_CORE_0_AREA_PIF_0_RD_ENA | ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_WR_ENA | ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_RD_ENA | ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_WR_ENA | ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_RD_ENA | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_RD_ENA** Monitoring enable bit for read operations in region 0 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_WR_ENA** Monitoring enable bit for write operations in region 0 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_RD_ENA** Monitoring enable bit for read operations in region 1 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_WR_ENA** Monitoring enable bit for write operations in region 1 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_RD_ENA** Monitoring enable bit for read operations in region 0 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_WR_ENA** Monitoring enable bit for write operations in region 0 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_RD_ENA** Monitoring enable bit for read operations in region 1 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_WR_ENA** Monitoring enable bit for write operations in region 1 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_SP_SPILL_MIN_ENA** Monitoring enable bit for SP exceeding the lower bound address of SP monitored region. (R/W)

**ASSIST_DEBUG_CORE_0_SP_SPILL_MAX_ENA** Monitoring enable bit for SP exceeding the upper bound address of SP monitored region. (R/W)

**Register 17.2. ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MIN_REG (0x0010)**

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MIN** The lower bound address of Data bus region 0. (R/W)

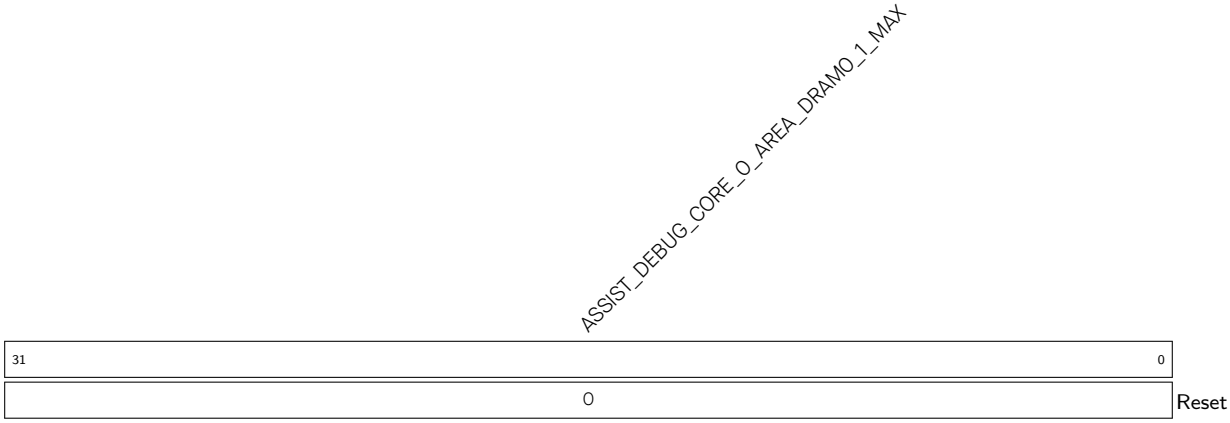
**Register 17.3. ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MAX_REG (0x0014)**

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_MAX** The upper bound address of Data bus region 0. (R/W)

**Register 17.4. ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_MIN_REG (0x0018)**

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_MIN** The lower bound address of Data bus region 1. (R/W)

**Register 17.5. ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_MAX_REG (0x001C)**




**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_MAX** The upper bound address of Data bus region 1. (R/W)

**Register 17.6. ASSIST_DEBUG_CORE_0_AREA_PIF_0_MIN_REG (0x0020)**

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_MIN** The lower bound address of Peripheral bus region 0. (R/W)

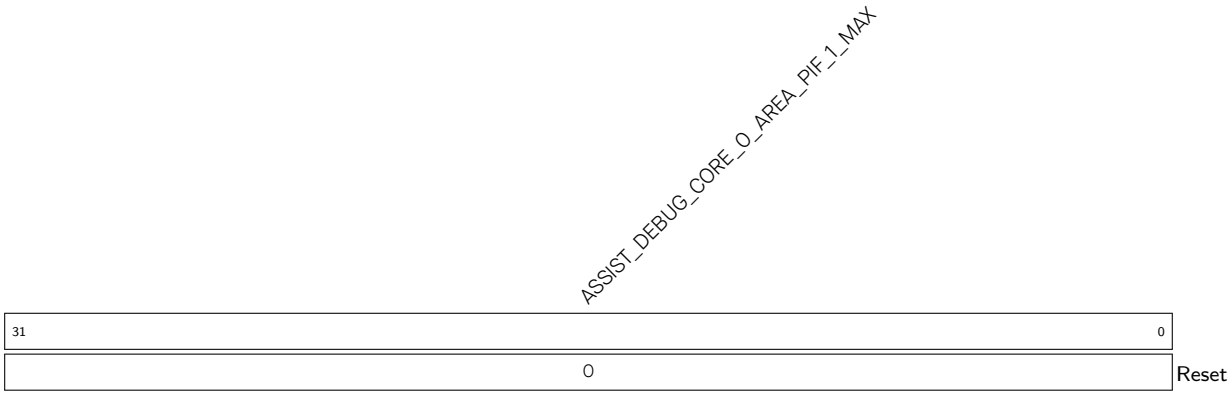
**Register 17.7. ASSIST_DEBUG_CORE_0_AREA_PIF_0_MAX_REG (0x0024)**

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_MAX** The upper bound address of Peripheral bus region 0. (R/W)

Register 17.8. ASSIST_DEBUG_CORE_0_AREA_PIF_1_MIN_REG (0x0028)

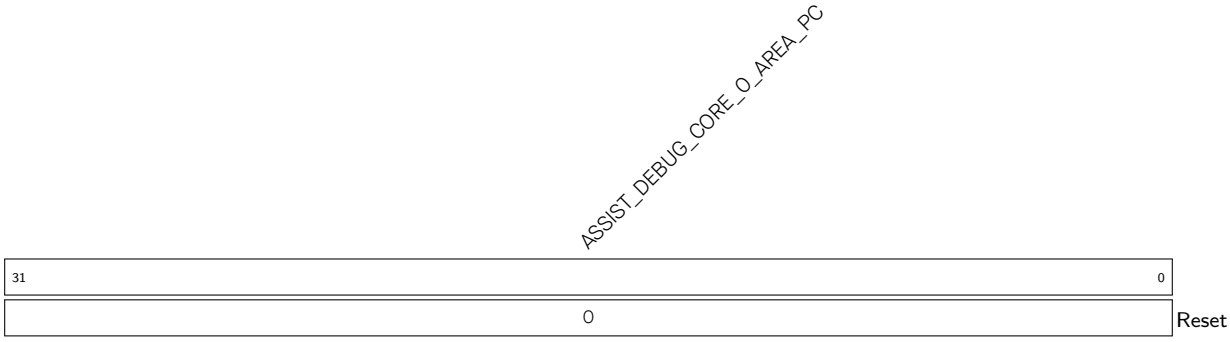
**ASSIST_DEBUG_CORE_0_AREA_PIF_1_MIN** The lower bound address of Peripheral bus region 1. (R/W)

Register 17.9. ASSIST_DEBUG_CORE_0_AREA_PIF_1_MAX_REG (0x002C)



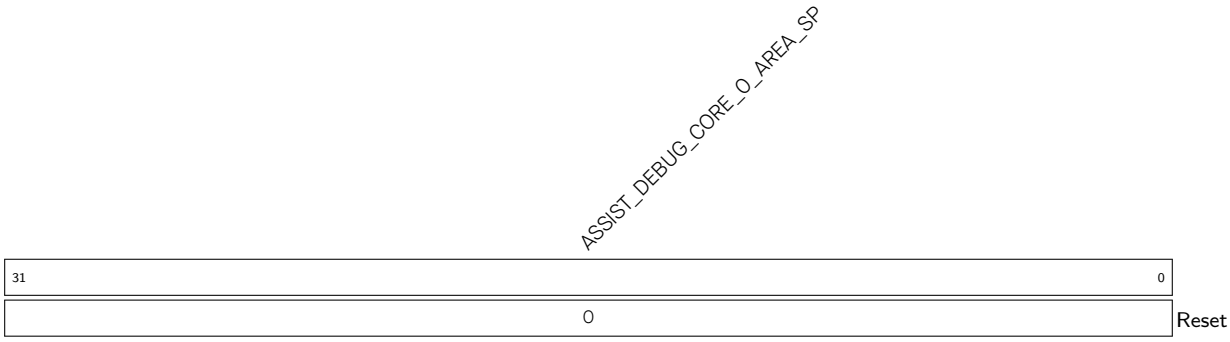

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_MAX** The upper bound address of Peripheral bus region 1. (R/W)

Register 17.10. ASSIST_DEBUG_CORE_0_AREA_PC_REG (0x0030)



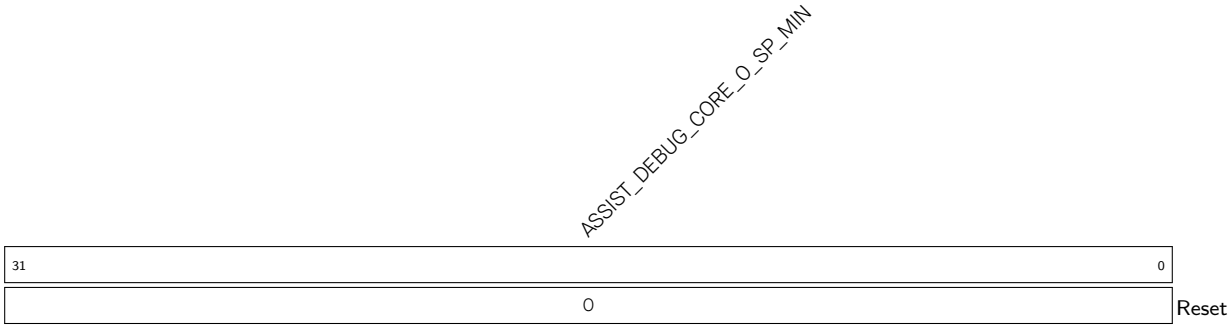

**ASSIST_DEBUG_CORE_0_AREA_PC** Records the PC value when interrupt triggers during region monitoring. (RO)

**Register 17.11. ASSIST_DEBUG_CORE_0_AREA_SP_REG (0x0034)**



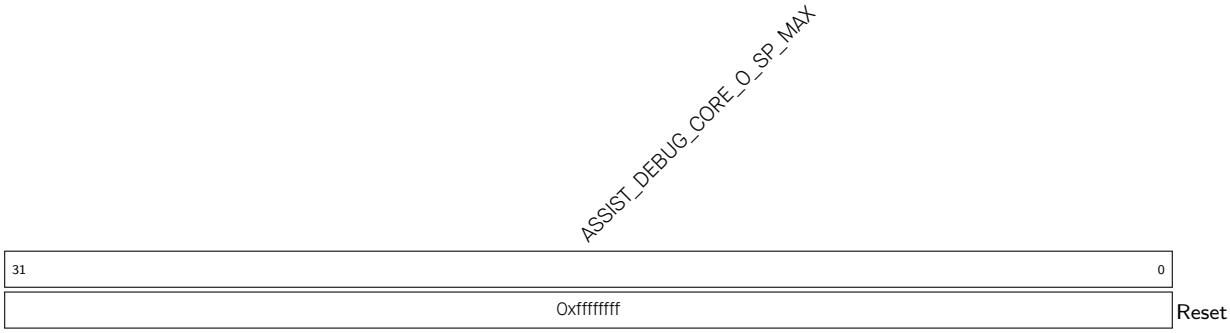

**ASSIST_DEBUG_CORE_0_AREA_SP** Records SP when interrupt triggers during region monitoring. (RO)

**Register 17.12. ASSIST_DEBUG_CORE_0_SP_MIN_REG (0x0038)**




**ASSIST_DEBUG_CORE_0_SP_MIN** The lower bound address of SP. (R/W)

**Register 17.13. ASSIST_DEBUG_CORE_0_SP_MAX_REG (0x003C)**




**ASSIST_DEBUG_CORE_0_SP_MAX** The upper bound address of SP. (R/W)

**Register 17.14. ASSIST_DEBUG_CORE_0_SP_PC_REG (0x0040)**

| ASSIST_DEBUG_CORE_0_SP_PC | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**ASSIST_DEBUG_CORE_0_SP_PC** Records the PC value during stack monitoring. (RO)

**Register 17.15. ASSIST_DEBUG_CORE_0_INTR_RAW_REG (0x0004)**

| 31 | | | | | | | | | | | | | | | | | | | | | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | ASSIST_DEBUG_CORE_0_SP_SPILL_MAX_RAW | ASSIST_DEBUG_CORE_0_SP_SPILL_MIN_RAW | ASSIST_DEBUG_CORE_0_AREA_PIF_1_WR_RAW | ASSIST_DEBUG_CORE_0_AREA_PIF_1_RD_RAW | ASSIST_DEBUG_CORE_0_AREA_PIF_0_WR_RAW | ASSIST_DEBUG_CORE_0_AREA_PIF_0_RD_RAW | ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_WR_RAW | ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_RD_RAW | ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_WR_RAW | ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_RD_RAW | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_RD_RAW** Interrupt status bit for read operations in region 0 by the Data bus. (RO)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_WR_RAW** Interrupt status bit for write operations in region 0 by the Data bus. (RO)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_RD_RAW** Interrupt status bit for read operations in region 1 by the Data bus. (RO)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_WR_RAW** Interrupt status bit for write operations in region 1 by the Data bus. (RO)

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_RD_RAW** Interrupt status bit for read operations in region 0 by the Peripheral bus. (RO)

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_WR_RAW** Interrupt status bit for write operations in region 0 by the Peripheral bus. (RO)

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_RD_RAW** Interrupt status bit for read operations in region 1 by the Peripheral bus. (RO)

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_WR_RAW** Interrupt status bit for write operations in region 1 by the Peripheral bus. (RO)

**ASSIST_DEBUG_CORE_0_SP_SPILL_MIN_RAW** Interrupt status bit for SP exceeding the lower bound address of SP monitored region. (RO)

**ASSIST_DEBUG_CORE_0_SP_SPILL_MAX_RAW** Interrupt status bit for SP exceeding the upper bound address of SP monitored region. (RO)

**Register 17.16. ASSIST_DEBUG_CORE_0_INTR_ENA_REG (0x0008)**

| 31 | | | | | | | | | | | | | | | | | | | | | | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | ASSIST_DEBUG_CORE_0_SP_SPILL_MAX_INTR_ENA | ASSIST_DEBUG_CORE_0_SP_SPILL_MIN_INTR_ENA | ASSIST_DEBUG_CORE_0_AREA_PIF_1_WR_INTR_ENA | ASSIST_DEBUG_CORE_0_AREA_PIF_1_RD_INTR_ENA | ASSIST_DEBUG_CORE_0_AREA_PIF_0_WR_INTR_ENA | ASSIST_DEBUG_CORE_0_AREA_PIF_0_RD_INTR_ENA | ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_WR_INTR_ENA | ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_RD_INTR_ENA | ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_WR_INTR_ENA | ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_RD_INTR_ENA | 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 | 0 | 0 | 0 | |

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_RD_INTR_ENA** Interrupt enable bit for read operations in region 0 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_WR_INTR_ENA** Interrupt enable bit for write operations in region 0 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_RD_INTR_ENA** Interrupt enable bit for read operations in region 1 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_WR_INTR_ENA** Interrupt enable bit for write operations in region 1 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_RD_INTR_ENA** Interrupt enable bit for read operations in region 0 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_WR_INTR_ENA** Interrupt enable bit for write operations in region 0 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_RD_INTR_ENA** Interrupt enable bit for read operations in region 1 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_WR_INTR_ENA** Interrupt enable bit for write operations in region 1 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_SP_SPILL_MIN_INTR_ENA** Interrupt enable bit for SP exceeding the lower bound address of SP monitored region. (R/W)

**ASSIST_DEBUG_CORE_0_SP_SPILL_MAX_INTR_ENA** Interrupt enable bit for SP exceeding the upper bound address of SP monitored region. (R/W)

**Register 17.17. ASSIST_DEBUG_CORE_0_INTR_CLR_REG (0x000C)**

| 31 | | | | | | | | | | | | | | | | | | | | | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | ASSIST_DEBUG_CORE_0_SP_SPILL_MAX_CLR | (ASSIST_DEBUG_CORE_0_SP_SPILL_MIN_CLR | (ASSIST_DEBUG_CORE_0_AREA_PIF_1_WR_CLR | (ASSIST_DEBUG_CORE_0_AREA_PIF_1_RD_CLR | (ASSIST_DEBUG_CORE_0_AREA_PIF_0_WR_CLR | (ASSIST_DEBUG_CORE_0_AREA_PIF_0_RD_CLR | (ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_WR_CLR | (ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_RD_CLR | (ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_WR_CLR | (ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_RD_CLR | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_RD_CLR** Interrupt clear bit for read operations in region 0 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_0_WR_CLR** Interrupt clear bit for write operations in region 0 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_RD_CLR** Interrupt clear bit for read operations in region 1 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_DRAM0_1_WR_CLR** Interrupt clear bit for write operations in region 1 by the Data bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_RD_CLR** Interrupt clear bit for read operations in region 0 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_0_WR_CLR** Interrupt clear bit for write operations in region 0 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_RD_CLR** Interrupt clear bit for read operations in region 1 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_AREA_PIF_1_WR_CLR** Interrupt clear bit for write operations in region 1 by the Peripheral bus. (R/W)

**ASSIST_DEBUG_CORE_0_SP_SPILL_MIN_CLR** Interrupt clear bit for SP exceeding the lower bound address of SP monitored region. (R/W)

**ASSIST_DEBUG_CORE_0_SP_SPILL_MAX_CLR** Interrupt clear bit for SP exceeding the upper bound address of SP monitored region. (R/W)

**Register 17.18. ASSIST_DEBUG_CORE_0_RCD_EN_REG (0x0044)**

| (reserved) | ASSIST_DEBUG_CORE_0_RCD_PDEBUGEN | ASSIST_DEBUG_CORE_0_RCD_RECORDEN | |
|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**ASSIST_DEBUG_CORE_0_RCD_RECORDEN** Set to 1 to enable ASSIST_DEBUG_CORE_0_RCD_PDEBUGPC_REG to record PC in real time. (R/W)

**ASSIST_DEBUG_CORE_0_RCD_PDEBUGEN** Set to 1 to enable CPU debug function. The CPU outputs PC only when this field is set to 1. (R/W)

**Register 17.19. ASSIST_DEBUG_CORE_0_RCD_PDEBUGPC_REG (0x0048)**

| ASSIST_DEBUG_CORE_0_RCD_PDEBUGPC | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**ASSIST_DEBUG_CORE_0_RCD_PDEBUGPC** Records the PC value at CPU reset. (RO)

**Register 17.20. ASSIST_DEBUG_CORE_0_RCD_PDEBUGSP_REG (0x004C)**

| ASSIST_DEBUG_CORE_0_RCD_PDEBUGSP | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**ASSIST_DEBUG_CORE_0_RCD_PDEBUGSP** Records SP. (RO)

**Register 17.21. ASSIST_DEBUG_LOG_SETTING_REG (0x0070)**

| (reserved) | ASSIST_DEBUG_LOG_MEM_LOOP_ENABLE | ASSIST_DEBUG_LOG_MODE | ASSIST_DEBUG_LOG_ENA | |
|---|---|---|---|---|
| 31 … 8 | 7 | 6 … 3 | 2 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 | 0 | 0 | Reset |

**ASSIST_DEBUG_LOG_ENA** Enables the CPU bus or DMA bus access logging. bit[0]: CPU bus access logging; bit[1]: reserved; bit[2]: DMA bus access logging. (R/W)

**ASSIST_DEBUG_LOG_MODE** Configures monitoring mode. bit[0]: write monitoring; bit[1]: word monitoring; bit[2]: halfword monitoring; bit[3]: byte monitoring. (R/W)

**ASSIST_DEBUG_LOG_MEM_LOOP_ENABLE** Configures the writing mode for recorded data. 1: loop mode; 0: non-loop mode. (R/W)

**Register 17.22. ASSIST_DEBUG_LOG_DATA_0_REG (0x0074)**

| ASSIST_DEBUG_LOG_DATA_0 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**ASSIST_DEBUG_LOG_DATA_0** Specifies the monitored data. (R/W)

**Register 17.23. ASSIST_DEBUG_LOG_DATA_MASK_REG (0x0078)**

| (reserved) | ASSIST_DEBUG_LOG_DATA_SIZE | |
|---|---|---|
| 31 … 16 | 15 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**ASSIST_DEBUG_LOG_DATA_SIZE** Masks the byte specified in ASSIST_DEBUG_LOG_DATA_0_REG. (R/W)

**Register 17.24. ASSIST_DEBUG_LOG_MIN_REG (0x007C)**

| ASSIST_DEBUG_LOG_MIN | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**ASSIST_DEBUG_LOG_MIN** Configures the lower bound address of monitored address space. (R/W)

**Register 17.25. ASSIST_DEBUG_LOG_MAX_REG (0x0080)**

| ASSIST_DEBUG_LOG_MAX | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**ASSIST_DEBUG_LOG_MAX** Configures the upper bound address of monitored address space. (R/W)

**Register 17.26. ASSIST_DEBUG_LOG_MEM_START_REG (0x0084)**

| ASSIST_DEBUG_LOG_MEM_START | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**ASSIST_DEBUG_LOG_MEM_START** Configures the starting address of the storage space for recorded data. (R/W)

**Register 17.27. ASSIST_DEBUG_LOG_MEM_END_REG (0x0088)**

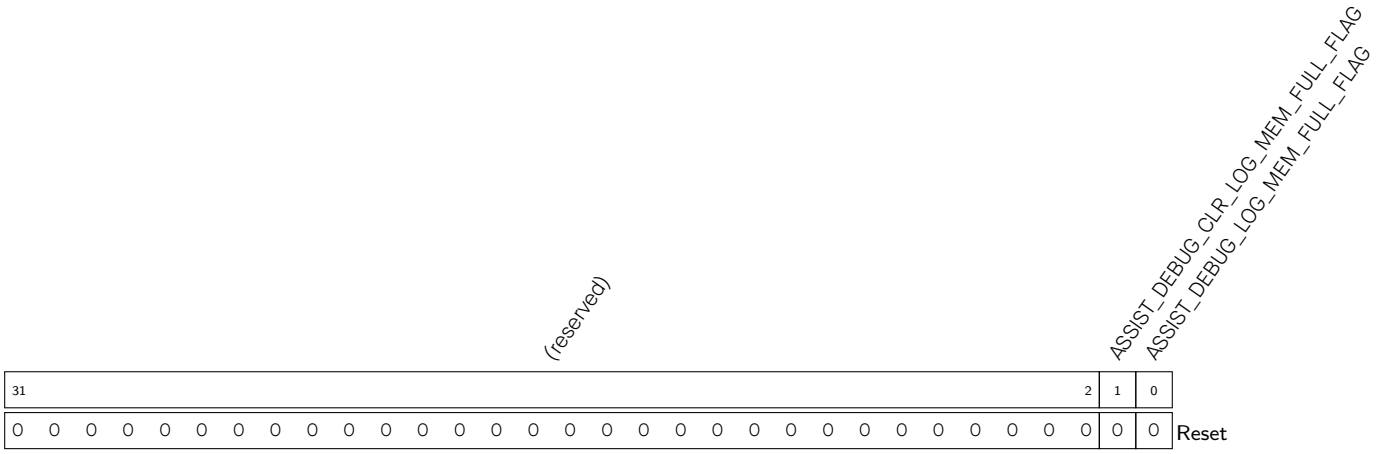
| ASSIST_DEBUG_LOG_MEM_END | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**ASSIST_DEBUG_LOG_MEM_END** Configures the end address of the storage space for recorded data. (R/W)

**Register 17.28. ASSIST_DEBUG_LOG_MEM_CURRENT_ADDR_REG (0x008C)**

**ASSIST_DEBUG_LOG_MEM_WRITING_ADDR** Indicates the address of the next write. (RO)

**Register 17.29. ASSIST_DEBUG_LOG_MEM_FULL_FLAG_REG (0x0090)**




**ASSIST_DEBUG_LOG_MEM_FULL_FLAG** The value "1" means there is a data overflow that exceeds the storage space. (RO)

**ASSIST_DEBUG_CLR_LOG_MEM_FULL_FLAG** Set to 1 to clear ASSIST_DEBUG_LOG_MEM_FULL_FLAG flag bit. Default value is "0". (R/W)

**Register 17.30. ASSIST_DEBUG_CORE_0_LASTPC_BEFORE_EXCEPTION_REG (0x0094)**

| ASSIST_DEBUG_CORE_0_LASTPC_BEFORE_EXC | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**ASSIST_DEBUG_CORE_0_LASTPC_BEFORE_EXC** Records the PC of the last command before the CPU enters exception. (RO)

**Register 17.31. ASSIST_DEBUG_CORE_0_DEBUG_MODE_REG (0x0098)**

| (reserved) | ASSIST_DEBUG_CORE_0_DEBUG_MODULE_ACTIVE | ASSIST_DEBUG_CORE_0_DEBUG_MODE | |
|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**ASSIST_DEBUG_CORE_0_DEBUG_MODE** Indicates whether the RISC-V CPU is in debug mode. 1: in debug mode; 0: not in debug mode. (RO)

**ASSIST_DEBUG_CORE_0_DEBUG_MODULE_ACTIVE** Indicates the status of the RISC-V CPU debug module. 1: active status; 0: inactive status. (RO)

**Register 17.32. ASSIST_DEBUG_DATE_REG (0x01FC)**

| (reserved) | ASSIST_DEBUG__DATE | |
|---|---|---|
| 31 28 | 27 0 | |
| 0 0 0 0 | 0x2008010 | Reset |

**ASSIST_DEBUG__DATE** Version control register. (R/W)

# Part IV

# Cryptography/Security Component

Dedicated to security features, this part explores cryptographic accelerators like SHA and AES. It also covers digital signatures, random number generation, and encryption/decryption algorithms, showcasing the SoC's capabilities in cryptography and secure data processing.

# Chapter 18

# AES Accelerator (AES)

## 18.1 Introduction

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

## 18.2 Features

The following functionality is supported:

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

## 18.3 AES Working Modes

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

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

  In this working mode, the plaintext and ciphertext is written and read via CPU directly.

- DMA-AES Working Mode:
  - Supports encryption and decryption using cryptographic keys of 128 and 256 bits, specified in NIST FIPS 197;
  - Supports block cipher modes ECB/CBC/OFB/CTR/CFB8/CFB128 under NIST SP 800-38A.

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

Users can choose the working mode for AES accelerator by configuring the AES_DMA_ENABLE_REG register according to Table 18.3-1 below.

**Table 18.3-1. AES Accelerator Working Mode**

| AES_DMA_ENABLE_REG | Working Mode |
|---|---|
| 0 | Typical AES |
| 1 | DMA-AES |

Users can choose the length of cryptographic keys and encryption / decryption by configuring the AES_MODE_REG register according to Table 18.3-2 below.

**Table 18.3-2. Key Length and Encryption/Decryption**

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

For detailed introduction on these two working modes, please refer to Section 18.4 and Section 18.5 below.

**Notice:**

ESP32-C3's Digital Signature (DS) module will call the AES accelerator. Therefore, users cannot access the AES accelerator when Digital Signature (DS) module is working.

## 18.4 Typical AES Working Mode

In the Typical AES working mode, users can check the working status of the AES accelerator by inquiring the AES_STATE_REG register and comparing the return value against the Table 18.4-1 below.

Table 18.4-1. 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. |

### 18.4.1 Key, Plaintext, and Ciphertext

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

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

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

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

### 18.4.2 Endianness

#### Text Endianness

In Typical AES working mode, the AES Accelerator uses cryptographic keys to encrypt and decrypt data in blocks of 128 bits. When filling data into AES_TEXT_IN_*m*_REG register or reading result from AES_TEXT_OUT_*m*_REG registers, users should follow the text endianness type specified in Table 18.4-2.

Table 18.4-2. Text Endianness Type for Typical AES

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

[1] The definition of "State (including c and r)" is described in Section 3.4 The State in NIST FIPS 197.
[2] Where *x* = IN or OUT.

### Key Endianness

In Typical AES working mode, when filling key into AES_KEY_*m*_REG registers, users should follow the key endianness type specified in Table 18.4-3 and Table 18.4-4.

**Table 18.4-3. Key Endianness Type for AES-128 Encryption and Decryption**

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

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

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

**Table 18.4-4. Key Endianness Type for AES-256 Encryption and Decryption**

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

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

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

### 18.4.3 Operation Process

**Single Operation**

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

**Consecutive Operations**

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

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

## 18.5 DMA-AES Working Mode

In the DMA-AES working mode, the AES accelerator supports six block cipher modes including ECB/CBC/OFB/CTR/CFB8/CFB128. Users can choose the block cipher mode by configuring the AES_BLOCK_MODE_REG register according to Table 18.5-1 below.

**Table 18.5-1. Block Cipher Mode**

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

Users can check the working status of the AES accelerator by inquiring the AES_STATE_REG register and comparing the return value against the Table 18.5-2 below.

Table 18.5-2. Working Status under DMA-AES Working mode

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

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

## 18.5.1 Key, Plaintext, and Ciphertext

**Block Operation**

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

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

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

The length of source data for AES Accelerator under DMA-AES working mode must be 128 bits or the integral multiples of 128 bits. Otherwise, trailing zeros will be added to the original source data, so the length of source data equals to the nearest integral multiples of 128 bits. Please see details in Table 18.5-3 below.

Table 18.5-3. TEXT-PADDING

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

### 18.5.2 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 18.5-4 below.

**Table 18.5-4. 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 |

### 18.5.3 Standard Incrementing Function

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

### 18.5.4 Block Number

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

### 18.5.5 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 18.5-4, 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.

### 18.5.6 Block Operation Process

1. Select one of DMA channels to connect with AES, configure the DMA chained list, and then start DMA. For details, please refer to Chapter 2 *GDMA Controller (GDMA)*.
2. Initialize the AES accelerator-related registers:
   - Write 1 to the AES_DMA_ENABLE_REG register.
   - Configure the AES_INT_ENA_REG register to enable or disable the interrupt function.
   - Initialize registers AES_MODE_REG and AES_KEY_*n*_REG.
   - Select block cipher mode by configuring the AES_BLOCK_MODE_REG register. For details, see Table 18.5-1.
   - Initialize the AES_BLOCK_NUM_REG register. For details, see Section 18.5.4.
   - Initialize the AES_INC_SEL_REG register (only needed when AES Accelerator is working under CTR block operation).
   - Initialize the AES_IV_MEM memory (This is always needed except for ECB block operation).
3. Start operation by writing 1 to the AES_TRIGGER_REG register.
4. Wait for the completion of computation, which happens when the content of AES_STATE_REG becomes 2 or the AES interrupt occurs.
5. Check if DMA completes data transmission from AES to memory. At this time, DMA had already written the result data in memory, which can be accessed directly. For details on DMA, please refer to Chapter 2 *GDMA Controller (GDMA)*.
6. Clear interrupt by writing 1 to the AES_INT_CLR_REG register, if any AES interrupt occurred during the computation.
7. Release the AES Accelerator by writing 0 to the AES_DMA_EXIT_REG register. After this, the content of the AES_STATE_REG register becomes 0. Note that, you can release DMA earlier, but only after Step 4 is completed.

## 18.6 Memory Summary

The addresses in this section are relative to the AES accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

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

## 18.7 Register Summary

The addresses in this section are relative to the AES accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

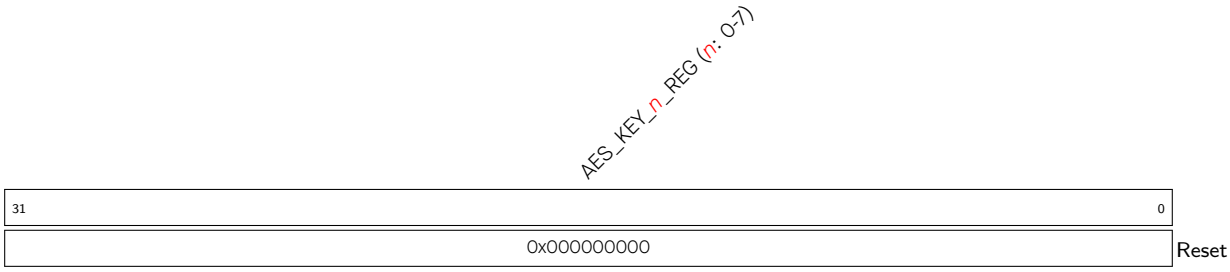
The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

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

## 18.8 Registers

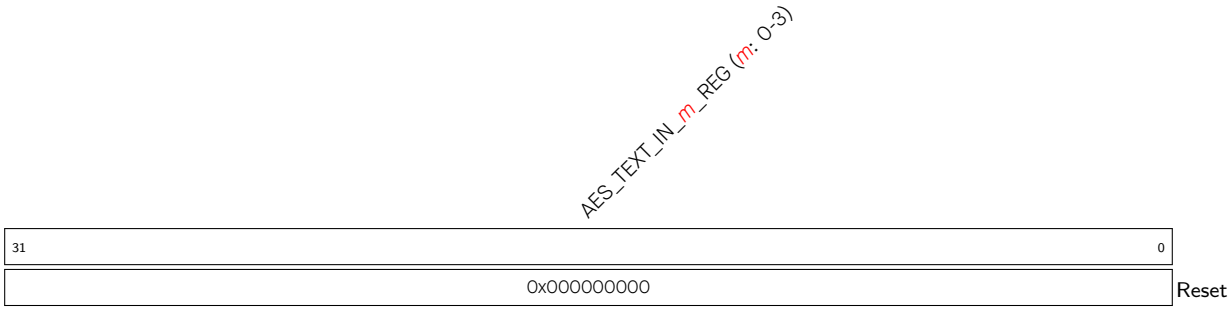
The addresses in this section are relative to the AES accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 18.1. AES_KEY_*n*_REG (*n*: 0-7) (0x0000+4**n*)**



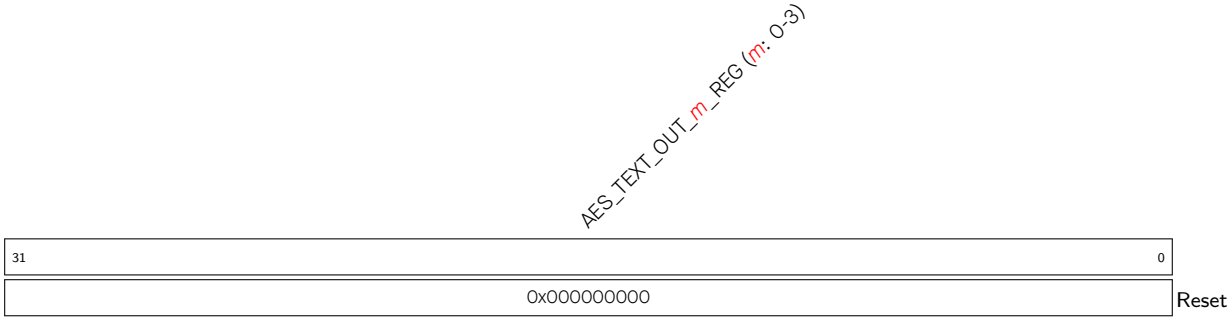

**AES_KEY_*n*_REG (*n*: 0-7)** Stores AES key data. (R/W)

**Register 18.2. AES_TEXT_IN_*m*_REG (*m*: 0-3) (0x0020+4**m*)**




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

**Register 18.3. AES_TEXT_OUT_*m*_REG (*m*: 0-3) (0x0030+4**m*)**




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

**Register 18.4. AES_MODE_REG (0x0040)**

| (reserved) | AES_MODE |
|---|---|
| 31 … 3 | 2 … 0 |
| 0x00000000 | 0 |

Reset

**AES_MODE** Defines the key length and encryption / decryption of the AES Accelerator. For details, see Table 18.3-2. (R/W)

**Register 18.5. AES_DMA_ENABLE_REG (0x0090)**

| (reserved) | AES_DMA_ENABLE |
|---|---|
| 31 … 1 | 0 |
| 0x00000000 | 0 |

Reset

**AES_DMA_ENABLE** Defines the working mode of the AES Accelerator. 0: Typical AES, 1: DMA-AES. For details, see Table 18.3-1. (R/W)

**Register 18.6. AES_BLOCK_MODE_REG (0x0094)**

| (reserved) | AES_BLOCK_MODE |
|---|---|
| 31 … 3 | 2 … 0 |
| 0x00000000 | 0 |

Reset

**AES_BLOCK_MODE** Defines the block cipher mode of the AES Accelerator operating under the DMA-AES working mode. For details, see Table 18.5-1. (R/W)

**Register 18.7. AES_BLOCK_NUM_REG (0x0098)**

| AES_BLOCK_NUM |
|---|
| 31 … 0 |
| 0x00000000 |

Reset

**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 18.5.4. (R/W)

### Register 18.8. AES_INC_SEL_REG (0x009C)

| (reserved) | AES_INC_SEL | |
|---|---|---|
| 31 … 1 | 0 | |
| 0x00000000 | 0 | Reset |

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

### Register 18.9. AES_TRIGGER_REG (0x0048)

| (reserved) | AES_TRIGGER | |
|---|---|---|
| 31 … 1 | 0 | |
| 0x00000000 | X | Reset |

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

### Register 18.10. AES_STATE_REG (0x004C)

| (reserved) | AES_STATE | |
|---|---|---|
| 31 … 2 | 1 … 0 | |
| 0x00000000 | 0x0 | Reset |

**AES_STATE** Stores the working status of the AES Accelerator. For details, see Table 18.4-1 for Typical AES working mode and Table 18.5-2 for DMA AES working mode. (RO)

### Register 18.11. AES_DMA_EXIT_REG (0x00B8)

| (reserved) | AES_DMA_EXIT | |
|---|---|---|
| 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 18.12. AES_INT_CLR_REG (0x00AC)

| (reserved) | AES_INT_CLR | |
|---|---|---|
| 31 … 1 | 0 | |
| 0x00000000 | X | Reset |

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

### Register 18.13. AES_INT_ENA_REG (0x00B0)

| (reserved) | AES_INT_ENA | |
|---|---|---|
| 31 … 1 | 0 | |
| 0x00000000 | 0 | Reset |

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

# Chapter 19

# HMAC Accelerator (HMAC)

The Hash-based Message Authentication Code (HMAC) module computes Message Authentication Codes (MACs) using Hash algorithm and keys as described in RFC 2104. The hash algorithm is SHA-256, the 256-bit HMAC key is stored in an eFuse key block and can be set as read-protected, i. e., the key is not accessible from outside the HMAC accelerator itself.

## 19.1 Main Features

- Standard HMAC-SHA-256 algorithm
- Hash result only accessible by configurable hardware peripheral (in downstream mode)
- Compatible to challenge-response authentication algorithm
- Generates required keys for the Digital Signature (DS) peripheral (in downstream mode)
- Re-enables soft-disabled JTAG (in downstream mode)

## 19.2 Functional Description

The HMAC module operates in two modes: upstream mode and downstream mode. In upstream mode, the HMAC message is provided by users and the calculation result is read back by them; in downstream mode, the HMAC module is used as a Key Derivation Function (KDF) for other internal hardware. For instance, the JTAG can be temporarily disabled by burning odd number bits of EFUSE_SOFT_DIS_JTAG in eFuse. In this case, users can temporarily re-enable JTAG using the HMAC module in downstream mode.

After the reset signal being released, the HMAC module will check whether the DS key exists in the eFuse. If the key exists, the HMAC module will enter downstream digital signature mode and finish the DS key calculation automatically.

### 19.2.1 Upstream Mode

Common use cases for the upstream mode are challenge-response protocols supporting HMAC-SHA-256. Assume the two entities in the challenge-response protocol are A and B respectively, and the data message they expect to exchange is M. The general process of this protocol is as follows:

- A calculates a unique random number M
- A sends M to B
- B calculates the HMAC (through M and KEY) and sends the result to A
- A calculates the HMAC (through M and KEY) internally

- A compares the two results. If they are the same, then the identity of B is authenticated

To calculate the HMAC value (the following steps should be done by the user):

1. Initialize the HMAC module, and enter upstream mode.
2. Write the correctly padded message to the HMAC, one block at a time.
3. Read back the result from HMAC.

For details of this process, please see Section 19.2.5.

### 19.2.2 Downstream JTAG Enable Mode

JTAG debugging can be disabled in a way which allows later re-enabling using the HMAC module. The HMAC module will expect the user to supply the HMAC result for one of the eFuse keys. The HMAC module will check whether the supplied HMAC matches the one calculated from the chosen key. If both HMACs are the same, JTAG will be enabled until the user calls the HMAC module to clear the results and consequently disable JTAG again.

There are two parameters in eFuse memory to disable JTAG: EFUSE_HARD_DIS_JTAG and EFUSE_SOFT_DIS_JTAG. Write 1 to EFUSE_DIS_PAD_JTAG to disable JTAG permanently, and write odd numbers of 1 to EFUSE_SOFT_DIS_JTAG to disable JTAG temporarily. For more details, please see Chapter 4 *eFuse Controller (EFUSE)*. After bit EFUSE_SOFT_DIS_JTAG is set, the key to re-enable JTAG can be calculated in HMAC module's downstream mode. JTAG is re-enabled when the result configured by the user is the same as the HMAC result.

To re-enable JTAG:

1. Users enable the HMAC module by initializing clock and reset signals of HMAC, and enter downstream JTAG enable mode by configuring HMAC_SET_PARA_PURPOSE_REG, then Wait for the calculation to complete. Please see Section 19.2.5 for more details.
2. Users write 1 to the HMAC_SOFT_JTAG_CTRL_REG register to enter JTAG re-enable compare mode.
3. Users write the 256-bit HMAC value which is calculated locally from the 32-byte 0x00 using SHA-256 and the generated key to register HMAC_WR_JTAG_REG by writing 8 times and 32-bit each time in big-endian word order.
4. If the HMAC result matches the value that users calculated locally, then JTAG is re-enabled. Otherwise, JTAG remains disabled.
5. After writing 1 to HMAC_SET_INVALIDATE_JTAG_REG or resetting the chip, JTAG will be disabled. If users want to re-enable JTAG again, they need to repeat the above steps again.

### 19.2.3 Downstream Digital Signature Mode

The Digital Signature (DS) module encrypts its parameters using the AES-CBC algorithm. The HMAC module is used as a Key Derivation Function (KDF) to derive the AES key to decrypt these parameters (parameter decryption key). The key used for the HMAC as KDF is stored in one of the eFuse key blocks.

Before starting the DS module, users need to obtain the parameter decryption key for the DS module through HMAC calculation. For more information, please see Chapter *22 Digital Signature (DS)*. After the chip is powered on, the HMAC module will check whether the key required to calculate the parameter decryption key

has been burned in the eFuse block. If the key has been burned, HMAC module will automatically enter the downstream digital signature mode and complete the HMAC calculation based on the chosen key.

### 19.2.4 HMAC eFuse Configuration

Each HMAC key burned into an eFuse block has a key purpose, also burned into the eFuse section. This purpose specifies for which functionality the key can be used. The HMAC module will not accept a key with a non-matching purpose for any functionality. The HMAC module provides three different functionalities: re-enabling JTAG and serving as DS KDF in downstream mode as well as pure HMAC calculation in upstream mode. For each functionality, there exists a corresponding key purpose, listed in Table 19.2-1. Additionally, another purpose specifies a key which may be used for re-enabling JTAG as well as for serving as DS KDF.

Before enabling HMAC to do calculations, user should make sure the key to be used has been burned in eFuse by reading EFUSE_KEY_PURPOSE_x (We totally have 6 keys in eFuse, so x = 0,1,2,..,5), registers from *4 eFuse Controller (EFUSE)*. Take upstream as example, if there is no EFUSE_KEY_PURPOSE_HMAC_UP in EFUSE_KEY_PURPOSE_0~5, means there is no upstream used key in efuse. You can burn key to efuse as follows:

1. Prepare a secret 256-bit HMAC key and burn the key to an empty eFuse block $y$ (there are six blocks for storing a key in eFuse. The numbers of those blocks range from 4 to 9, so y = 4,5,..,9. Hence, if we are talking about key0, we mean eFuse block4), and then program the purpose to EFUSE_KEY_PURPOSE_$(y-4)$. Take upstream mode as an example: after programming the key, the user should program EFUSE_KEY_PURPOSE_HMAC_UP (corresponding value is 6) to EFUSE_KEY_PURPOSE_$(y-4)$. Please see Chapter *4 eFuse Controller (EFUSE)* on how to program eFuse keys.
2. Configure this eFuse key block to be read protected, so that users cannot read its value. A copy of this key should be kept by any party who needs to verify this device.

Please note that the key whose purpose is EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL can be used for both re-enabling JTAG or DS.

Table 19.2-1. HMAC Purposes and Configuration Value

| Purpose | Mode | Value | Description |
|---|---|---|---|
| JTAG Re-enable | Downstream | 6 | EFUSE_KEY_PURPOSE_HMAC_DOWN_JTAG |
| DS Key Derivation | Downstream | 7 | EFUSE_KEY_PURPOSE_HMAC_DOWN_DIGITAL_SIGNATURE |
| HMAC Calculation | Upstream | 8 | EFUSE_KEY_PURPOSE_HMAC_UP |
| Both JTAG Re-enable and DS KDF | Downstream | 5 | EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL |

#### Configure HMAC Purposes

The correct purpose has to be written to register HMAC_SET_PARA_PURPOSE_REG (see Section 19.2.5). If there is no valid value in efuse purpose section, HMAC will terminate calculation.

#### Select eFuse Key Blocks

The eFuse controller provides six key blocks, i.e., KEY0 ~ 5. To select a particular KEYn for an HMAC calculation, write the key number n to register HMAC_SET_PARA_KEY_REG.

Note that the purpose of the key has also been programmed to eFuse memory. Only when the configured HMAC purpose matches the defined purpose of KEYn, will the HMAC module execute the configured calculation. Otherwise, it will return a matching error and stop the current calculation. For example, suppose a user selects KEY3 for HMAC calculation, and the value programmed to KEY_PURPOSE_3 is 6 (EFUSE_KEY_PURPOSE_HMAC_DOWN_JTAG). Based on Table 19.2-1, KEY3 can be used to re-enable JTAG. If the value written to register HMAC_SET_PARA_PURPOSE_REG is also 6, then the HMAC module will start the process to re-enable JTAG.

### 19.2.5 HMAC Process (Detailed)

The process to call HMAC is as follows:

1. Enable HMAC module
   (a) Set the peripheral clock bits for HMAC and SHA peripherals in register SYSTEM_PERIP_CLK_EN1_REG, and clear the corresponding peripheral reset bits in register SYSTEM_PERIP_RST_EN1_REG. For information on those registers, please see Chapter 3 *System and Memory*.
   (b) Write 1 to register HMAC_SET_START_REG.
2. Configure HMAC keys and key purposes
   (a) Write the key purpose $m$ to register HMAC_SET_PARA_PURPOSE_REG. The possible key purpose values are shown in Table 19.2-1. For more information, please refer to Section 19.2.4.
   (b) Select KEYn in eFuse memory as the key by writing n (ranges from 0 to 5) to register HMAC_SET_PARA_KEY_REG. For more information, please refer to Section 19.2.4.
   (c) Write 1 to register HMAC_SET_PARA_FINISH_REG to complete the configuration.
   (d) Read register HMAC_QUERY_ERROR_REG. If its value is 1, it means the purpose of the selected block does not match the configured key purpose and the calculation will not proceed. If its value is 0, it means the purpose of the selected block matches the configured key purpose, and then the calculation can proceed.
   (e) When the value of HMAC_SET_PARA_PURPOSE_REG is not 8, it means the HMAC module is in downstream mode, proceed with step 3. When the value is 8, it means the HMAC module is in upstream mode, proceed with step 4.
3. Downstream mode
   (a) Poll Status register HMAC_QUERY_BUSY_REG until it reads 0.
   (b) To clear the result and make further usage of the dependent hardware (JTAG or DS) impossible, write 1 to either register HMAC_SET_INVALIDATE_JTAG_REG to clear the result generated by the JTAG key; or to register HMAC_SET_INVALIDATE_DS_REG to clear the result generated by DS key. Afterwards, the HMAC Process needs to be restarted to re-enable any of the dependent peripherals.
4. Transmit message block Block_n ($n >= 1$) in upstream mode
   (a) Poll Status register HMAC_QUERY_BUSY_REG until it reads 0.
   (b) Write the 512-bit Block_n to register HMAC_WDATA0~15_REG. Write 1 to register HMAC_SET_MESSAGE_ONE_REG, to trigger the processing of this message block.

(c) Poll Status register HMAC_QUERY_BUSY_REG until it reads 0.

(d) Different message blocks will be generated, depending on whether the size of the to-be-processed message is a multiple of 512 bits.

- If the bit length of the message is a multiple of 512 bits, there are three possible options:
  i. If Block_n+1 exists, write 1 to register HMAC_SET_MESSAGE_ING_REG to make $n = n + 1$, and then jump to step 4.(b).
  ii. If Block_n is the last block of the message and users expects to apply SHA padding in hardware, write 1 to register HMAC_SET_MESSAGE_END_REG, and then jump to step 6.
  iii. If Block_n is the last block of the padded message and SHA padding has been applied by users, write 1 to register HMAC_SET_MESSAGE_PAD_REG, and then jump to step 5.
- If the bit length of the message is not a multiple of 512 bits, there are three possible options as follows. Note that in this case, the user is required to apply SHA padding to the message, after which the padded message length should be a multiple of 512 bits.
  i. If there is only one message block in total which has included all padding bits, write 1 to register HMAC_ONE_BLOCK_REG, and then jump to step 6.
  ii. If Block_n is the second last padded block, write 1 to register HMAC_SET_MESSAGE_PAD_REG, and then jump to step 5.
  iii. If Block_n is neither the last nor the second last message block, write 1 to register HMAC_SET_MESSAGE_ING_REG and define $n = n + 1$, and then jump to step 4.(b).

5. Apply SHA padding to message

(a) Users apply SHA padding to the last message block as described in Section 19.3.1, write this block to register HMAC_WDATA0~15_REG, and then write 1 to register HMAC_SET_MESSAGE_ONE_REG. Then the HMAC module will process this message block.

(b) Jump to step 6.

6. Read hash result in upstream mode

(a) Poll Status register HMAC_QUERY_BUSY_REG until it reads 0.

(b) Read hash result from register HMAC_RDATA0~7_REG.

(c) Write 1 to register HMAC_SET_RESULT_FINISH_REG to finish calculation. The result will be cleared at the same time.

(d) Upstream mode operation is completed.

**Note:**
The SHA accelerator can be called directly, or used internally by the DS module and the HMAC module. However, they can not share the hardware resources simultaneously. Therefore, the SHA module must not be called neither by the CPU nor by the DS module when the HMAC module is in use.

## 19.3 HMAC Algorithm Details

### 19.3.1 Padding Bits

The HMAC module uses SHA-256 as hash algorithm. If the input message is not a multiple of 512 bits, the user must apply a SHA-256 padding algorithm in software. The SHA-256 padding algorithm is the same as described in Section *Padding the Message* of FIPS PUB 180-4. In downstream mode, users do not need to input any message or apply padding. The HMAC module uses a default 32-byte pattern of 0x00 for re-enabling JTAG and a 32-byte pattern of 0xff for deriving the AES key for the DS module.

As shown in Figure 19.3-1, suppose the length of the unpadded message is $m$ bits. Padding steps are as follows:

1. Append one bit of value "1" to the end of the unpadded message;
2. Append $k$ bits of value "0", where $k$ is the smallest non-negative number which satisfies $m + 1 + k \equiv 448 (mod 512)$;
3. Append a 64-bit integer value as a binary block. This block consists of the length of the unpadded message as a big-endian binary integer value $m$.

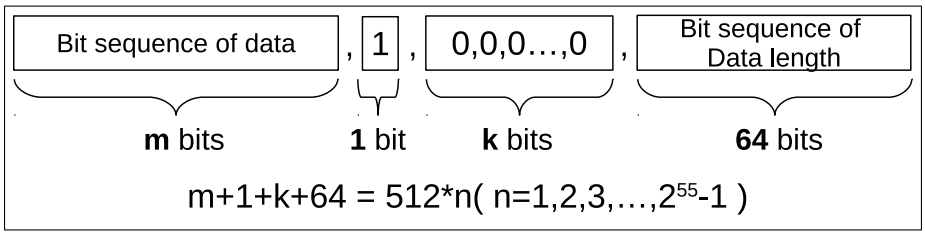



Figure 19.3-1. HMAC SHA-256 Padding Diagram

In upstream mode, if the length of the unpadded message is a multiple of 512 bits, users can configure hardware to apply SHA padding by writing 1 to HMAC_SET_MESSGAE_END_REG or do padding work themselves by writing 1 to HMAC_SET_MESSAGE_PAD_REG. If the length is not a multiple of 512 bits, SHA padding must be manually applied by the user. After the user prepared the padding data, they should complete the subsequent configuration according to the Section 19.2.5.

### 19.3.2 HMAC Algorithm Structure

The structure of the implemented algorithm in the HMAC module is shown in Figure 19.3-2. This is the standard HMAC algorithm as described in RFC 2104.

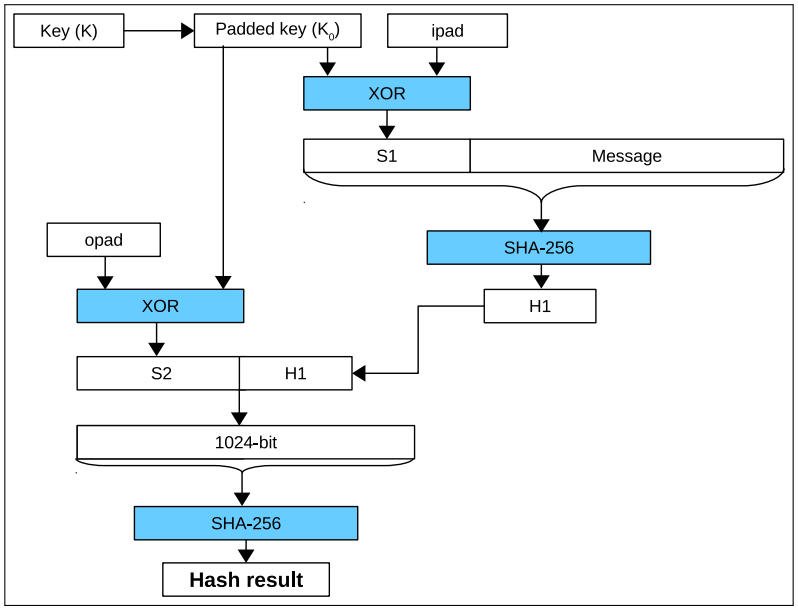



Figure 19.3-2. HMAC Structure Schematic Diagram

In Figure 19.3-2:

1. ipad is a 512-bit message block composed of 64 bytes of 0x36.
2. opad is a 512-bit message block composed of 64 bytes of 0x5c.

The HMAC module appends a 256-bit 0 sequence after the bit sequence of the 256-bit key k in order to get a 512-bit $K_0$. Then, the HMAC module XORs $K_0$ with ipad to get the 512-bit S1. Afterwards, the HMAC module appends the input message (multiple of 512 bits) after the 512-bit S1, and exercises the SHA-256 algorithm to get the 256-bit H1.

The HMAC module appends the 256-bit SHA-256 hash result H1 to the 512-bit S2 value, which is calculated using the XOR operation of $K_0$ and opad. A 768-bit sequence will be generated. Then, the HMAC module uses the SHA padding algorithm described in Section 19.3.1 to pad the 768-bit sequence to a 1024-bit sequence, and applies the SHA-256 algorithm to get the final hash result (256-bit).

## 19.4 Register Summary

The addresses in this section are relative to HMAC Accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Control/Status Registers** | | | |
| HMAC_SET_START_REG | HMAC start control register | 0x0040 | WO |
| HMAC_SET_PARA_FINISH_REG | HMAC configuration completion register | 0x004C | WO |
| HMAC_SET_MESSAGE_ONE_REG | HMAC message control register | 0x0050 | WO |
| HMAC_SET_MESSAGE_ING_REG | HMAC message continue register | 0x0054 | WO |
| HMAC_SET_MESSAGE_END_REG | HMAC message end register | 0x0058 | WO |
| HMAC_SET_RESULT_FINISH_REG | HMAC result reading finish register | 0x005C | WO |
| HMAC_SET_INVALIDATE_JTAG_REG | Invalidate JTAG result register | 0x0060 | WO |
| HMAC_SET_INVALIDATE_DS_REG | Invalidate digital signature result register | 0x0064 | WO |
| HMAC_QUERY_ERROR_REG | Stores matching results between keys generated by users and corresponding purposes | 0x0068 | RO |
| HMAC_QUERY_BUSY_REG | Busy state of HMAC module | 0x006C | RO |
| **configuration Registers** | | | |
| HMAC_SET_PARA_PURPOSE_REG | HMAC parameter configuration register | 0x0044 | WO |
| HMAC_SET_PARA_KEY_REG | HMAC parameters configuration register | 0x0048 | WO |
| HMAC_SOFT_JTAG_CTRL_REG | Re-enable JTAG register 0 | 0x00F8 | WO |
| HMAC_WR_JTAG_REG | Re-enable JTAG register 1 | 0x00FC | WO |
| **HMAC Message Block** | | | |
| HMAC_WR_MESSAGE_0_REG | Message register 0 | 0x0080 | WO |
| HMAC_WR_MESSAGE_1_REG | Message register 1 | 0x0084 | WO |
| HMAC_WR_MESSAGE_2_REG | Message register 2 | 0x0088 | WO |
| HMAC_WR_MESSAGE_3_REG | Message register 3 | 0x008C | WO |
| HMAC_WR_MESSAGE_4_REG | Message register 4 | 0x0090 | WO |
| HMAC_WR_MESSAGE_5_REG | Message register 5 | 0x0094 | WO |
| HMAC_WR_MESSAGE_6_REG | Message register 6 | 0x0098 | WO |
| HMAC_WR_MESSAGE_7_REG | Message register 7 | 0x009C | WO |
| HMAC_WR_MESSAGE_8_REG | Message register 8 | 0x00A0 | WO |
| HMAC_WR_MESSAGE_9_REG | Message register 9 | 0x00A4 | WO |
| HMAC_WR_MESSAGE_10_REG | Message register 10 | 0x00A8 | WO |
| HMAC_WR_MESSAGE_11_REG | Message register 11 | 0x00AC | WO |
| HMAC_WR_MESSAGE_12_REG | Message register 12 | 0x00B0 | WO |
| HMAC_WR_MESSAGE_13_REG | Message register 13 | 0x00B4 | WO |
| HMAC_WR_MESSAGE_14_REG | Message register 14 | 0x00B8 | WO |
| HMAC_WR_MESSAGE_15_REG | Message register 15 | 0x00BC | WO |
| **HMAC Upstream Result** | | | |
| HMAC_RD_RESULT_0_REG | Hash result register 0 | 0x00C0 | RO |
| HMAC_RD_RESULT_1_REG | Hash result register 1 | 0x00C4 | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| HMAC_RD_RESULT_2_REG | Hash result register 2 | 0x00C8 | RO |
| HMAC_RD_RESULT_3_REG | Hash result register 3 | 0x00CC | RO |
| HMAC_RD_RESULT_4_REG | Hash result register 4 | 0x00D0 | RO |
| HMAC_RD_RESULT_5_REG | Hash result register 5 | 0x00D4 | RO |
| HMAC_RD_RESULT_6_REG | Hash result register 6 | 0x00D8 | RO |
| HMAC_RD_RESULT_7_REG | Hash result register 7 | 0x00DC | RO |
| **Control/Status Registers** | | | |
| HMAC_SET_MESSAGE_PAD_REG | Software padding register | 0x00F0 | WO |
| HMAC_ONE_BLOCK_REG | One block message register | 0x00F4 | WO |
| **Version Register** | | | |
| HMAC_DATE_REG | Version control register | 0x00F8 | R/W |

## 19.5 Registers

The addresses in this section are relative to HMAC Accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 19.1. HMAC_SET_START_REG (0x0040)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | HMAC_SET_START | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**HMAC_SET_START** Set this bit to enable HMAC. (WO)

**Register 19.2. HMAC_SET_PARA_FINISH_REG (0x004C)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | HMAC_SET_PARA_END | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**HMAC_SET_PARA_END** Set this bit to finish HMAC configuration. (WO)

**Register 19.3. HMAC_SET_MESSAGE_CALC_BLOCK_REG (0x0050)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | HMAC_SET_TEXT_ONE | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**HMAC_SET_TEXT_ONE** Call SHA to calculate one message block. (WO)

Register 19.4. HMAC_SET_MESSAGE_ING_REG (0x0054)

| (reserved) | HMAC_SET_TEXT_ING |
|---|---|
| 31 … 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 |
| | Reset |

**HMAC_SET_TEXT_ING** Set this bit to show there are still some message blocks to be processed. (WO)

Register 19.5. HMAC_SET_MESSAGE_END_REG (0x0058)

| (reserved) | HMAC_SET_TEXT_END |
|---|---|
| 31 … 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 |
| | Reset |

**HMAC_SET_TEXT_END** Set this bit to start hardware padding. (WO)

Register 19.6. HMAC_SET_RESULT_FINISH_REG (0x005C)

| (reserved) | HMAC_SET_RESULT_END |
|---|---|
| 31 … 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 |
| | Reset |

**HMAC_SET_RESULT_END** Set this bit to exit upstream mode and clear calculation results. (WO)

**Register 19.7. HMAC_SET_INVALIDATE_JTAG_REG (0x0060)**

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | HMAC_SET_INVALIDATE_JTAG | |
| 0 0 0 0 0 0 0 0 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 |

**HMAC_SET_INVALIDATE_JTAG** Set this bit to clear calculation results when re-enabling JTAG in downstream mode. (WO)

**Register 19.8. HMAC_SET_INVALIDATE_DS_REG (0x0064)**

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | HMAC_SET_INVALIDATE_DS | |
| 0 0 0 0 0 0 0 0 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 |

**HMAC_SET_INVALIDATE_DS** Set this bit to clear calculation results of the DS module in downstream mode. (WO)

**Register 19.9. HMAC_QUERY_ERROR_REG (0x0068)**

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | HMAC_QUREY_CHECK | |
| 0 0 0 0 0 0 0 0 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 |

**HMAC_QUREY_CHECK** Indicates whether an HMAC key matches the purpose.(RO)

- 0: HMAC key and purpose match.
- 1: error.

**Register 19.10. HMAC_QUERY_BUSY_REG (0x006C)**

| (reserved) | HMAC_BUSY_STATE | |
|---|---|---|
| 31 … 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 | Reset |

**HMAC_BUSY_STATE** Indicates whether HMAC is in busy state. Before configuring HMAC, please make sure HMAC is in IDLE state. (RO)

- 0: idle.
- 1: HMAC is still working on calculation.

**Register 19.11. HMAC_SET_PARA_PURPOSE_REG (0x0044)**

| (reserved) | HMAC_PURPOSE_SET | |
|---|---|---|
| 31 … 4 | 3 … 0 | |
| 0 0 0 0 0 0 0 0 0 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 |

**HMAC_PURPOSE_SET** Determines the HMAC purpose, refer to the Table 19.2-1. (WO)

**Register 19.12. HMAC_SET_PARA_KEY_REG (0x0048)**

| (reserved) | HMAC_KEY_SET | |
|---|---|---|
| 31 … 3 | 2 … 0 | |
| 0 0 0 0 0 0 0 0 0 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 |

**HMAC_KEY_SET** Selects HMAC key. There are six keys with index 0~5. Write the index of the selected key to this field. (WO)

**Register 19.13. HMAC_WR_MESSAGE_*n*_REG (*n*: 0-15) (0x0080+4**n*)**

| HMAC_WDATA_0 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**HMAC_WDATA_*n*** Store the *n*th 32-bit of message. (WO)

**Register 19.14. HMAC_RD_RESULT_*n*_REG (*n*: 0-7) (0x00C0+4**n*)**

| HMAC_RDATA_0 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**HMAC_RDATA_*n*** Read the *n*th 32-bit of hash result. (RO)

**Register 19.15. HMAC_SET_MESSAGE_PAD_REG (0x00F0)**

| (reserved) | HMAC_SET_TEXT_PAD | |
|---|---|---|
| 31 … 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 | Reset |

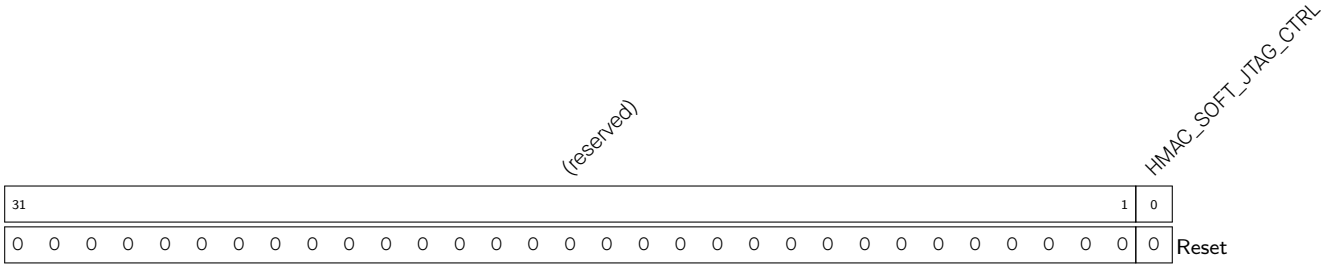
**HMAC_SET_TEXT_PAD** Set this bit to indicate that padding is applied by software. (WO)

**Register 19.16. HMAC_ONE_BLOCK_REG (0x00F4)**

| (reserved) | HMAC_SET_ONE_BLOCK | |
|---|---|---|
| 31 … 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 | Reset |

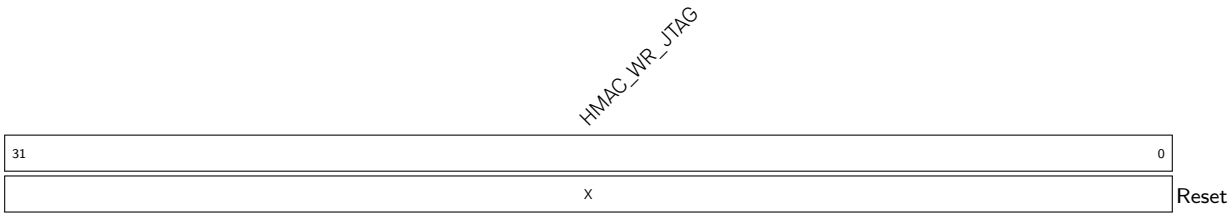
**HMAC_SET_ONE_BLOCK** Set this bit when there is only one block which already contins padding bits. (WO)

### Register 19.17. HMAC_SOFT_JTAG_CTRL_REG (0x00F8)



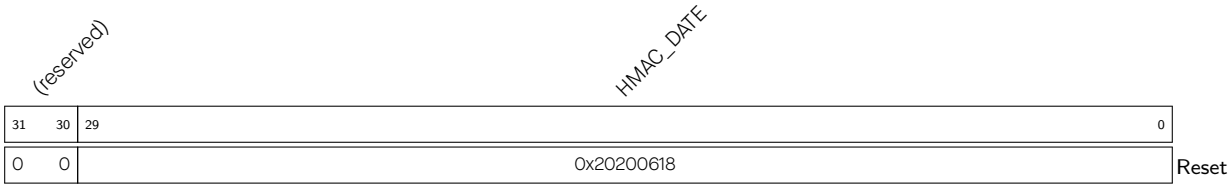

**HMAC_SOFT_JTAG_CTRL** Set this bit to enable JTAG authentication mode. (WO)

### Register 19.18. HMAC_WR_JTAG_REG (0x00FC)




**HMAC_WR_JTAG** Set this field to re-enable the JTAG comparing input register. (WO)

### Register 19.19. HMAC_DATE_REG (0x00F8)




**HMAC_DATE** Version control register. (R/W)

# Chapter 20

# RSA Accelerator (RSA)

## 20.1 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.

## 20.2 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

## 20.3 Functional Description

The RSA Accelerator is activated by setting the SYSTEM_CRYPTO_RSA_CLK_EN bit in the SYSTEM_PERIP_CLK
_EN1_REG register and clearing the SYSTEM_RSA_MEM_PD bit in the SYSTEM_RSA_PD_CTRL_REG register. This releases the RSA Accelerator from reset.

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

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

**Notice:**

ESP32-C3's Digital Signature (DS) module also calls the RSA accelerator. Therefore, users cannot access the RSA

accelerator when Digital Signature (DS) is working.

### 20.3.1 Large Number Modular Exponentiation

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

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

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

$$b = 2^{32}$$

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

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

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

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

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

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

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

Large-number modular exponentiation can be implemented as follows:

1. Write 1 or 0 to the RSA_INTERRUPT_ENA_REG register to enable or disable the interrupt function.
2. Configure relevant registers:
   (a) Write $(\frac{N}{32} - 1)$ to the RSA_MODE_REG register.
   (b) Write $M'$ to the RSA_M_PRIME_REG register.

(c) Configure registers related to the acceleration options, which are described later in Section 20.3.4.

3. Write $X_i$, $Y_i$, $M_i$ and $\overline{r}_i$ for $i \in \{0, 1, \ldots, n-1\}$ to memory blocks RSA_X_MEM, RSA_Y_MEM, RSA_M_MEM and RSA_Z_MEM. The capacity of each memory block is 96 words. Each word of each memory block can store one base-$b$ digit. The memory blocks use the little endian format for storage, i.e. the least significant digit of each number is in the lowest address.

   Users need to write data to each memory block only according to the length of the number; data beyond this length are ignored.

4. Write 1 to the RSA_MODEXP_START_REG register to start computation.
5. Wait for the completion of computation, which happens when the content of RSA_IDLE_REG becomes 1 or the RSA interrupt occurs.
6. Read the result $Z_i$ for $i \in \{0, 1, \ldots, n-1\}$ from RSA_Z_MEM.
7. Write 1 to RSA_CLEAR_INTERRUPT_REG to clear the interrupt, if you have enabled the interrupt function.

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

### 20.3.2 Large Number Modular Multiplication

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

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

The computation can be executed as follows:

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

   Users need to write data to each memory block only according to the length of the number; data beyond this length are ignored.

4. Write 1 to the RSA_MODMULT_START_REG register.
5. Wait for the completion of computation, which happens when the content of RSA_IDLE_REG becomes 1 or the RSA interrupt occurs.
6. Read the result $Z_i$ for $i \in \{0, 1, \ldots, n-1\}$ from RSA_Z_MEM.

7. Write 1 to RSA_CLEAR_INTERRUPT_REG to clear the interrupt, if you have enabled the interrupt function.

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

### 20.3.3 Large Number Multiplication

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

The computation can be executed as follows:

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

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

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

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

### 20.3.4 Options for Acceleration

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

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

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

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

where,

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

When either of these two options is configured for acceleration:

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

We provide an example to demonstrate the performance of the RSA Accelerator under different combinations of SEARCH and CONSTANT_TIME configuration. Here we perform $Z = X^Y$ mod $M$ with $N$ = 3072 and $Y$ = 65537. Table 20.3-1 below demonstrates the time costs under different combinations of SEARCH and CONSTANT_TIME configuration. Here, we should also mention that, $\alpha$ is set to 16 when the SEARCH option is enabled.

**Table 20.3-1. Acceleration Performance**

| SEARCH Option | CONSTANT_TIME Option | Time Cost (ms) |
|---|---|---|
| No acceleration | No acceleration | 752.81 |
| Accelerated | No acceleration | 4.52 |
| No acceleration | Acceleration | 2.406 |
| Acceleration | Acceleration | 2.33 |

It's obvious that:

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

## 20.4 Memory Summary

The addresses in this section are relative to the RSA accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

**Table 20.4-1. RSA Accelerator Memory Blocks**

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

## 20.5 Register Summary

The addresses in this section are relative to the RSA accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

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

## 20.6 Registers

The addresses in this section are relative to the RSA accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 20.1. RSA_M_PRIME_REG (0x0800)**

| RSA_M_PRIME_REG | |
|---|---|
| 31 … 0 | |
| 0x000000000 | Reset |

**RSA_M_PRIME_REG** Stores M'.(R/W)

**Register 20.2. RSA_MODE_REG (0x0804)**

| (reserved) | RSA_MODE | |
|---|---|---|
| 31 … 7 | 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 | Reset |

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

**Register 20.3. RSA_CLEAN_REG (0x0808)**

| (reserved) | RSA_CLEAN | |
|---|---|---|
| 31 … 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 | Reset |

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

### Register 20.4. RSA_MODEXP_START_REG (0x080C)

| (reserved) | RSA_MODEXP_START | |
|---|---|---|
| 31 … 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 | Reset |

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

### Register 20.5. RSA_MODMULT_START_REG (0x0810)

| (reserved) | RSA_MODMULT_START | |
|---|---|---|
| 31 … 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 | Reset |

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

### Register 20.6. RSA_MULT_START_REG (0x0814)

| (reserved) | RSA_MULT_START | |
|---|---|---|
| 31 … 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 | Reset |

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

### Register 20.7. RSA_IDLE_REG (0x0818)

| (reserved) | RSA_IDLE | |
|---|---|---|
| 31 … 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 | Reset |

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

### Register 20.8. RSA_CLEAR_INTERRUPT_REG (0x081C)

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | RSA_CLEAR_INTERRUPT | |
| 0 0 0 0 0 0 0 0 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 |

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

### Register 20.9. RSA_CONSTANT_TIME_REG (0x0820)

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | RSA_CONSTANT_TIME | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | | 1 | Reset |

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

### Register 20.10. RSA_SEARCH_ENABLE_REG (0x0824)

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | RSA_SEARCH_ENABLE | |
| 0 0 0 0 0 0 0 0 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 |

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

### Register 20.11. RSA_SEARCH_POS_REG (0x0828)

| (reserved) | RSA_SEARCH_POS | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 20.12. RSA_INTERRUPT_ENA_REG (0x082C)

| (reserved) | RSA_INTERRUPT_ENA | |
|---|---|---|
| 31 … 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 | 1 | Reset |

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

### Register 20.13. RSA_DATE_REG (0x0830)

| (reserved) | RSA_DATE | |
|---|---|---|
| 31 30 | 29 … 0 | |
| 0 0 | 0x20200618 | Reset |

**RSA_DATE** Version control register. (R/W)

# Chapter 21

# SHA Accelerator (SHA)

## 21.1 Introduction

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

## 21.2 Features

The following functionality is supported:

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

## 21.3 Working Modes

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

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

Users can start the SHA accelerator with different working modes by configuring registers SHA_START_REG and SHA_DMA_START_REG. For details, please see Table 21.3-1.

**Table 21.3-1. SHA Accelerator Working Mode**

| Working Mode | Configuration Method |
|---|---|
| Typical SHA | Set SHA_START_REG to 1 |

| | |
|---|---|
| DMA-SHA | Set SHA_DMA_START_REG to 1 |

Users can choose hash algorithms by configuring the SHA_MODE_REG register. For details, please see Table 21.3-2.

**Table 21.3-2. SHA Hash Algorithm Selection**

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

**Notice:**

ESP32-C3's Digital Signature (DS) and HMAC Accelerator (HMAC) modules also call the SHA accelerator. Therefore, users cannot access the SHA accelerator when these modules are working.

## 21.4 Function Description

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

### 21.4.1 Preprocessing

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

#### 21.4.1.1 Padding the Message

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

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

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

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

#### 21.4.1.2 Parsing the Message

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

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

**Note:**

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

#### 21.4.1.3 Setting the Initial Hash Value

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

### 21.4.2 Hash Operation

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

#### 21.4.2.1 Typical SHA Mode Process

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

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

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

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

**Typical SHA Process**

1. Select a hash algorithm.
   - Configure the SHA_MODE_REG register based on Table 21.3-2.
2. Process the current message block [1].
   - Write the message block in registers SHA_M_*n*_REG.
3. Start the SHA accelerator.
   - If this is the first time to execute this step, set the SHA_START_REG register to 1 to start the SHA accelerator. In this case, the accelerator uses the initial hash value stored in hardware for a given algorithm configured in Step 1 to start the calculation;

- If this is not the first time to execute this step[2], set the SHA_CONTINUE_REG register to 1 to start the SHA accelerator. In this case, the accelerator uses the hash value stored in the SHA_H_*n*_REG register to start calculation.

4. Check the progress of the current message block.
   - Poll register SHA_BUSY_REG until the content of this register becomes 0, indicating the accelerator has completed the calculation for the current message block and now is in the "idle" status [3].
5. Decide if you have more message blocks to process:
   - If yes, please go back to Step 2.
   - Otherwise, please continue.
6. Obtain the message digest.
   - Read the message digest from registers SHA_H_*n*_REG.

**Note:**

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

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

The process to implement interleaved calculation is described below.

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

#### 21.4.2.2 DMA-SHA Mode Process

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

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

In contrast to the Typical SHA working mode, when the SHA accelerator is working under the DMA-SHA mode, all data read are completed via DMA. Therefore, users are required to configure the DMA controller following the description in Chapter 2 *GDMA Controller (GDMA)*.

**DMA-SHA process**

1. Select a hash algorithm.
   - Select a hash algorithm by configuring the SHA_MODE_REG register. For details, please refer to Table 21.3-2.
2. Configure the SHA_INT_ENA_REG register to enable or disable interrupt (Set 1 to enable).
3. Configure the number of message blocks.
   - Write the number of message blocks $M$ to the SHA_DMA_BLOCK_NUM_REG register.
4. Start the DMA-SHA calculation.
   - If the current DMA-SHA calculation follows a previous calculation, firstly write the message digest from the previous calculation to registers SHA_H_*n*_REG, then write 1 to register SHA_DMA_CONTINUE_REG to start SHA accelerator;
   - Otherwise, write 1 to register SHA_DMA_START_REG to start the accelerator.
5. Wait till the completion of the DMA-SHA calculation, which happens when:
   - The content of SHA_BUSY_REG register becomes 0, or
   - An SHA interrupt occurs. In this case, please clear interrupt by writing 1 to the SHA_INT_CLEAR_REG register.
6. Obtain the message digest:
   - Read the message digest from registers SHA_H_*n*_REG.

### 21.4.3 Message Digest

After the hash task completes, the SHA accelerator writes the message digest from the task to registers SHA_H_*n*_REG(*n*: 0~7). The lengths of the generated message digest are different depending on different hash algorithms. For details, see Table 21.4-1 below:

Table 21.4-1. The Storage and Length of Message Digest from Different Algorithms

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

[1] The message digest is stored in registers from most significant bits to the least significant bits, with the first word stored in register SHA_H_0_REG and the second word stored in register SHA_H_1_REG... For details, please see subsection 21.4.1.2.

### 21.4.4 Interrupt

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

## 21.5 Register Summary

The addresses in this section are relative to the SHA accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

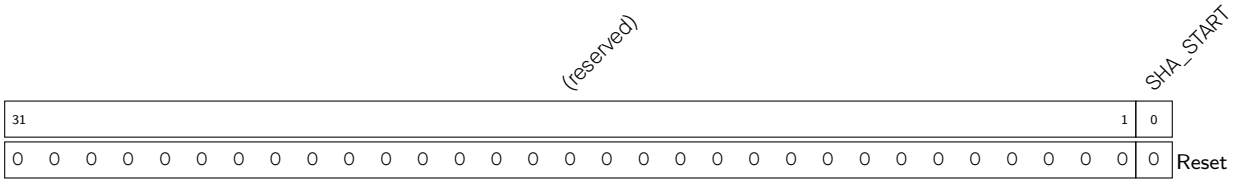
| Name | Description | Address | Access |
|---|---|---|---|
| **Control/Status registers** | | | |
| SHA_CONTINUE_REG | Continues SHA operation (only effective in Typical SHA mode) | 0x0014 | WO |
| SHA_BUSY_REG | Indicates if SHA Accelerator is busy or not | 0x0018 | RO |
| SHA_DMA_START_REG | Starts the SHA accelerator for DMA-SHA operation | 0x001C | WO |
| SHA_START_REG | Starts the SHA accelerator for Typical SHA operation | 0x0010 | WO |
| SHA_DMA_CONTINUE_REG | Continues SHA operation (only effective in DMA-SHA mode) | 0x0020 | WO |
| SHA_INT_CLEAR_REG | DMA-SHA interrupt clear register | 0x0024 | WO |
| SHA_INT_ENA_REG | DMA-SHA interrupt enable register | 0x0028 | R/W |
| **Version Register** | | | |
| SHA_DATE_REG | Version control register | 0x002C | R/W |
| **Configuration Registers** | | | |
| SHA_MODE_REG | Defines the algorithm of SHA accelerator | 0x0000 | R/W |
| **Data Registers** | | | |
| SHA_DMA_BLOCK_NUM_REG | Block number register (only effective for DMA-SHA) | 0x000C | R/W |
| SHA_H_0_REG | Hash value | 0x0040 | R/W |
| SHA_H_1_REG | Hash value | 0x0044 | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| SHA_H_2_REG | Hash value | 0x0048 | R/W |
| SHA_H_3_REG | Hash value | 0x004C | R/W |
| SHA_H_4_REG | Hash value | 0x0050 | R/W |
| SHA_H_5_REG | Hash value | 0x0054 | R/W |
| SHA_H_6_REG | Hash value | 0x0058 | R/W |
| SHA_H_7_REG | Hash value | 0x005C | R/W |
| SHA_M_0_REG | Message | 0x0080 | R/W |
| SHA_M_1_REG | Message | 0x0084 | R/W |
| SHA_M_2_REG | Message | 0x0088 | R/W |
| SHA_M_3_REG | Message | 0x008C | R/W |
| SHA_M_4_REG | Message | 0x0090 | R/W |
| SHA_M_5_REG | Message | 0x0094 | R/W |
| SHA_M_6_REG | Message | 0x0098 | R/W |
| SHA_M_7_REG | Message | 0x009C | R/W |
| SHA_M_8_REG | Message | 0x00A0 | R/W |
| SHA_M_9_REG | Message | 0x00A4 | R/W |
| SHA_M_10_REG | Message | 0x00A8 | R/W |
| SHA_M_11_REG | Message | 0x00AC | R/W |
| 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 |

## 21.6 Registers

The addresses in this section are relative to the SHA accelerator base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 21.1. SHA_START_REG (0x0010)**




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

### Register 21.2. SHA_CONTINUE_REG (0x0014)

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | SHA_CONTINUE | |
| 0 0 0 0 0 0 0 0 0 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 |

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

### Register 21.3. SHA_BUSY_REG (0x0018)

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | SHA_BUSY_STATE | |
| 0 0 0 0 0 0 0 0 0 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 |

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

### Register 21.4. SHA_DMA_START_REG (0x001C)

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | SHA_DMA_START | |
| 0 0 0 0 0 0 0 0 0 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 |

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

### Register 21.5. SHA_DMA_CONTINUE_REG (0x0020)

| 31 | 1 | 0 | |
|---|---|---|---|
| (reserved) | | SHA_DMA_CONTINUE | |
| 0 0 0 0 0 0 0 0 0 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 |

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

### Register 21.6. SHA_INT_CLEAR_REG (0x0024)

| (reserved) | SHA_CLEAR_INTERRUPT | |
|---|---|---|
| 31 … 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 | Reset |

**SHA_CLEAR_INTERRUPT** Clears DMA-SHA interrupt. (WO)

### Register 21.7. SHA_INT_ENA_REG (0x0028)

| (reserved) | SHA_INTERRUPT_ENA | |
|---|---|---|
| 31 … 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 | Reset |

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

### Register 21.8. SHA_DATE_REG (0x002C)

| (reserved) | SHA_DATE | |
|---|---|---|
| 31 30 | 29 … 0 | |
| 0 0 | 0x20190402 | Reset |

**SHA_DATE** Version control register. (R/W)

### Register 21.9. SHA_MODE_REG (0x0000)

| (reserved) | SHA_MODE | |
|---|---|---|
| 31 … 3 | 2 … 0 | |
| 0 0 0 0 0 0 0 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 |

**SHA_MODE** Defines the SHA algorithm. For details, please see Table 21.3-2. (R/W)

### Register 21.10. SHA_DMA_BLOCK_NUM_REG (0x000C)

| (reserved) | SHA_DMA_BLOCK_NUM |
|---|---|
| 31 … 6 | 5 … 0 |
| 0 0 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 |

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

### Register 21.11. SHA_H_*n*_REG (*n*: 0-7) (0x0040+4**n*)

| SHA_H_*n* |
|---|
| 31 … 0 |
| 0x000000 |
| Reset |

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

### Register 21.12. SHA_M_*n*_REG (*n*: 0-15) (0x0080+4**n*)

| SHA_M_*n* |
|---|
| 31 … 0 |
| 0x000000 |
| Reset |

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

# Chapter 22

# Digital Signature (DS)

## 22.1 Overview

A Digital Signature is used to verify the authenticity and integrity of a message using a cryptographic algorithm. This can be used to validate a device's identity to a server, or to check the integrity of a message.

The ESP32-C3 includes a Digital Signature (DS) module providing hardware acceleration of messages' signatures based on RSA. It uses pre-encrypted parameters to calculate a signature. The parameters are encrypted using HMAC as a key-derivation function. In turn, the HMAC uses eFuses as an input key. The whole process happens in hardware so that neither the decryption key for the RSA parameters nor the input key for the HMAC key derivation function can be seen by users while calculating the signature.

## 22.2 Features

- RSA digital signatures with key length up to 3072 bits
- Encrypted private key data, only decryptable by DS module
- SHA-256 digest to protect private key data against tampering by an attacker

## 22.3 Functional Description

### 22.3.1 Overview

The DS peripheral calculates RSA signature as $Z = X^Y \bmod M$ where $Z$ is the signature, $X$ is the input message, and $Y$ and $M$ are the RSA private key parameters.

Private key parameters are stored in flash as ciphertext. They are decrypted using a key ($DS_KEY$) which can only be calculated by the DS peripheral via the HMAC peripheral. The required inputs ($HMAC_KEY$) to generate the key are only stored in eFuse and can only be accessed by the HMAC peripheral. That is to say, the DS peripheral hardware can decrypt the private key, and the private key in plaintext is never accessed by the software. For more detailed information about eFuse and HMAC peripherals, please refer to Chapter 4 *eFuse Controller (EFUSE)* and 19 *HMAC Accelerator (HMAC)* peripheral.

The input message $X$ will be sent directly to the DS peripheral by the software each time a signature is needed. After the RSA signature operation, the signature $Z$ is read back by the software.

For better understanding, we define some symbols and functions here, which are only applicable to this chapter:

- $1^s$ A bit string consist of $s$ bits with the value of "1".

- $[x]_s$ A bit string of $s$ bits, in which $s$ should be an integer multiple of 8 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 such as $[Y]_{4096}$ or as a hexadecimal constant such as $[0x0C]_8$. If necessary, the value $[x]_t$ can be right-padded with $(s-t)$ number of 0 to reach $s$ bits in length, and finally get $[x]_s$. For example, $[0x05]_8 = 00000101$, $[0x05]_{16} = 0000010100000000$, $[0x0005]_{16} = 0000000000000101$, $[0x13]_8 = 00010011$, $[0x13]_{16} = 0001001100000000$, $[0x0013]_{16} = 0000000000010011$.
- $||$ A bit string concatenation operator for joining multiple bit strings into a longer bit string.

### 22.3.2 Private Key Operands

Private key operands $Y$ (private key exponent) and $M$ (key modulus) are generated by you. They have a particular RSA key length (up to 3072 bits). Two additional private key operands are needed: $\overline{r}$ and $M'$. These two operands are derived from $Y$ and $M$.

Operands $Y$, $M$, $\overline{r}$ and $M'$ are encrypted by you along with an authentication digest and stored as a single ciphertext $C$. $C$ is input to the DS peripheral in this encrypted format, decrypted by the hardware, and then used for RSA signature calculation. Detailed description of how to generate $C$ is provided in Section 22.3.3.

The DS peripheral supports RSA signature calculation $Z = X^Y \bmod M$, in which the length of operands should be $N = 32 \times x$ where $x \in \{1, 2, 3, \ldots, 96\}$. The bit lengths of arguments $Z$, $X$, $Y$, $M$ and $\overline{r}$ should be an arbitrary value in $N$, and all of them in a calculation must be of the same length, while the bit length of $M'$ should always be 32. For more detailed information about RSA calculation, please refer to Section 20.3.1 *Large Number Modular Exponentiation* in Chapter 20 *RSA Accelerator (RSA)*.

### 22.3.3 Software Prerequisites

If users want to use the DS module for digital signature, the software and hardware must work closely to implement this successfully, and the software needs to do a series of preparations, as shown in Figure 22.3-1. The left side lists preparations required by the software before the hardware starts RSA signature calculation, while the right side lists the hardware workflow during the entire calculation procedure.

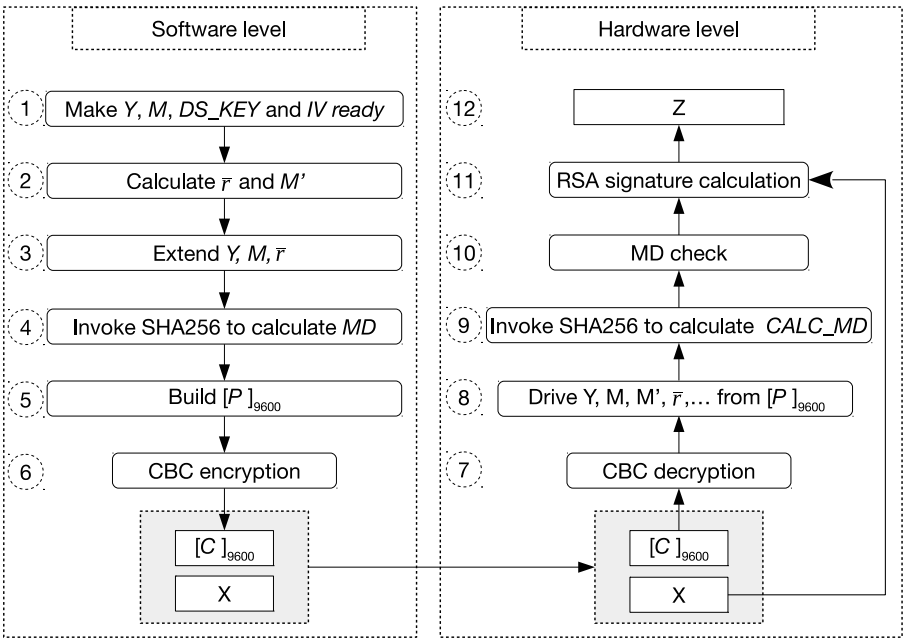



Figure 22.3-1. Software Preparations and Hardware Working Process

**Note:**

1. The software preparation (left side in the Figure 22.3-1) is a one-time operation before any signature is calculated, while the hardware calculation (right side in the Figure 22.3-1) repeats for every signature calculation.

You need to follow the steps shown in the left part of Figure 22.3-1 to calculate $C$. Detailed instructions are as follows:

- **Step 1**: Prepare operands $Y$ and $M$ whose lengths should meet the requirements in Section 22.3.2. Define $[L]_{32} = \frac{N}{32} - 1$ (i.e., for RSA 3072, $[L]_{32}$ == [0x60-1]$_{32}$). Prepare $[HMAC_KEY]_{256}$ and calculate $[DS_KEY]_{256}$ based on $DS_KEY$ = HMAC-SHA256 ($[HMAC_KEY]_{256}$, $1^{256}$). 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 18 *AES Accelerator (AES)*.
- **Step 2**: Calculate $\overline{r}$ and $M'$ based on $M$.
- **Step 3**: Extend $Y$, $M$ and $\overline{r}$, in order to get $[Y]_{3072}$, $[M]_{3072}$ and $[\overline{r}]_{3072}$, respectively. This step is only required for $Y$, $M$ and $\overline{r}$ whose length are less than 3072 bits, since their largest length are 3072 bits.
- **Step 4**: Calculate MD authentication code using the SHA-256: $[MD]_{256}$ = SHA256 ($[Y]_{3072}||[M]_{3072}||[\overline{r}]_{3072}||[M']_{32}||[L]_{32}||[IV]_{128}$)
- **Step 5**: Build $[P]_{9600}$ = ( $[Y]_{3072}||[M]_{3072}||[\overline{r}]_{3072}||[Box]_{384}$), where $[Box]_{384}$ = ( $[MD]_{256}||[M']_{32}||[L]_{32}||[\beta]_{64}$) and $[\beta]_{64}$ is a PKCS#7 padding value, i.e., a [0x0808080808080808]$_{64}$ string composed of 8 bytes (0x80). The purpose of $[\beta]_{64}$ is to make the bit length of $P$ a multiple of 128.
- **Step 6**: Calculate $C$ = $[C]_{9600}$ = AES-CBC-ENC ($[P]_{9600}$, $[DS_KEY]_{256}$, $[IV]_{128}$), where C is the ciphertext with a length of 1200 bytes. $C$ can also be calculated as $C$ = $[C]_{9600}$ = ($[\widehat{Y}]_{3072}||[\widehat{M}]_{3072}||[\widehat{\overline{r}}]_{3072}||[\widehat{Box}]_{384}$), where $[\widehat{Y}]_{3072}$, $[\widehat{M}]_{3072}$, $[\widehat{\overline{r}}]_{3072}$, $[\widehat{Box}]_{384}$ are the four sub-parameters of $C$, and correspond to the ciphertext of $[Y]_{3072}$, $[M]_{3072}$, $[\overline{r}]_{3072}$, $[Box]_{384}$ respectively.

### 22.3.4 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 can be divided into the following three stages:

1. **Decryption: Step 7 and 8 in Figure 22.3-1**

   The decryption process is the inverse of Step 6 in figure 22.3-1. The DS module will call AES accelerator to decrypt $C$ in CBC block mode and get the resulted plaintext. The decryption process can be represented by $P$ = AES-CBC-DEC ($C$, $DS_KEY$, $IV$), where $IV$ (i.e., $[IV]_{128}$) is defined by you. $[DS_KEY]_{256}$ is provided by HMAC module, derived from $HMAC_KEY$ stored in eFuse. $[DS_KEY]_{256}$, as well as $[HMAC_KEY]_{256}$ are not readable by users. For more information, please refer to Chapter 19*HMAC Accelerator (HMAC)*.

   With P, the DS module can derive $[Y]_{3072}$, $[M]_{3072}$, $[\overline{r}]_{3072}$, $[M']_{32}$, $[L]_{32}$, MD authentication code, and the padding value $[\beta]_{64}$. This process is the inverse of Step 5.

2. **Check: Step 9 and 10 in Figure 22.3-1**

   The DS module will perform two checks: MD check and padding check. Padding check is not shown in Figure 22.3-1, as it happens at the same time with MD check.

   - MD check: The DS module calls SHA-256 to calculate the hash value $[CALC_MD]_{256}$ (i.e., step 4). Then, $[CALC_MD]_{256}$ is compared against the MD authentication code $[MD]_{256}$ from step 4. Only when the two match does the MD check pass.
   - Padding check: The DS module checks if $[\beta]_{64}$ complies with the aforementioned PKCS#7 format. Only when $[\beta]_{64}$ complies with the format does the padding check pass.

   The DS module will only perform subsequent operations if MD check passes. If padding check fails, a warning message is generated, but it does not affect the subsequent operations.

3. **Calculation: Step 11 and 12 in Figure 22.3-1**

   The DS module treats $X$ (input by you) and $Y$, $M$, $\overline{r}$ (compiled) as big numbers. With $M'$, all operands to perform $X^Y$ mod $M$ are in place. The operand length is defined by $L$ only. The DS module will get the signed result $Z$ by calling RSA to perform $Z = X^Y$ mod $M$.

### 22.3.5 DS Operation at the Software Level

The software steps below 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 22.3.4.

We assume that the software has called the HMAC peripheral and HMAC on the hardware has calculated $DS_KEY$ based on $HMAC_KEY$.

1. **Prerequisites**: Prepare operands $C$, $X$, $IV$ according to Section 22.3.3.
2. **Activate the DS peripheral**: Write 1 to DS_SET_START_REG.
3. **Check if $DS_KEY$ is ready**: Poll DS_QUERY_BUSY_REG until the software reads 0.

If the software does not read 0 in DS_QUERY_BUSY_REG after approximately 1 ms, it indicates a problem with HMAC initialization. In such a case, the software can read register DS_QUERY_KEY_WRONG_REG to get more information:

- If the software reads 0 in DS_QUERY_KEY_WRONG_REG, it indicates that the HMAC peripheral has not been called.
- If the software reads any value from 1 to 15 in DS_QUERY_KEY_WRONG_REG, it indicates that HMAC was called, but the DS module did not successfully get the $DS_KEY$ value from the HMAC peripheral. This may indicate that the HMAC operation has been interrupted due to a software concurrency problem.

4. **Configure register**: Write $IV$ block to register DS_IV_*m*_REG (*m*: 0 ~ 3). For more information on the $IV$ block, please refer to Chapter 18 *AES Accelerator (AES)*.
5. **Write $X$ to memory block DS_X_MEM**: Write $X_i$ ($i \in \{0, 1, \ldots, n-1\}$), where $n = \frac{N}{32}$, to memory block DS_X_MEM whose capacity is 96 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 22.3.2), are ignored.
6. **Write $C$ to corresponding memory blocks**: Write the four sub-parameters of $C$ to corresponding memory blocks:
   - Write $\widehat{Y}_i$ ($i \in \{0, 1, \ldots, 95\}$) to DS_Y_MEM.
   - Write $\widehat{M}_i$ ($i \in \{0, 1, \ldots, 95\}$) to DS_M_MEM.
   - Write $\widehat{r}_i$ ($i \in \{0, 1, \ldots, 95\}$) to DS_RB_MEM.
   - write $\widehat{Box}_i$ ($i \in \{0, 1, \ldots, 11\}$) to DS_BOX_MEM.

   The capacity of DS_Y_MEM, DS_M_MEM, and DS_RB_MEM is 96 words, whereas the capacity of DS_BOX_MEM is only 12 words. Each word can store one base-$b$ digit. The memory blocks use the little endian format for storage, i.e., the least significant digit of the operand is in the lowest address.
7. **Start DS operation**: Write 1 to register DS_SET_ME_REG.
8. **Wait for the operation to be completed**: Poll register DS_QUERY_BUSY_REG until the software reads 0.
9. **Query check result**: Read register DS_QUERY_CHECK_REG and conduct subsequent operations as illustrated below based on the return value:
   - If the value is 0, it indicates that both padding check and MD check pass. You 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 will resume directly from Step 11.
   - If the value is 2, it indicates that the padding check fails but MD check passes. You can continue to get the signed result $Z$. But please note that the data does not comply with the aforementioned PKCS#7 padding format, which may not be what you want.
   - If the value is 3, it indicates that both padding check and MD check fail. In this case, some fatal errors have occurred and the signed result $Z$ is invalid. The operation will resume directly from Step 11.

10. **Read the signed result**: Read the signed result $Z_i$ ($i \in \{0, 1, \ldots, n-1\}$), where $n = \frac{N}{32}$, from memory block DS_Z_MEM. The memory block stores $Z$ in little-endian byte order.
11. **Exit the operation**: Write 1 to DS_SET_FINISH_REG, and then poll DS_QUERY_BUSY_REG until the software reads 0.

After the operation, all the input/output registers and memory blocks are cleared.

## 22.4 Memory Summary

The addresses in this section are relative to the Digital Signature base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

| Name | Description | Size (byte) | Starting Address | Ending Address | Access |
|---|---|---|---|---|---|
| DS_Y_MEM | Memory block Y | 384 | 0x0000 | 0x017F | WO |
| DS_M_MEM | Memory block M | 384 | 0x0200 | 0x037F | WO |
| DS_RB_MEM | Memory block $\overline{r}$ | 384 | 0x0400 | 0x057F | WO |
| DS_BOX_MEM | Memory block Box | 48 | 0x0600 | 0x062F | WO |
| DS_X_MEM | Memory block X | 384 | 0x0800 | 0x097F | WO |
| DS_Z_MEM | Memory block Z | 384 | 0x0A00 | 0x0B7F | RO |

## 22.5 Register Summary

The addresses in this section are relative to Digital Signature base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Registers** | | | |
| DS_IV_0_REG | $IV$ block data | 0x0630 | WO |
| DS_IV_1_REG | $IV$ block data | 0x0634 | WO |
| DS_IV_2_REG | $IV$ block data | 0x0638 | WO |
| DS_IV_3_REG | $IV$ block data | 0x063C | WO |
| **Status/Control Registers** | | | |
| DS_SET_START_REG | Activates the DS module | 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 module | 0x0E0C | RO |
| DS_QUERY_KEY_WRONG_REG | Checks the reason why $DS_KEY$ is not ready | 0x0E10 | RO |
| DS_QUERY_CHECK_REG | Queries DS check result | 0x0814 | RO |
| **Version control register** | | | |
| DS_DATE_REG | Version control register | 0x0820 | W/R |

## 22.6 Registers

The addresses in this section are relative to Digital Signature base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 22.1. DS_IV_*m*_REG (*m*: 0-3) (0x0630+4**m*)**

| DS_IV_*m*_REG (*m*: 0-3) | |
|---|---|
| 31 … 0 | |
| 0x000000000 | Reset |

**DS_IV_*m*_REG (*m*: 0-3)** $IV$ block data. (WO)

**Register 22.2. DS_SET_START_REG (0x0E00)**

| (reserved) | DS_SET_START | |
|---|---|---|
| 31 … 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 | Reset |

**DS_SET_START** Write 1 to this register to activate the DS peripheral. (WO)

**Register 22.3. DS_SET_ME_REG (0x0E04)**

| (reserved) | DS_SET_ME | |
|---|---|---|
| 31 … 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 | Reset |

**DS_SET_ME** Write 1 to this register to start DS operation. (WO)

### Register 22.4. DS_SET_FINISH_REG (0x0E08)

| (reserved) | DS_SET_FINISH | |
|---|---|---|
| 31 … 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 | Reset |

**DS_SET_FINISH** Write 1 to this register to end DS operation. (WO)

### Register 22.5. DS_QUERY_BUSY_REG (0x0E0C)

| (reserved) | DS_QUERY_BUSY | |
|---|---|---|
| 31 … 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 | Reset |

**DS_QUERY_BUSY** 1: The DS module is busy; 0: The DS module is idle. (RO)

### Register 22.6. DS_QUERY_KEY_WRONG_REG (0x0E10)

| (reserved) | DS_QUERY_KEY_WRONG | |
|---|---|---|
| 31 … 4 | 3 … 0 | |
| 0 0 0 0 0 0 0 0 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 |

**DS_QUERY_KEY_WRONG** 1-15: HMAC was activated, but the DS peripheral did not successfully receive the $DS_KEY$ from the HMAC peripheral. (The biggest value is 15); 0: HMAC is not called. (RO)

### Register 22.7. DS_QUERY_CHECK_REG (0x0E14)

| (reserved) | | DS_PADDING_BAD | DS_MD_ERROR | |
|---|---|---|---|---|
| 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 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 22.8. DS_DATE_REG (0x0E20)

| (reserved) | DS_DATE | |
|---|---|---|
| 31 30 | 29 0 | |
| 0 0 | 0x20200618 | Reset |

**DS_DATE** Version control register. (R/W)

# Chapter 23

# External Memory Encryption and Decryption (XTS_AES)

## 23.1 Overview

The ESP32-C3 integrates an External Memory Encryption and Decryption module that complies with the XTS_AES standard algorithm specified in IEEE Std 1619-2007, providing security for users' application code and data stored in the external memory (flash). Users can store proprietary firmware and sensitive data (e.g., credentials for gaining access to a private network) to the external flash.

## 23.2 Features

- General XTS_AES algorithm, compliant with IEEE Std 1619-2007
- Software-based manual encryption
- High-speed auto decryption, without software's participation
- Encryption and decryption functions jointly determined by registers configuration, eFuse parameters, and boot mode

## 23.3 Module Structure

The External Memory Encryption and Decryption module consists of two blocks, namely the Manual Encryption block and Auto Decryption block. The module architecture is shown in Figure 23.3-1.

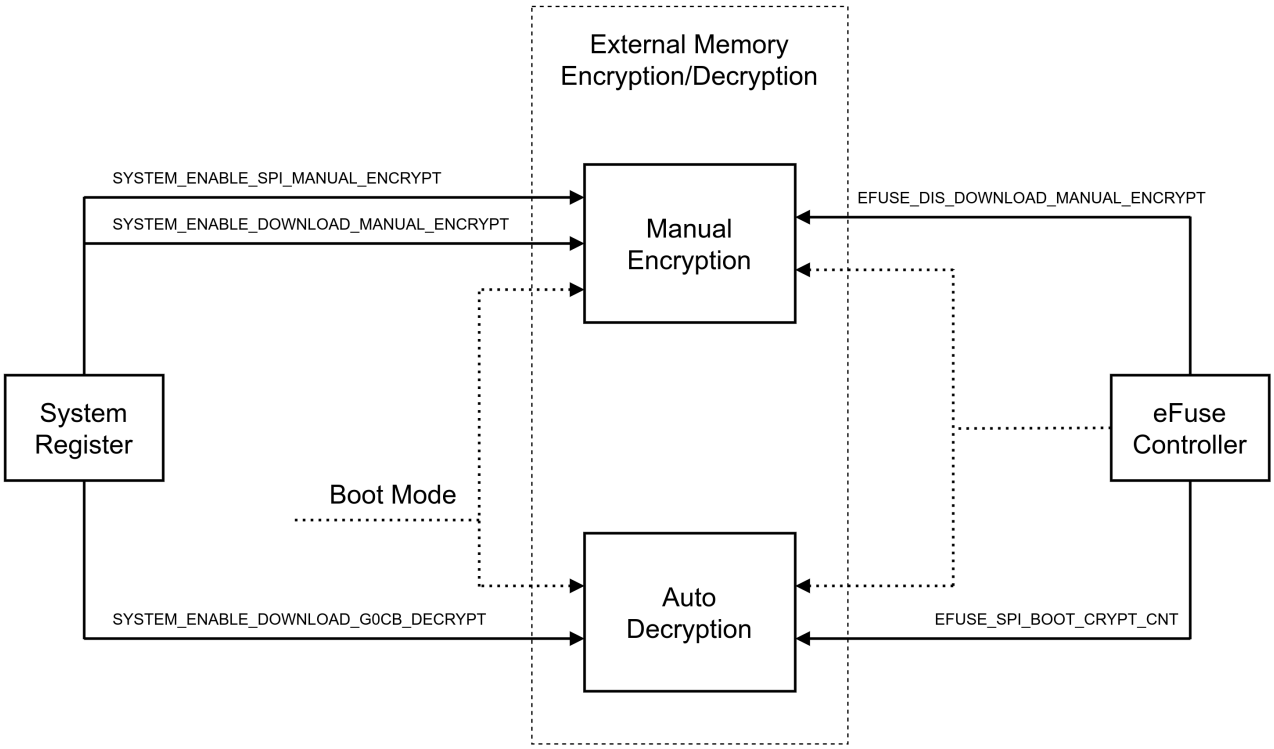



Figure 23.3-1. Architecture of the External Memory Encryption and Decryption

The Manual Encryption block can encrypt instructions/data which will then be written to the external flash as ciphertext via SPI1.

In the System Registers (SYSREG) peripheral (see 16 *System Registers (SYSREG)*), the following four bits in register SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG are relevant to the external memory encryption and decryption:

- SYSTEM_ENABLE_DOWNLOAD_MANUAL_ENCRYPT
- SYSTEM_ENABLE_DOWNLOAD_GOCB_DECRYPT
- SYSTEM_ENABLE_DOWNLOAD_DB_ENCRYPT
- SYSTEM_ENABLE_SPI_MANUAL_ENCRYPT

The XTS_AES module also fetches two parameters from the peripheral eFuse Controller, which are: EFUSE_DIS_DOWNLOAD_MANUAL_ENCRYPT and EFUSE_SPI_BOOT_CRYPT_CNT. For detailed information, please see 4 *eFuse Controller (EFUSE)*.

## 23.4 Functional Description

### 23.4.1 XTS Algorithm

The manual encryption and auto decryption use the XTS algorithm. During implementation, the XTS algorithm is characterized by a "data unit" of 1024 bits, defined in the Section *XTS-AES encryption procedure* of *XTS-AES Tweakable Block Cipher* Standard. For more information about XTS-AES algorithm, please refer to IEEE Std 1619-2007.

### 23.4.2 Key

The Manual Encryption block and Auto Decryption block share the same $Key$ when implementing XTS algorithm. The $Key$ is provided by the eFuse hardware and cannot be accessed by users.

The $Key$ is 256-bit long. The value of the $Key$ is determined by the content in one eFuse block from BLOCK4 ~ BLOCK9. For easier description, we define:

- Block$_A$: the block whose key purpose is EFUSE_KEY_PURPOSE_XTS_AES_128_KEY (please refer to Table 4.3-2 *Secure Key Purpose Values*). The 256-bit $Key_A$ is stored in it.

There are two possibilities of how the $Key$ is generated depending on whether Block$_A$ exists or not, as shown in Table 23.4-1. In each case, the $Key$ can be uniquely determined by Block$_A$.

**Table 23.4-1. $Key$ generated based on $Key_A$**

| Block$_A$ | $Key$ | $Key$ Length (bit) |
|---|---|---|
| Yes | $Key_A$ | 256 |
| No | $0^{256}$ | 256 |

**Notes:**
"YES" indicates that the block exists; "NO" indicates that the block does not exist; "$0^{256}$" indicates a bit string that consists of 256-bit zeros. Note that using $0^{256}$ as Key is not secure. We strongly recommend to configure a valid key.

For more information of key purposes, please refer to Table 4.3-2 *Secure Key Purpose Values* in Chapter 4 *eFuse Controller (EFUSE)*.

### 23.4.3 Target Memory Space

The target memory space refers to a continuous address space in the external memory (flash) where the ciphertext is stored. The target memory space can be uniquely determined by two relevant parameters: size and base address, whose definitions are listed below.

- Size: the $size$ of the target memory space, indicating the number of bytes encrypted in one encryption operation, which supports 16 or 32 bytes.
- Base address: the $base_addr$ of the target memory space. It is a 24-bit physical address, with range of 0x0000_0000 ~ 0x00FF_FFFF. It should be aligned to $size$, i.e., $base_addr\%size == 0$.

For example, if there are 16 bytes of instruction data need to be encrypted and written to address 0x130 ~ 0x13F in the external flash, then the target space is 0x130 ~ 0x13F, size is 16 (bytes), and base address is 0x130.

The encryption of any length (must be multiples of 16 bytes) of plaintext instruction/data can be completed separately in multiple operations, and each operation has its individual target memory space and the relevant parameters.

For Auto Decryption blocks, these parameters are automatically determined by hardware. For Manual Encryption blocks, these parameters should be configured by users.

**Note:**
The "tweak" defined in Section *Data units and tweaks* of IEEE Std 1619-2007 is a 128-bit non-negative integer

($tweak$), which can be generated according to $tweak$ = ($base_addr$ & 0x00FFFF80). The lowest 7 bits and the highest 97 bits in $tweak$ are always zero.

### 23.4.4 Data Writing

For Auto Decryption blocks, data writing is automatically applied in hardware. For Manual Encryption blocks, data writing should be applied by users. The Manual Encryption block has a register block which consists of 8 registers, i.e., XTS_AES_PLAIN_$n$_REG ($n$: 0 ~ 7), that are dedicated to data writing and can store up to 256 bits of plaintext at a time.

Actually, the Manual Encryption block does not care where the plaintext comes from, but only where the ciphertext will be stored. Because of the strict correspondence between plaintext and ciphertext, in order to better describe how the plaintext is stored in the register block, we assume 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. Please note that the plaintext can come from everywhere in actual use, but users should understand how the plaintext is stored in the register block.

**How mapping between target memory space and registers works:**

Assume a word in the target memory space is stored in $address$, define $offset = address\%32$, $n$ = $\frac{offset}{4}$, then the word will be stored in register XTS_AES_PLAIN_$n$_REG.

The mapping between $offset$ and registers is shown in Table 23.4-2.

**Table 23.4-2. Mapping Between Offsets and Registers**

| $offset$ | Register | $offset$ | Register |
|---|---|---|---|
| 0x00 | XTS_AES_PLAIN_0_REG | 0x10 | XTS_AES_PLAIN_4_REG |
| 0x04 | XTS_AES_PLAIN_1_REG | 0x14 | XTS_AES_PLAIN_5_REG |
| 0x08 | XTS_AES_PLAIN_2_REG | 0x18 | XTS_AES_PLAIN_6_REG |
| 0x0C | XTS_AES_PLAIN_3_REG | 0x1C | XTS_AES_PLAIN_7_REG |

### 23.4.5 Manual Encryption Block

The Manual Encryption block is a peripheral module. It is equipped with registers and can be accessed by the CPU directly. Registers embedded in this block, the System Registers (SYSREG) peripheral, eFuse parameters, and boot mode jointly configure and use this module. Please note that the Manual Encryption block can only encrypt for storage in external flash.

**The Manual Encryption block is operational only under certain conditions.** 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 can be enabled. 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 can be enabled. Otherwise, it is not operational.

**Note:**

- Even though the CPU can skip cache and get the encrypted instruction/data directly by reading the external memory, users can by no means access $Key$.

### 23.4.6 Auto Decryption Block

The Auto Decryption block is not a conventional peripheral, so it does not have any registers and cannot be accessed by the CPU directly. The System Registers (SYSREG) peripheral, eFuse parameters, and boot mode jointly configure and use this block.

**The Auto Decryption block is operational only under certain conditions.** The operating conditions are:

- In SPI Boot mode

  If the first bit or the third bit in parameter SPI_BOOT_CRYPT_CNT (3 bits) is set to 1, then the Auto Decryption block can be enabled. Otherwise, it is not operational.

- In Download Boot mode

  If bit SYSTEM_ENABLE_DOWNLOAD_GOCB_DECRYPT in register SYSTEM_EXTERNAL_DEVICE_ENCRYPT_DECRYPT_CONTROL_REG is 1, the Auto Decryption block can be enabled. Otherwise, it is not operational.

**Note:**

- When the Auto Decryption block is enabled, it will automatically decrypt the ciphertext if the CPU reads instructions/data from the external memory via cache to retrieve the instructions/data. The entire decryption process does not need software participation and is transparent to the cache. Users can by no means obtain the decryption $Key$ during the process.
- When the Auto Decryption block is disabled, it does not have any effect on the contents stored in the external memory, no matter if they are encrypted or not. Therefore, what the CPU reads via cache is the original information stored in the external memory.

## 23.5 Software Process

When the Manual Encryption block operates, software needs to be involved in the process. The steps are as follows:

1. Configure XTS_AES:

   - Set register XTS_AES_PHYSICAL_ADDRESS_REG to $base_addr$.
   - Set register XTS_AES_LINESIZE_REG to $\frac{size}{32}$.

   For definitions of $base_addr$ and $size$, please refer to Section 23.4.3.

2. Write plaintext data to the registers block XTS_AES_PLAIN_*n*_REG (*n*: 0-7). For detailed information, please refer to Section 23.4.4.
   Please write data to registers according to your actual needs, and the unused ones could be set to arbitrary values.
3. Wait for Manual Encryption block to be idle. Poll register XTS_AES_STATE_REG until it reads 0 that indicates the Manual Encryption block is idle.
4. Trigger manual encryption by writing 1 to register XTS_AES_TRIGGER_REG.
5. Wait for the encryption process completion. Poll register XTS_AES_STATE_REG until it reads 2.
   *Step 1 to 5 are the steps of encrypting plaintext instructions with the Manual Encryption block using the* $Key$.
6. Write 1 to register XTS_AES_RELEASE_REG to grant SPI1 the access to the encrypted ciphertext. After this, the value of register XTS_AES_STATE_REG will become 3.
7. Call SPI1 to write the ciphertext in the external flash (see Chapter *27 SPI Controller (SPI)*).
8. Write 1 to register
   XTS_AES_DESTROY_REG to destroy the ciphertext. After this, the value of register XTS_AES_STATE_REG will become 0.

Repeat above steps according to the amount of plaintext instructions/data that need to be encrypted.

## 23.6 Register Summary

The addresses in this section are relative to External Memory Encryption and Decryption base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Plaintext Register Heap** | | | |
| XTS_AES_PLAIN_0_REG | Plaintext register 0 | 0x0000 | R/W |
| XTS_AES_PLAIN_1_REG | Plaintext register 1 | 0x0004 | R/W |
| XTS_AES_PLAIN_2_REG | Plaintext register 2 | 0x0008 | R/W |
| XTS_AES_PLAIN_3_REG | Plaintext register 3 | 0x000C | R/W |
| XTS_AES_PLAIN_4_REG | Plaintext register 4 | 0x0010 | R/W |
| XTS_AES_PLAIN_5_REG | Plaintext register 5 | 0x0014 | R/W |
| XTS_AES_PLAIN_6_REG | Plaintext register 6 | 0x0018 | R/W |
| XTS_AES_PLAIN_7_REG | Plaintext register 7 | 0x001C | R/W |
| **Configuration Registers** | | | |
| XTS_AES_LINESIZE_REG | Configures the size of target memory space | 0x0040 | R/W |
| XTS_AES_DESTINATION_REG | Configures the type of the external memory | 0x0044 | R/W |
| XTS_AES_PHYSICAL_ADDRESS_REG | Physical address | 0x0048 | R/W |
| **Control/Status Registers** | | | |
| XTS_AES_TRIGGER_REG | Activates AES algorithm | 0x004C | WO |
| XTS_AES_RELEASE_REG | Release control | 0x0050 | WO |
| XTS_AES_DESTROY_REG | Destroys control | 0x0054 | WO |
| XTS_AES_STATE_REG | Status register | 0x0058 | RO |
| **Version Register** | | | |
| XTS_AES_DATE_REG | Version control register | 0x005C | RO |

## 23.7 Registers

The addresses in this section are relative to External Memory Encryption and Decryption base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 23.1. XTS_AES_PLAIN_*n*_REG (*n*: 0-15) (0x0000+4**n*)**

| XTS_AES_PLAIN_*n* | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**XTS_AES_PLAIN_*n*** Stores *n*th 32-bit piece of plain text. (R/W)

**Register 23.2. XTS_AES_LINESIZE_REG (0x0040)**

| (reserved) | XTS_AES_LINESIZE | |
|---|---|---|
| 31 … 1 | 0 | |
| 0x00000000 | 0 | Reset |

**XTS_AES_LINESIZE** Configures the data size of one encryption operation.

- 0: 16 bytes;
- 1: 32 bytes. (R/W)

**Register 23.3. XTS_AES_DESTINATION_REG (0x0044)**

| (reserved) | XTS_AES_DESTINATION | |
|---|---|---|
| 31 … 1 | 0 | |
| 0x00000000 | 0 | Reset |

**XTS_AES_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 23.4. XTS_AES_PHYSICAL_ADDRESS_REG (0x0048)

| (reserved) | XTS_AES_PHYSICAL_ADDRESS | |
|---|---|---|
| 31 30 | 29 0 | |
| 0x0 | 0x00000000 | Reset |

**XTS_AES_PHYSICAL_ADDRESS** Physical address. (Note that its value should be within the range between 0x0000_0000 and 0x00FF_FFFF). (R/W)

### Register 23.5. XTS_AES_TRIGGER_REG (0x004C)

| (reserved) | XTS_AES_TRIGGER | |
|---|---|---|
| 31 1 | 0 | |
| 0x00000000 | X | Reset |

**XTS_AES_TRIGGER** Write 1 to enable manual encryption. (WO)

### Register 23.6. XTS_AES_RELEASE_REG (0x0050)

| (reserved) | XTS_AES_RELEASE | |
|---|---|---|
| 31 1 | 0 | |
| 0x00000000 | X | Reset |

**XTS_AES_RELEASE** Write 1 to grant SPI1 access to the encrypted result. (WO)

### Register 23.7. XTS_AES_DESTROY_REG (0x0054)

| (reserved) | XTS_AES_DESTROY | |
|---|---|---|
| 31 … 1 | 0 | |
| 0x00000000 | X | Reset |

**XTS_AES_DESTROY** Write 1 to destroy encrypted result. (WO)

### Register 23.8. XTS_AES_STATE_REG (0x0058)

| (reserved) | XTS_AES_STATE | |
|---|---|---|
| 31 … 2 | 1 … 0 | |
| 0x00000000 | 0x0 | Reset |

**XTS_AES_STATE** Indicates the status of the Manual Encryption block.

- 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. (RO)

### Register 23.9. XTS_AES_DATE_REG (0x005C)

| (reserved) | XTS_AES_DATE | |
|---|---|---|
| 31 … 30 | 29 … 0 | |
| 0 … 0 | 0x20200111 | Reset |

**XTS_AES_DATE** Version control register. (R/W)

# Chapter 24

# Random Number Generator (RNG)

## 24.1 Introduction

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

## 24.2 Features

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

## 24.3 Functional Description

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

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

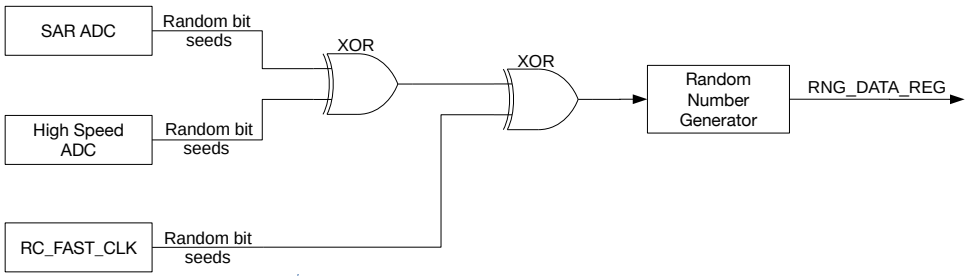



**Figure 24.3-1. Noise Source**

When there is noise coming from the SAR ADC, the random number generator is fed with a 2-bit entropy in one clock cycle of RC_FAST_CLK (20 MHz), which is generated from an internal RC oscillator (see Chapter 6

*Reset and Clock* for details). Thus, it is advisable to read the RNG_DATA_REG register at a maximum rate of 1 MHz to obtain the maximum entropy.

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

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

## 24.4 Programming Procedure

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

- SAR ADC can be enabled by using the DIG ADC controller. For details, please refer to Chapter 34 *On-Chip Sensor and Analog Signal Processing*.
- High-speed ADC is enabled automatically when the Wi-Fi or Bluetooth modules is enabled.
- RC_FAST_CLK is enabled by setting the RTC_CNTL_DIG_FOSC_EN bit in the RTC_CNTL_CLK_CONF_REG register.

**Note:**

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

When using the random number generator, read the RNG_DATA_REG register multiple times until sufficient random numbers have been generated. Ensure the rate at which the register is read does not exceed the frequencies described in section 24.3 above.

## 24.5 Register Summary

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| RNG_DATA_REG | Random number data | 0x6002_60B0 | RO |

## 24.6 Register

**Register 24.1. RNG_DATA_REG (0x6002_60B0)**

| RNG_DATA | |
|---|---|
| 31 … 0 | |
| 0x00000000 | Reset |

**RNG_DATA** Random number source. (RO)

# Chapter 25

# Clock Glitch Detection

## 25.1 Overview

The Clock Glitch Detection module on ESP32-C3 detects glitches in external crystal XTAL_CLK signals, and generates a system reset signal when detecting glitches to reset the whole digital circuit including RTC. By doing so, it prevents attackers from injecting glitches on external crystal XTAL_CLK clock to compromise ESP32-C3 and thus strengthens chip security.

## 25.2 Functional Description

### 25.2.1 Clock Glitch Detection

The Clock Glitch Detection module on ESP32-C3 monitors input clock signals from XTAL_CLK. If it detects a glitch, namely a clock pulse (a or b in the figure below) with a width shorter than 3 ns, input clock signals from XTAL_CLK are blocked.

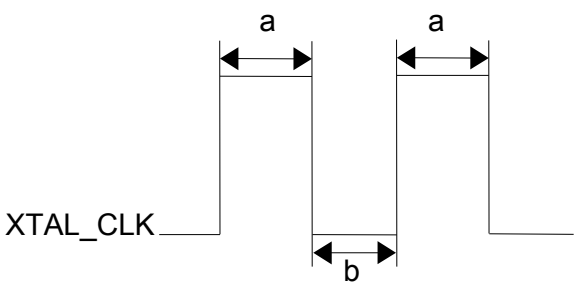



Figure 25.2-1. XTAL_CLK Pulse Width

### 25.2.2 Reset

Once detecting a glitch on XTAL_CLK that affects the circuit's normal operation, the Clock Glitch Detection module triggers a system reset if RTC_CNTL_GLITCH_RST_EN bit is enabled. By default, this bit is set to enable a reset.

# Part V

# Connectivity Interface

This part addresses the connectivity aspects of the system, describing components related to various communication interfaces like I2C, I2S, SPI, UART, USB, and more. The part also covers interfaces to generate signals used in remote control, motor control, LED control, etc.

# Chapter 26

# UART Controller (UART)

## 26.1 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-C3 has two UART controllers compatible with various UART devices. They support Infrared Data Association (IrDA) and RS485 transmission.

Each of the two UART controllers 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.

A UART is a character-oriented data link for asynchronous communication between devices. Such communication does not add clock signals to the 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(s) and parity bit.

A UART data frame usually begins with one start bit, followed by data bits, one parity bit (optional) and one or more stop bits. UART controllers on ESP32-C3 support various lengths of data bits and stop bits. These controllers also support software and hardware flow control as well as GDMA for seamless high-speed data transfer. This allows developers to use multiple UART ports at minimal software cost.

## 26.2 Features

Each UART controller has the following features:

- Three clock sources that can be divided
- Programmable baud rate
- 512 x 8-bit RAM shared by TX FIFOs and RX FIFOs of the two UART controllers
- Full-duplex asynchronous communication
- Automatic baud rate detection of input signals
- Data bits ranging from 5 to 8
- Stop bits whose length can be 1, 1.5 or 2 bits
- Parity bit
- Special character AT_CMD detection
- RS485 protocol

- IrDA protocol
- High-speed data communication using GDMA
- UART as wake-up source
- Software and hardware flow control

## 26.3 UART Structure

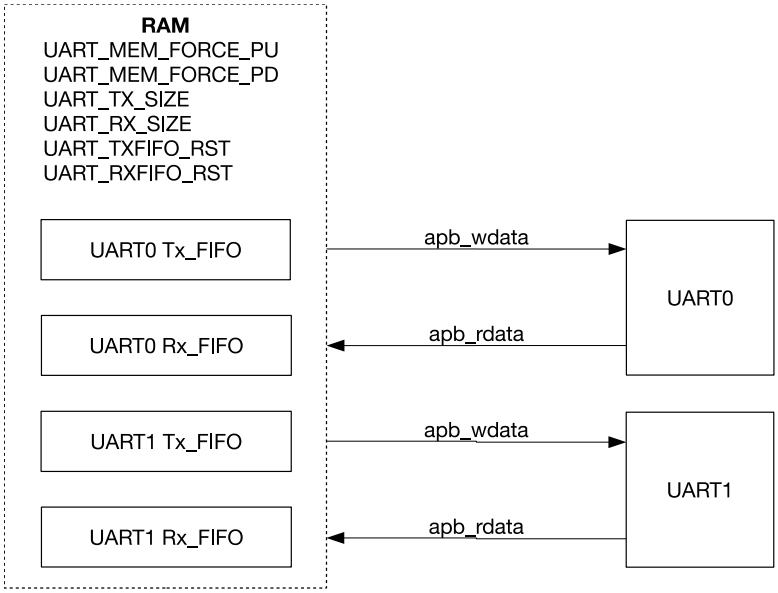



Figure 26.3-1. UART Architecture Overview

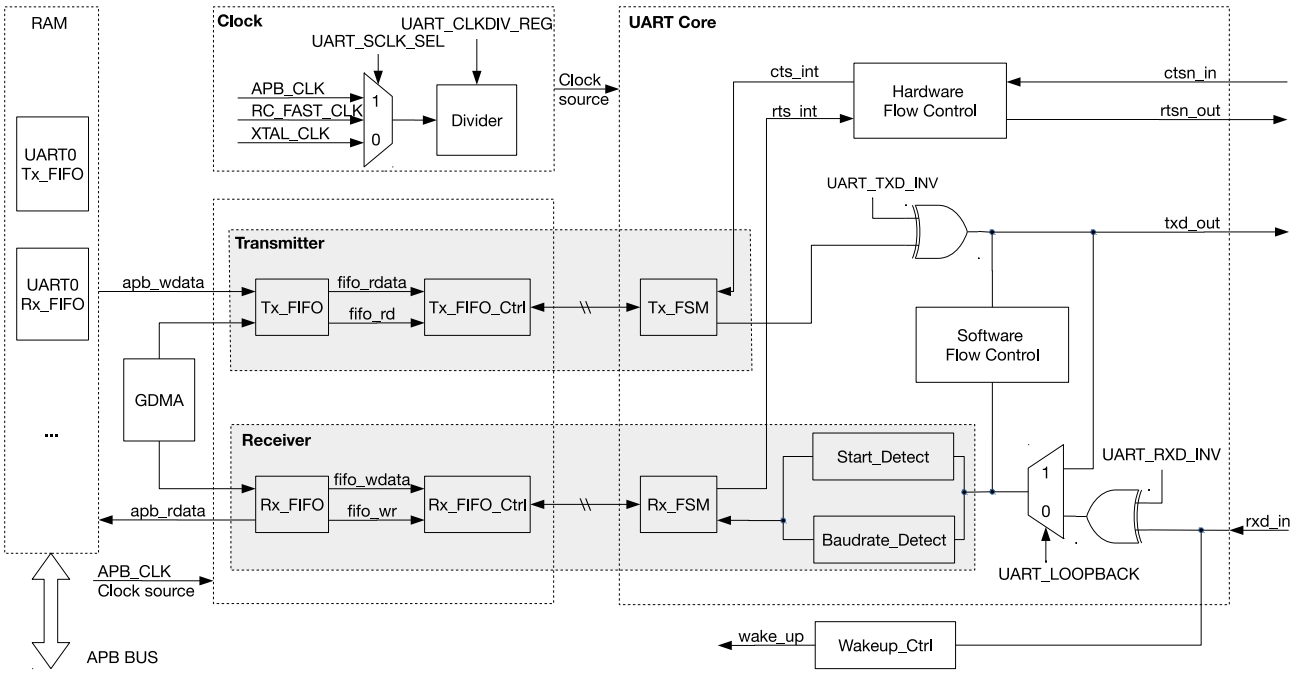



Figure 26.3-2. UART Structure

Figure 26.3-2 shows the basic structure of a UART controller. A UART controller works in two clock domains, namely APB_CLK domain and Core Clock domain (the UART Core's clock domain). The UART Core has three clock sources: a 80 MHz APB_CLK, RC_FAST_CLK and external crystal clock XTAL_CLK (for details, please refer to Chapter 6 *Reset and Clock*), which are selected by configuring UART_SCLK_SEL. The selected clock source is divided by a divider to generate clock signals that drive the UART Core. 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 GDMA. Tx_FIFO_Ctrl controls writing and reading Tx_FIFO. When Tx_FIFO is not empty, Tx_FSM reads data bits in the data frame via Tx_FIFO_Ctrl, and converts them into a bitstream. The levels of output signal txd_out can be inverted by configuring the UART_TXD_INV field.

The receiver contains a RX FIFO, which buffers data to be processed. The levels of input signal rxd_in can be inverted by configuring UART_RXD_INV field. 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. Software can read data from Rx_FIFO via the APB bus, or receive data using GDMA.

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 Light-sleep mode (see Chapter 9 *Low-power Management* for more details), Wakeup_Ctrl counts up rising edges of rxd_in. When the number is equal to or greater than (UART_ACTIVE_THRESHOLD + 3), a wake_up signal is generated and sent to RTC, which then wakes up the ESP32-C3 chip.

## 26.4 Functional Description

### 26.4.1 Clock and Reset

UART controllers are asynchronous. Their register configuration module, TX FIFO and RX FIFO are in APB_CLK domain, while the UART Core that controls transmission and reception is in Core Clock domain. The three clock sources of the UART core, namely APB_CLK, RC_FAST_CLK and external crystal clock XTAL_CLK, are selected by configuring UART_SCLK_SEL. The selected clock source is divided by a divider. This divider supports fractional frequency division: UART_SCLK_DIV_NUM field is the integral part, UART_SCLK_DIV_B field is the numerator of the fractional part, and UART_SCLK_DIV_A is the denominator of the fractional part. The divisor ranges from 1 ~ 256.

When the frequency of the UART Core's clock is higher than the frequency needed to generate the baud rate, the UART Core can be clocked at a lower frequency by the divider, in order to reduce power consumption. Usually, the UART Core's clock frequency is lower than the APB_CLK's frequency, and can be divided by the largest divisor value when higher than the frequency needed to generate the baud rate. The frequency of the UART Core's clock can also be at most twice higher than the APB_CLK. The clock for the UART transmitter and the UART receiver can be controlled independently. To enable the clock for the UART transmitter, UART_TX_SCLK_EN shall be set; to enable the clock for the UART receiver, UART_RX_SCLK_EN shall be set.

To ensure that the configured register values are synchronized from APB_CLK domain to Core Clock domain,

please follow procedures in Section26.5.

To reset the whole UART, please:

- enable the clock for UART RAM by setting SYSTEM_UART_MEM_CLK_EN to 1;
- enable APB_CLK for UART*n* by setting SYSTEM_UART*n*_CLK_EN to 1
- clear SYSTEM_UART*n*_RST to 0;
- write 1 to UART_RST_CORE;
- write 1 to SYSTEM_UART*n*_RST;
- clear SYSTEM_UART*n*_RST to 0;
- clear UART_RST_CORE to 0.

Note that it is not recommended to reset the APB clock domain module or UART Core only.

## 26.4.2 UART RAM

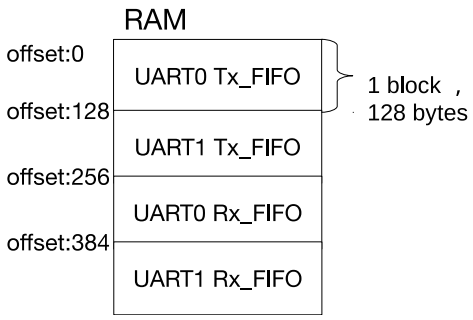



Figure 26.4-1. UART Controllers Sharing RAM

The two UART controllers on ESP32-C3 share 512 × 8 bits of FIFO RAM. As Figure 26.4-1 illustrates, RAM is divided into 4 blocks, each has 128 × 8 bits. Figure 26.4-1 shows how many RAM blocks are allocated to TX FIFOs and RX FIFOs of the two UART controllers by default. UART*n* Tx_FIFO can be expanded by configuring UART_TX_SIZE, while UART*n* Rx_FIFO can be expanded by configuring UART_RX_SIZE. Some limits are imposed:

- UART0 Tx_FIFO can be increased up to 4 blocks (the whole RAM);
- UART1 Tx_FIFO can be increased up to 3 blocks (from offset 128 to the end address);
- UART0 Rx_FIFO can be increased up to 2 blocks (from offset 256 to the end address);
- UART1 Rx_FIFO cannot be increased.

Please note that starting addresses of all FIFOs are fixed, so 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 GDMA, 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 GDMA. The two UART controllers share one GDMA channel.

The empty signal threshold for Tx_FIFO is configured by setting UART_TXFIFO_EMPTY_THRHD. When data stored in Tx_FIFO is less than UART_TXFIFO_EMPTY_THRHD, a UART_TXFIFO_EMPTY_INT interrupt is generated. The full signal threshold for Rx_FIFO is configured by setting UART_RXFIFO_FULL_THRHD. When data stored in Rx_FIFO is greater than UART_RXFIFO_FULL_THRHD, a UART_RXFIFO_FULL_INT interrupt is generated. In addition, when Rx_FIFO receives more data than its capacity, a UART_RXFIFO_OVF_INT interrupt is generated.

UART*n* can access FIFO via register UART_FIFO_REG. Writing to UART_RXFIFO_RD_BYTE stores the data into the TX FIFO. As UART_RXFIFO_RD_BYTE is a read-only register field, the hardware does not actually perform a write operation on UART_RXFIFO_RD_BYTE; instead, upon detecting a write request to this field's address, it passes the corresponding write data to the TX FIFO via a separate bypass. Reading UART_RXFIFO_RD_BYTE retrieves the data from the RX FIFO.

## 26.4.3 Baud Rate Generation and Detection

### 26.4.3.1 Baud Rate Generation

Before a UART controller sends or receives data, the baud rate should be configured by setting corresponding registers. The baud rate generator of a UART controller 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 the 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

$$UART_CLKDIV + \frac{UART_CLKDIV_FRAG}{16}$$

meaning that the final baud rate is equal to

$$\frac{INPUT_FREQ}{UART_CLKDIV + \frac{UART_CLKDIV_FRAG}{16}}$$

where INPUT_FREQ is the frequency of UART Core's source clock. For example, if UART_CLKDIV = 694 and UART_CLKDIV_FRAG = 7 then the divisor value is

$$694 + \frac{7}{16} = 694.4375$$

When UART_CLKDIV_FRAG is 0, 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 0, the divider is fractional and the output baud rate clock pulses are not strictly uniform. As shown in Figure 26.4-2, 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 - UART_CLKDIV_FRAG) output pulses are generated by dividing UART_CLKDIV input pulses.

The output pulses are interleaved as shown in Figure 26.4-2 below, to make the output timing more uniform:

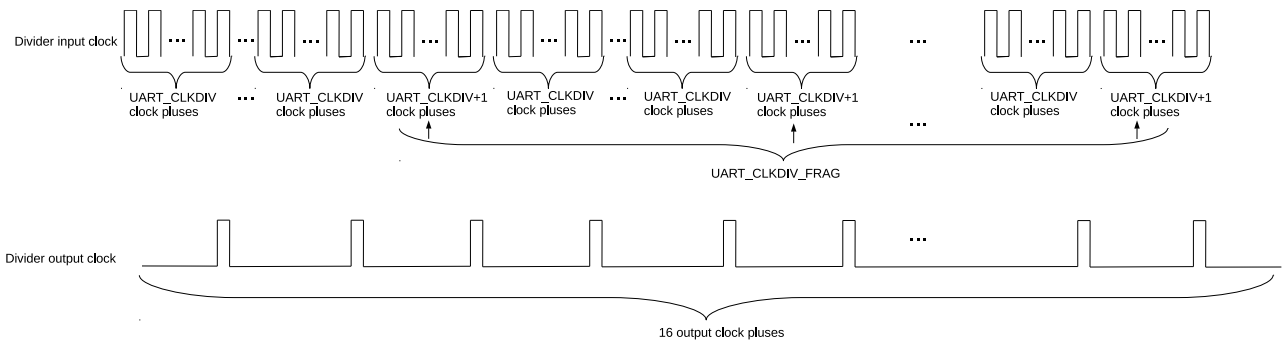



Figure 26.4-2. UART Controllers Division

To support IrDA (see Section 26.4.7 for details), the fractional clock divider for IrDA data transmission generates clock signals divided by 16 × 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.

### 26.4.3.2 Baud Rate Detection

Automatic baud rate detection (Autobaud) on UARTs is enabled by setting UART_AUTOBAUD_EN. The Baudrate_Detect module shown in Figure 26.3-2 filters any noise whose pulse width is shorter than UART_GLITCH_FILT.

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 fields are read by software to determine the transmitter's baud rate.

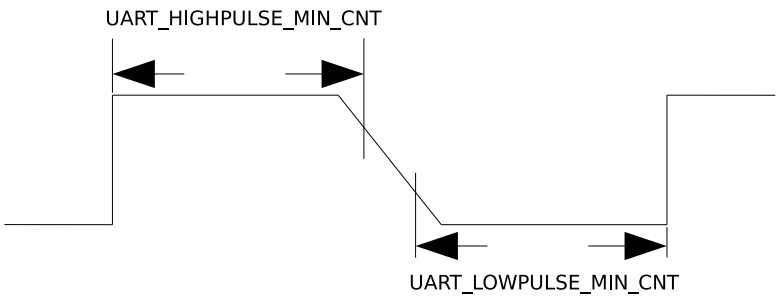



Figure 26.4-3. The Timing Diagram of Weak UART Signals Along Falling Edges

The baud rate can be determined in the following three ways:

1. Normally, to avoid sampling erroneous data along rising or falling edges in a metastable state, which results in the 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, the baud rate is calculated as follows:

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{(\text{UART_LOWPULSE_MIN_CNT} + \text{UART_HIGHPULSE_MIN_CNT} + 2)/2}$$

2. If UART signals are weak along falling edges as shown in Figure 26.4-3, which leads to an 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:

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{(\text{UART_POSEDGE_MIN_CNT} + 1)/2}$$

3. If UART signals are weak along rising edges, use UART_NEGEDGE_MIN_CNT to determine the transmitter's baud rate as follows:

$$B_{\text{uart}} = \frac{f_{\text{clk}}}{(\text{UART_NEGEDGE_MIN_CNT} + 1)/2}$$

### 26.4.4 UART Data Frame

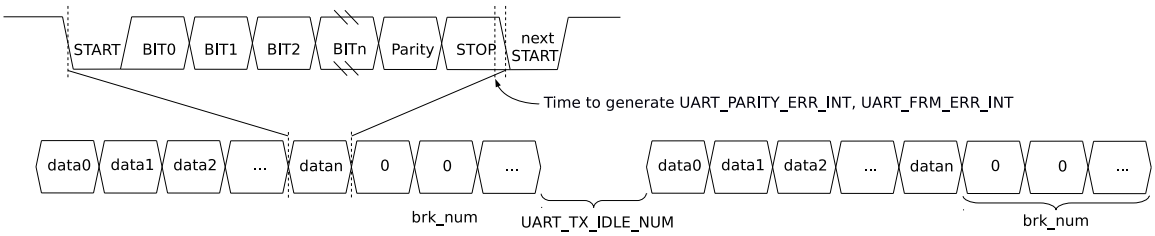



Figure 26.4-4. Structure of UART Data Frame

Figure 26.4-4 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, or 2 bits long, configured by UART_STOP_BIT_NUM (in RS485 mode turnaround delay may be added. See details in Section 26.4.6.2). 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 the data received, a UART_PARITY_ERR_INT interrupt is generated, and the 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 the 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 enter the Break condition and 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.

The receiver can also detect the Break conditions when the RX data line remains logical low for one NULL character transmission, and a UART_BRK_DET_INT interrupt will be triggered to detect that a Break condition has been completed.

The receiver can detect the current bus state through the timeout interrupt UART_RXFIFO_TOUT_INT. The UART_RXFIFO_TOUT_INT interrupt will be triggered when the bus is in the idle state for more than UART_RX_TOUT_THRHD bit time on current baud rate after the receiver has received at least one byte. You can use this interrupt to detect whether all the data from the transmitter has been sent.

## 26.4.5 AT_CMD Character Structure

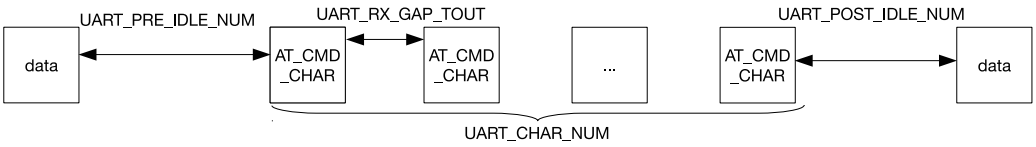



Figure 26.4-5. AT_CMD Character Structure

Figure 26.4-5 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 cycles.
- The interval between two AT_CMD_CHAR characters is less than UART_RX_GAP_TOUT 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 cycles.

## 26.4.6 RS485

The two UART controllers support RS485 protocol. This protocol 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.

### 26.4.6.1 Driver Control

As shown in Figure 26.4-6, in a two-wire multidrop network, an external RS485 transceiver is needed for differential to single-ended conversion. An 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 UART receiver 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 the data sent by itself.

DE can be controlled by either software or hardware. To reduce the cost of software, in our design DE is controlled by hardware. As shown in Figure 26.4-6, DE is connected to dtrn_out of UART (please refer to Section 26.4.9.1 for more details).

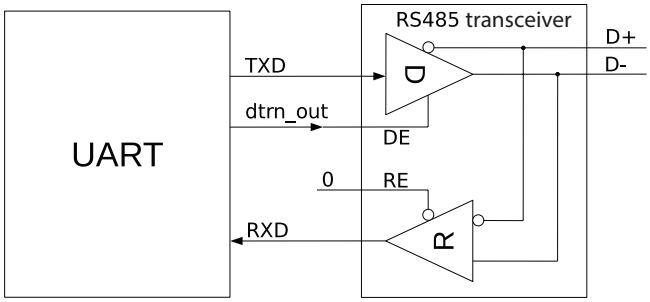



Figure 26.4-6. Driver Control Diagram in RS485 Mode

### 26.4.6.2 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 one cycle after the stop bit. The UART transmitter supports adding a turnaround delay of one cycle before the start bit or after the stop bit. When UART_DL0_EN is set, a turnaround delay of one cycle is added before the start bit; when UART_DL1_EN is set, a turnaround delay of one cycle is added after the stop bit.

### 26.4.6.3 Bus Snooping

In a two-wire multidrop network, UART controllers support bus snooping if RE of the external RS485 transceiver is 0. By default, a UART controller is not allowed to transmit and receive data simultaneously. If UART_RS485TX_RX_EN is set and the external RS485 transceiver is configured as in Figure 26.4-6, 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 the data sent by themselves. In transmitter mode, when a UART controller monitors a collision between the data sent and the 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.

## 26.4.7 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's 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 26.4-7, the IrDA encoder converts a NRZ (non-return to zero code) signal to a RZI (return to zero inverted 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 are high.

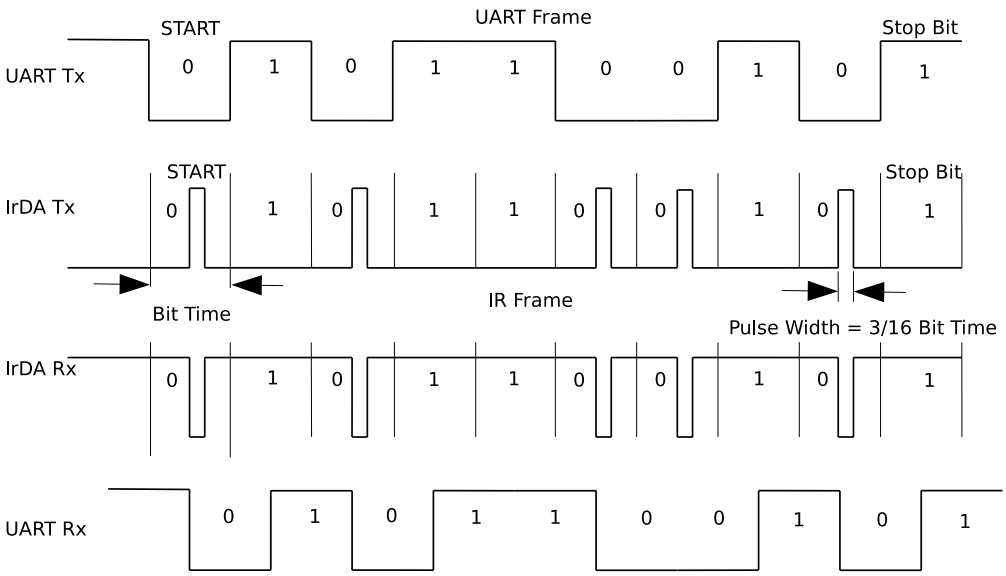



Figure 26.4-7. The Timing Diagram of Encoding and Decoding in SIR mode

The IrDA transceiver is half-duplex, meaning that it cannot send and receive data simultaneously. As shown in Figure 26.4-8, 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.

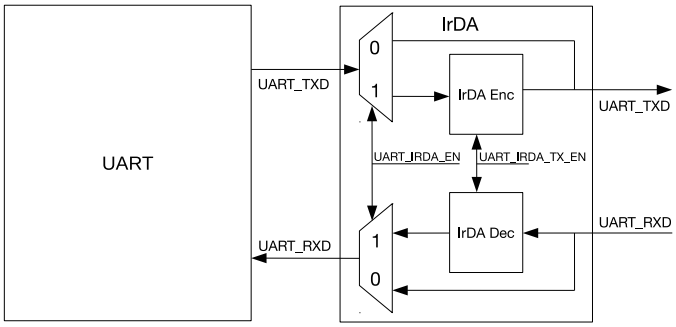



Figure 26.4-8. IrDA Encoding and Decoding Diagram

## 26.4.8 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 rxd_in. When the number of rising edges is is equal to or greater than (UART_ACTIVE_THRESHOLD + 3), a wake_up signal is generated and sent to RTC, which then wakes up ESP32-C3.

After the chip is woken up by UART, it is necessary to clear the wake_up signal by transmitting data to UART in Active mode or resetting the whole UART, otherwise the number of rising edges required for the next wakeup will be reduced.

## 26.4.9 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 rtsn_out and input signal dsrn_in. Software flow control is achieved by inserting special characters in the data flow sent and detecting special characters in the data flow received.

### 26.4.9.1 Hardware Flow Control

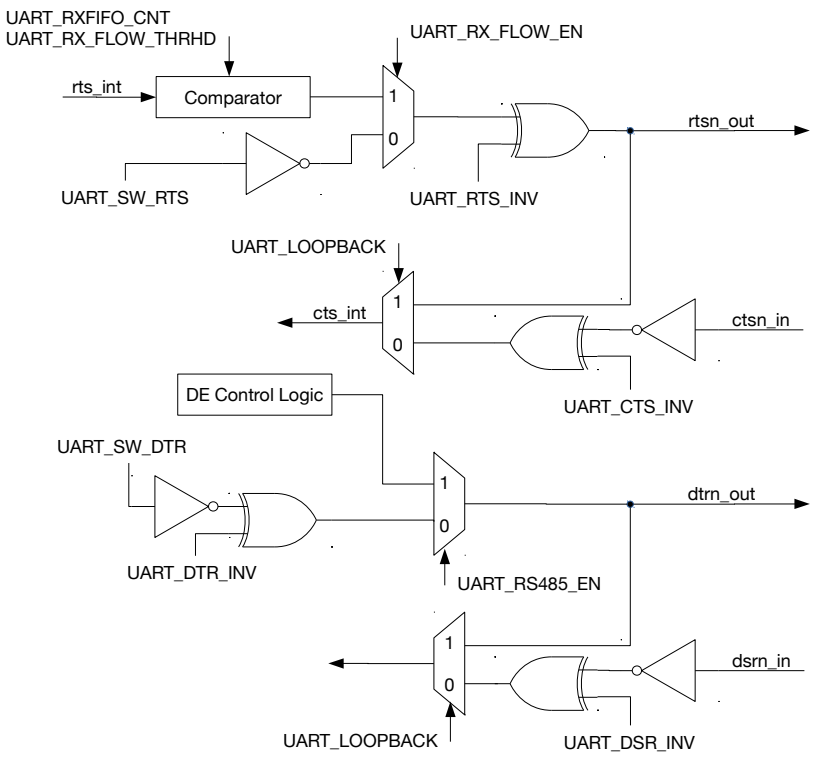



Figure 26.4-9. Hardware Flow Control Diagram

Figure 26.4-9 shows the hardware flow control of a UART controller. Hardware flow control uses output signal rtsn_out and input signal dsrn_in. Figure 26.4-10 illustrates how these signals are connected between UART on ESP32-C3 (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. The output signal rtsn_out can be controlled in two ways.

- Software control: Enter this mode by clearing 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.

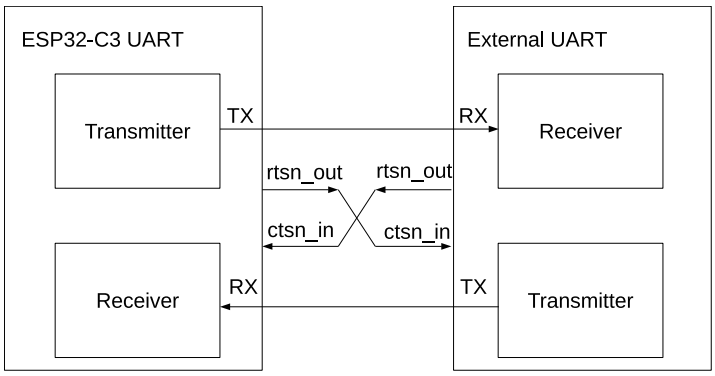



Figure 26.4-10. Connection between Hardware Flow Control Signals

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 the UART_SW_DTR field. 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 generated by hardware and used for transmit/receive turnaround. 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 a 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 the data sent matches the data received, it indicates that UART controllers are working properly.

### 26.4.9.2 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 the 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 configured by UART_XOFF_CHAR 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 configured by UART_XON_CHAR after the current byte in transmission.

### 26.4.10 GDMA Mode

The two UART controllers on ESP32-C3 share one TX/RX GDMA (general direct memory access) channel via UHCI. In GDMA mode, UART controllers support the decoding and encoding of HCI data packets. The UHCI_UART*n*_CE field determines which UART controller occupies the GDMA TX/RX channel.

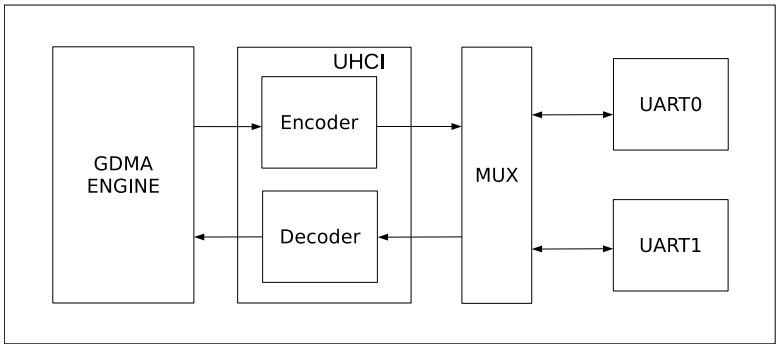



**Figure 26.4-11. Data Transfer in GDMA Mode**

Figure 26.4-11 shows how data is transferred using GDMA. Before GDMA receives data, software prepares an inlink. GDMA_INLINK_ADDR_CH*n* points to the first receive descriptor in the inlink. After GDMA_INLINK_START_CH*n* 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 GDMA.

Before GDMA sends data, software prepares an outlink and data to be sent. GDMA_OUTLINK_ADDR_CH*n* points to the first transmit descriptor in the outlink. After GDMA_OUTLINK_START_CH*n* is set, GDMA reads data from the RAM pointed by outlink. The data is then encoded by the encoder, and sent sequentially by the UART transmitter.

HCI data packets have separators at the beginning and the end, with data bits in the middle (separators + data bits + separators). 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 GDMA_OUT_TOTAL_EOF_CH*n*_INT interrupt is generated. When all data has been received, a GDMA_IN_SUC_EOF_CH*n*_INT is generated.

### 26.4.11 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 has sent all NULL characters after all data in TX FIFO had 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 (i.e. logic 0 for one NULL character transmission) 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.

### 26.4.12 UHCI Interrupts

- UHCI_APP_CTRL1_INT: Triggered when software sets UHCI_APP_CTRL1_INT_RAW.
- UHCI_APP_CTRL0_INT: Triggered when software sets UHCI_APP_CTRL0_INT_RAW.
- UHCI_OUTLINK_EOF_ERR_INT: Triggered when an EOF error is detected in a transmit descriptor.
- UHCI_SEND_A_REG_Q_INT: Triggered when UHCI has sent a series of short packets using always_send.
- UHCI_SEND_S_REG_Q_INT: Triggered when UHCI has sent a series of short packets using single_send.
- UHCI_TX_HUNG_INT: Triggered when UHCI takes too long to read RAM using a GDMA transmit channel.
- UHCI_RX_HUNG_INT: Triggered when UHCI takes too long to receive data using a GDMA receive channel.
- UHCI_TX_START_INT: Triggered when GDMA detects a separator character.

- UHCI_RX_START_INT: Triggered when a separator character has been sent.

## 26.5 Programming Procedures

### 26.5.1 Register Type

All UART registers are in APB_CLK domain. According to whether clock domain crossing and synchronization are required, UART registers that can be configured by software are classified into three types, namely immediate registers, synchronous registers, and static registers. Immediate registers are read in APB_CLK domain, and take effect after configured via the APB bus. Synchronous registers are read in Core Clock domain, and take effect after synchronization. Static registers are also read in Core Clock domain, but would not change dynamically. Therefore, for static registers clock domain crossing is not required, and software can turn on and off the clock for the UART transmitter or receiver to ensure that the configuration sampled in Core Clock domain is correct.

#### 26.5.1.1 Synchronous Registers

Read in Core Clock domain, synchronous registers implement the clock domain crossing design to ensure that their values sampled in Core Clock domain are correct. These registers as listed in Table 26.5-1 are configured as follows:

- Enable register synchronization by clearing UART_UPDATE_CTRL to 0;
- Wait for UART_REG_UPDATE to become 0, which indicates the completion of last synchronization;
- Configure synchronous registers;
- Synchronize the configured values to Core Clock domain by writting 1 to UART_REG_UPDATE.

Table 26.5-1. UART*n* Synchronous Registers

| Register | Field |
|---|---|
| UART_CLKDIV_REG | UART_CLKDIV_FRAG[3:0] |
| | UART_CLKDIV[11:0] |
| UART_CONF0_REG | UART_AUTOBAUD_EN |
| | UART_ERR_WR_MASK |
| | UART_TXD_INV |
| | UART_RXD_INV |
| | UART_IRDA_EN |
| | UART_TX_FLOW_EN |
| | UART_LOOPBACK |
| | UART_IRDA_RX_INV |
| | UART_IRDA_TX_EN |
| | UART_IRDA_WCTL |
| | UART_IRDA_TX_EN |
| | UART_IRDA_DPLX |
| | UART_STOP_BIT_NUM |
| | UART_BIT_NUM |

Cont'd on next page

Table 26.5-1 – cont'd from previous page

| Register | Field |
|---|---|
| | UART_PARITY_EN |
| | UART_PARITY |
| UART_FLOW_CONF_REG | UART_SEND_XOFF |
| | UART_SEND_XON |
| | UART_FORCE_XOFF |
| | UART_FORCE_XON |
| | UART_XONOFF_DEL |
| | UART_SW_FLOW_CON_EN |
| UART_TXBRK_CONF_REG | UART_RS485_TX_DLY_NUM[3:0] |
| | UART_RS485_RX_DLY_NUM |
| | UART_RS485RXBY_TX_EN |
| | UART_RS485TX_RX_EN |
| | UART_DL1_EN |
| | UART_DLO_EN |
| | UART_RS485_EN |

### 26.5.1.2 Static Registers

Static registers, though also read in Core Clock domain, would not change dynamically when UART controllers are at work, so they do not implement the clock domain crossing design. These registers must be configured when the UART transmitter or receiver is not at work. In this case, software can turn off the clock for the UART transmitter or receiver, so that static registers are not sampled in their metastable state. When software turns on the clock, the configured values are stable to be correctly sampled. Static registers as listed in Table 26.5-2 are configured as follows:

- Turn off the clock for the UART transmitter by clearing UART_TX_SCLK_EN, or the clock for the UART receiver by clearing UART_RX_SCLK_EN, depending on which one (transmitter or receiver) is not at work;
- Configure static registers;
- Turn on the clock for the UART transmitter by writing 1 to UART_TX_SCLK_EN, or the clock for the UART receiver by writing 1 to UART_RX_SCLK_EN.

Table 26.5-2. UART*n* Static Registers

| Register | Field |
|---|---|
| UART_RX_FILT_REG | UART_GLITCH_FILT_EN |
| | UART_GLITCH_FILT[7:0] |
| UART_SLEEP_CONF_REG | UART_ACTIVE_THRESHOLD[9:0] |
| UART_SWFC_CONFO_REG | UART_XOFF_CHAR[7:0] |
| UART_SWFC_CONF1_REG | UART_XON_CHAR[7:0] |
| UART_IDLE_CONF_REG | UART_TX_IDLE_NUM[9:0] |
| UART_AT_CMD_PRECNT_REG | UART_PRE_IDLE_NUM[15:0] |
| UART_AT_CMD_POSTCNT_REG | UART_POST_IDLE_NUM[15:0] |
| UART_AT_CMD_GAPTOUT_REG | UART_RX_GAP_TOUT[15:0] |

Cont'd on next page

Table 26.5-2 – cont'd from previous page

| Register | Field |
|---|---|
| UART_AT_CMD_CHAR_REG | UART_CHAR_NUM[7:0] |
| | UART_AT_CMD_CHAR[7:0] |

#### 26.5.1.3 Immediate Registers

Except those listed in Table 26.5-1 and Table 26.5-2, registers that can be configured by software are immediate registers read in APB_CLK domain, such as interrupt and FIFO configuration registers.

### 26.5.2 Detailed Steps

Figure 26.5-1 illustrates the process to program UART controllers, namely initialize UART, configure registers, enable the UART transmitter or receiver, and finish data transmission.

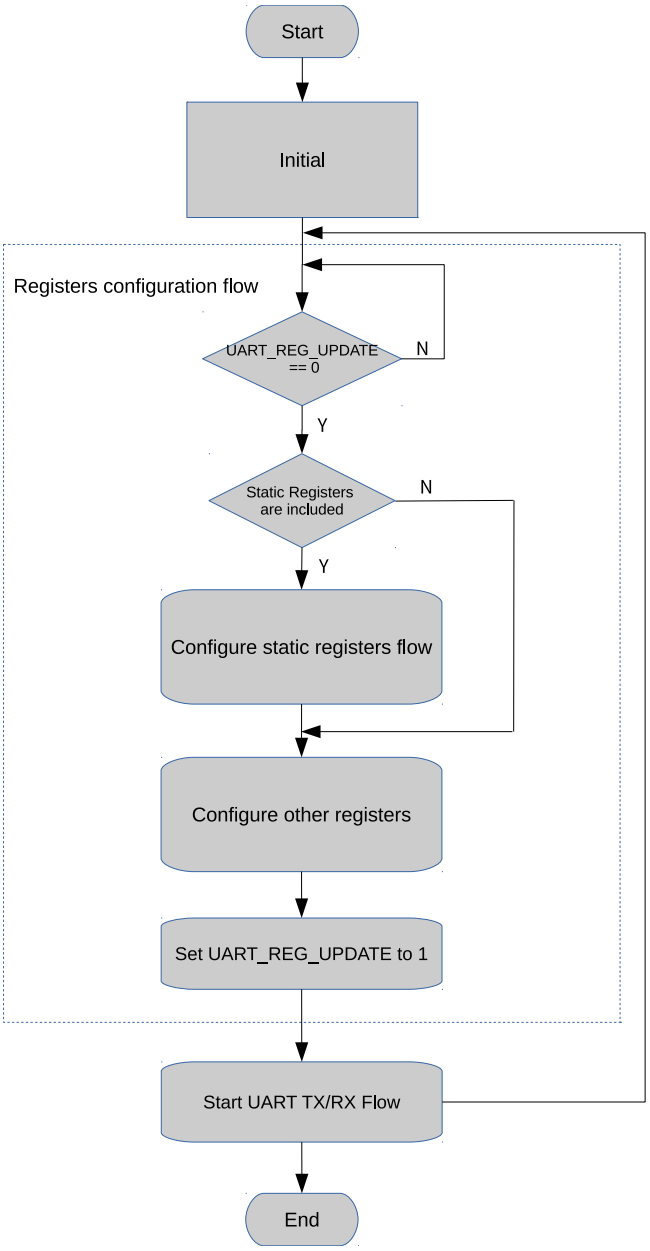



Figure 26.5-1. UART Programming Procedures

#### 26.5.2.1 Initializing UART*n*

To initialize UART*n*:

- enable the clock for UART RAM by setting SYSTEM_UART_MEM_CLK_EN to 1;
- enable APB_CLK for UART*n* by setting SYSTEM_UART*n*_CLK_EN to 1;
- clear SYSTEM_UART*n*_RST;
- write 1 to UART_RST_CORE;
- write 1 to SYSTEM_UART*n*_RST;
- clear SYSTEM_UART*n*_RST;
- clear UART_RST_CORE;
- enable register synchronization by clearing UART_UPDATE_CTRL.

#### 26.5.2.2 Configuring UART*n* Communication

To configure UART*n* communication:

- wait for UART_REG_UPDATE to become 0, which indicates the completion of the last synchronization;
- configure static registers (if any) following Section 26.5.1.2;
- select the clock source via UART_SCLK_SEL;
- configure divisor of the divider via UART_SCLK_DIV_NUM, UART_SCLK_DIV_A, and UART_SCLK_DIV_B;
- configure the baud rate for transmission via UART_CLKDIV and UART_CLKDIV_FRAG;
- configure data length via UART_BIT_NUM;
- configure odd or even parity check via UART_PARITY_EN and UART_PARITY;
- optional steps depending on application ...
- synchronize the configured values to the Core Clock domain by writing 1 to UART_REG_UPDATE.

#### 26.5.2.3 Enabling UART*n*

To enable UART*n* transmitter:

- configure TX FIFO's empty threshold via UART_TXFIFO_EMPTY_THRHD;
- disable UART_TXFIFO_EMPTY_INT interrupt by clearing UART_TXFIFO_EMPTY_INT_ENA;
- write data to be sent to UART_RXFIFO_RD_BYTE;
- clear UART_TXFIFO_EMPTY_INT interrupt by setting UART_TXFIFO_EMPTY_INT_CLR;
- enable UART_TXFIFO_EMPTY_INT interrupt by setting UART_TXFIFO_EMPTY_INT_ENA;
- detect UART_TXFIFO_EMPTY_INT and wait for the completion of data transmission.

To enable UART*n* receiver:

- configure RX FIFO's full threshold via UART_RXFIFO_FULL_THRHD;
- enable UART_RXFIFO_FULL_INT interrupt by setting UART_RXFIFO_FULL_INT_ENA;

- detect UART_TXFIFO_FULL_INT and wait until the RX FIFO is full;
- read data from RX FIFO via UART_RXFIFO_RD_BYTE, and obtain the number of bytes received in RX FIFO via UART_RXFIFO_CNT.

## 26.6 Register Summary

The addresses in this section are relative to UART Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **FIFO Configuration** | | | |
| UART_FIFO_REG | FIFO data register | 0x0000 | RO |
| UART_MEM_CONF_REG | UART threshold and allocation configuration | 0x0060 | R/W |
| **UART Interrupt Register** | | | |
| UART_INT_RAW_REG | Raw interrupt status | 0x0004 | R/WTC/SS |
| 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 | WT |
| **Configuration Register** | | | |
| UART_CLKDIV_REG | Clock divider configuration | 0x0014 | R/W |
| UART_RX_FILT_REG | RX filter configuration | 0x0018 | R/W |
| 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 | varies |
| UART_SLEEP_CONF_REG | Sleep 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_TXBRK_CONF_REG | TX break character configuration | 0x0044 | R/W |
| UART_IDLE_CONF_REG | Frame end idle time configuration | 0x0048 | R/W |
| UART_RS485_CONF_REG | RS485 mode configuration | 0x004C | R/W |
| UART_CLK_CONF_REG | UART core clock configuration | 0x0078 | R/W |
| **Status Register** | | | |
| UART_STATUS_REG | UART status register | 0x001C | RO |
| UART_MEM_TX_STATUS_REG | TX FIFO write and read offset address | 0x0064 | RO |
| UART_MEM_RX_STATUS_REG | RX FIFO write and read offset address | 0x0068 | RO |
| UART_FSM_STATUS_REG | UART transmitter and receiver status | 0x006C | RO |
| **Autobaud Register** | | | |
| 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 | 0x0070 | RO |
| UART_NEGPULSE_REG | Autobaud low pulse register | 0x0074 | RO |
| **AT Escape Sequence Selection Configuration** | | | |
| UART_AT_CMD_PRECNT_REG | Pre-sequence timing configuration | 0x0050 | R/W |
| UART_AT_CMD_POSTCNT_REG | Post-sequence timing configuration | 0x0054 | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| UART_AT_CMD_GAPTOUT_REG | Timeout configuration | 0x0058 | R/W |
| UART_AT_CMD_CHAR_REG | AT escape sequence detection configuration | 0x005C | R/W |
| **Version Register** | | | |
| UART_DATE_REG | UART version control register | 0x007C | R/W |
| UART_ID_REG | UART ID register | 0x0080 | varies |

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Register** | | | |
| UHCI_CONF0_REG | UHCI configuration register | 0x0000 | R/W |
| UHCI_CONF1_REG | UHCI configuration register | 0x0014 | varies |
| UHCI_ESCAPE_CONF_REG | Escape character configuration | 0x0020 | R/W |
| UHCI_HUNG_CONF_REG | Timeout configuration | 0x0024 | R/W |
| UHCI_ACK_NUM_REG | UHCI ACK number configuration | 0x0028 | varies |
| UHCI_QUICK_SENT_REG | UHCI quick_sent configuration register | 0x0030 | varies |
| UHCI_REG_Q0_WORD0_REG | Q0_WORD0 quick_sent register | 0x0034 | R/W |
| UHCI_REG_Q0_WORD1_REG | Q0_WORD1 quick_sent register | 0x0038 | R/W |
| UHCI_REG_Q1_WORD0_REG | Q1_WORD0 quick_sent register | 0x003C | R/W |
| UHCI_REG_Q1_WORD1_REG | Q1_WORD1 quick_sent register | 0x0040 | R/W |
| UHCI_REG_Q2_WORD0_REG | Q2_WORD0 quick_sent register | 0x0044 | R/W |
| UHCI_REG_Q2_WORD1_REG | Q2_WORD1 quick_sent register | 0x0048 | R/W |
| UHCI_REG_Q3_WORD0_REG | Q3_WORD0 quick_sent register | 0x004C | R/W |
| UHCI_REG_Q3_WORD1_REG | Q3_WORD1 quick_sent register | 0x0050 | R/W |
| UHCI_REG_Q4_WORD0_REG | Q4_WORD0 quick_sent register | 0x0054 | R/W |
| UHCI_REG_Q4_WORD1_REG | Q4_WORD1 quick_sent register | 0x0058 | R/W |
| UHCI_REG_Q5_WORD0_REG | Q5_WORD0 quick_sent register | 0x005C | R/W |
| UHCI_REG_Q5_WORD1_REG | Q5_WORD1 quick_sent register | 0x0060 | R/W |
| UHCI_REG_Q6_WORD0_REG | Q6_WORD0 quick_sent register | 0x0064 | R/W |
| UHCI_REG_Q6_WORD1_REG | Q6_WORD1 quick_sent register | 0x0068 | R/W |
| UHCI_ESC_CONF0_REG | Escape sequence configuration register 0 | 0x006C | R/W |
| UHCI_ESC_CONF1_REG | Escape sequence configuration register 1 | 0x0070 | R/W |
| UHCI_ESC_CONF2_REG | Escape sequence configuration register 2 | 0x0074 | R/W |
| UHCI_ESC_CONF3_REG | Escape sequence configuration register 3 | 0x0078 | R/W |
| UHCI_PKT_THRES_REG | Configuration register for packet length | 0x007C | R/W |
| **UHCI Interrupt Register** | | | |
| UHCI_INT_RAW_REG | Raw interrupt status | 0x0004 | varies |
| 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 | WT |
| **UHCI Status Register** | | | |
| UHCI_STATE0_REG | UHCI receive status | 0x0018 | RO |
| UHCI_STATE1_REG | UHCI transmit status | 0x001C | RO |
| UHCI_RX_HEAD_REG | UHCI packet header register | 0x002C | RO |
| **Version Register** | | | |

| Name | Description | Address | Access |
|---|---|---|---|
| UHCI_DATE_REG | UHCI version control register | 0x0080 | R/W |

## 26.7 Registers

The addresses in this section are relative to UART Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 26.1. UART_FIFO_REG (0x0000)**

| (reserved) | UART_RXFIFO_RD_BYTE | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**UART_RXFIFO_RD_BYTE** UART*n* accesses FIFO via this field. (RO)

**Register 26.2. UART_MEM_CONF_REG (0x0060)**

| (reserved) | UART_MEM_FORCE_PU | UART_MEM_FORCE_PD | UART_RX_TOUT_THRHD | UART_RX_FLOW_THRHD | UART_TX_SIZE | UART_RX_SIZE | (reserved) | |
|---|---|---|---|---|---|---|---|---|
| 31 … 28 | 27 | 26 | 25 … 16 | 15 … 7 | 6 … 4 | 3 … 1 | 0 | |
| 0 0 0 0 | 0 | 0 | 0xa | 0x0 | 0x1 | 1 | 0 | Reset |

**UART_RX_SIZE** This field is used to configure the amount of RAM allocated for RX FIFO. The default number is 128 bytes. (R/W)

**UART_TX_SIZE** This field is used to configure the amount of RAM allocated for TX FIFO. The default number is 128 bytes. (R/W)

**UART_RX_FLOW_THRHD** This field is used to configure the maximum amount of data bytes that can be received when hardware flow control works. (R/W)

**UART_RX_TOUT_THRHD** This field is used to configure the threshold time that the receiver takes to receive one byte, in the unit of bit time (the time it takes to transfer one bit). 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 RAM. (R/W)

**UART_MEM_FORCE_PU** Set this bit to force power up UART RAM. (R/W)

**Register 26.3. UART_INT_RAW_REG (0x0004)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | UART_WAKEUP_INT_RAW | UART_AT_CMD_CHAR_DET_INT_RAW | UART_RS485_CLASH_INT_RAW | UART_RS485_FRM_ERR_INT_RAW | UART_RS485_PARITY_ERR_INT_RAW | UART_TX_DONE_INT_RAW | UART_TX_BRK_IDLE_DONE_INT_RAW | UART_GLITCH_DET_INT_RAW | UART_SW_XOFF_INT_RAW | UART_SW_XON_INT_RAW | UART_RXFIFO_TOUT_INT_RAW | UART_BRK_DET_INT_RAW | UART_CTS_CHG_INT_RAW | UART_DSR_CHG_INT_RAW | UART_RXFIFO_OVF_INT_RAW | UART_FRM_ERR_INT_RAW | UART_PARITY_ERR_INT_RAW | UART_TXFIFO_EMPTY_INT_RAW | UART_RXFIFO_FULL_INT_RAW | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Reset |

**UART_RXFIFO_FULL_INT_RAW** This interrupt raw bit turns to high level when the receiver receives more data than what UART_RXFIFO_FULL_THRHD specifies. (R/WTC/SS)

**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. (R/WTC/SS)

**UART_PARITY_ERR_INT_RAW** This interrupt raw bit turns to high level when the receiver detects a parity error in the data. (R/WTC/SS)

**UART_FRM_ERR_INT_RAW** This interrupt raw bit turns to high level when the receiver detects a data frame error. (R/WTC/SS)

**UART_RXFIFO_OVF_INT_RAW** This interrupt raw bit turns to high level when the receiver receives more data than the capacity of RX FIFO. (R/WTC/SS)

**UART_DSR_CHG_INT_RAW** This interrupt raw bit turns to high level when the receiver detects the edge change of DSRn signal. (R/WTC/SS)

**UART_CTS_CHG_INT_RAW** This interrupt raw bit turns to high level when the receiver detects the edge change of CTSn signal. (R/WTC/SS)

**UART_BRK_DET_INT_RAW** This interrupt raw bit turns to high level when the receiver detects a 0 after the stop bit. (R/WTC/SS)

**UART_RXFIFO_TOUT_INT_RAW** This interrupt raw bit turns to high level when the receiver takes more time than UART_RX_TOUT_THRHD to receive a byte. (R/WTC/SS)

**UART_SW_XON_INT_RAW** This interrupt raw bit turns to high level when the receiver receives an XON character and UART_SW_FLOW_CON_EN is set to 1. (R/WTC/SS)

**UART_SW_XOFF_INT_RAW** This interrupt raw bit turns to high level when the receiver receives an XOFF character and UART_SW_FLOW_CON_EN is set to 1. (R/WTC/SS)

**UART_GLITCH_DET_INT_RAW** This interrupt raw bit turns to high level when the receiver detects a glitch in the middle of a start bit. (R/WTC/SS)

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

Register 26.3. UART_INT_RAW_REG (0x0004)

Continued from the previous page...

**UART_TX_BRK_DONE_INT_RAW** This interrupt raw bit turns to high level when the transmitter completes sending NULL characters, after all data in TX FIFO are sent. (R/WTC/SS)

**UART_TX_BRK_IDLE_DONE_INT_RAW** This interrupt raw bit turns to high level when the transmitter has kept the shortest duration after sending the last data. (R/WTC/SS)

**UART_TX_DONE_INT_RAW** This interrupt raw bit turns to high level when the transmitter has sent out all data in FIFO. (R/WTC/SS)

**UART_RS485_PARITY_ERR_INT_RAW** This interrupt raw bit turns to high level when the receiver detects a parity error from the echo of the transmitter in RS485 mode. (R/WTC/SS)

**UART_RS485_FRM_ERR_INT_RAW** This interrupt raw bit turns to high level when the receiver detects a data frame error from the echo of the transmitter in RS485 mode. (R/WTC/SS)

**UART_RS485_CLASH_INT_RAW** This interrupt raw bit turns to high level when a collision is detected between the transmitter and the receiver in RS485 mode. (R/WTC/SS)

**UART_AT_CMD_CHAR_DET_INT_RAW** This interrupt raw bit turns to high level when the receiver detects the configured UART_AT_CMD_CHAR. (R/WTC/SS)

**UART_WAKEUP_INT_RAW** This interrupt raw bit turns to high level when the input RXD edge changes more times than what (UART_ACTIVE_THRESHOLD + 3) specifies in Light-sleep mode. (R/WTC/SS)

**Register 26.4. 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | UART_WAKEUP_INT_ST | UART_AT_CMD_CHAR_DET_INT_ST | UART_RS485_CLASH_INT_ST | UART_RS485_FRM_ERR_INT_ST | UART_RS485_PARITY_ERR_INT_ST | UART_TX_DONE_INT_ST | UART_TX_BRK_IDLE_DONE_INT_ST | UART_TX_BRK_DONE_INT_ST | UART_GLITCH_DET_INT_ST | UART_SW_XOFF_INT_ST | UART_SW_XON_INT_ST | UART_RXFIFO_TOUT_INT_ST | UART_BRK_DET_INT_ST | UART_CTS_CHG_INT_ST | UART_DSR_CHG_INT_ST | UART_RXFIFO_OVF_INT_ST | UART_FRM_ERR_INT_ST | UART_PARITY_ERR_INT_ST | UART_TXFIFO_EMPTY_INT_ST | UART_RXFIFO_FULL_INT_ST | |
| 0 0 0 0 0 0 0 0 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 |

**UART_RXFIFO_FULL_INT_ST** This is the status bit for UART_RXFIFO_FULL_INT 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 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 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 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 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 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 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 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 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 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 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 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 when UART_TX_BRK_DONE_INT_ENA is set to 1. (RO)

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

**Register 26.4. UART_INT_ST_REG (0x0008)**

**Continued from the previous page...**

**UART_TX_BRK_IDLE_DONE_INT_ST** This is the status bit for UART_TX_BRK_IDLE_DONE_INT when UART_TX_BRK_IDLE_DONE_INT_ENA is set to 1. (RO)

**UART_TX_DONE_INT_ST** This is the status bit for UART_TX_DONE_INT 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 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 when UART_RS485_FRM_ERR_INT_ENA is set to 1. (RO)

**UART_RS485_CLASH_INT_ST** This is the status bit for UART_RS485_CLASH_INT 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_CHAR_DET_INT 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 when UART_WAKEUP_INT_ENA is set to 1. (RO)

**Register 26.5. UART_INT_ENA_REG (0x000C)**

| 31–20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | 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_TX_DONE_INT_ENA | UART_TX_BRK_IDLE_DONE_INT_ENA | UART_TX_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_BRK_DET_INT_ENA | UART_CTS_CHG_INT_ENA | UART_DSR_CHG_INT_ENA | UART_RXFIFO_OVF_INT_ENA | UART_FRM_ERR_INT_ENA | UART_PARITY_ERR_INT_ENA | UART_TXFIFO_EMPTY_INT_ENA | UART_RXFIFO_FULL_INT_ENA | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**UART_RXFIFO_FULL_INT_ENA** This is the enable bit for UART_RXFIFO_FULL_INT. (R/W)

**UART_TXFIFO_EMPTY_INT_ENA** This is the enable bit for UART_TXFIFO_EMPTY_INT. (R/W)

**UART_PARITY_ERR_INT_ENA** This is the enable bit for UART_PARITY_ERR_INT. (R/W)

**UART_FRM_ERR_INT_ENA** This is the enable bit for UART_FRM_ERR_INT. (R/W)

**UART_RXFIFO_OVF_INT_ENA** This is the enable bit for UART_RXFIFO_OVF_INT. (R/W)

**UART_DSR_CHG_INT_ENA** This is the enable bit for UART_DSR_CHG_INT. (R/W)

**UART_CTS_CHG_INT_ENA** This is the enable bit for UART_CTS_CHG_INT. (R/W)

**UART_BRK_DET_INT_ENA** This is the enable bit for UART_BRK_DET_INT. (R/W)

**UART_RXFIFO_TOUT_INT_ENA** This is the enable bit for UART_RXFIFO_TOUT_INT. (R/W)

**UART_SW_XON_INT_ENA** This is the enable bit for UART_SW_XON_INT. (R/W)

**UART_SW_XOFF_INT_ENA** This is the enable bit for UART_SW_XOFF_INT. (R/W)

**UART_GLITCH_DET_INT_ENA** This is the enable bit for UART_GLITCH_DET_INT. (R/W)

**UART_TX_BRK_DONE_INT_ENA** This is the enable bit for UART_TX_BRK_DONE_INT. (R/W)

**UART_TX_BRK_IDLE_DONE_INT_ENA** This is the enable bit for UART_TX_BRK_IDLE_DONE_INT. (R/W)

**UART_TX_DONE_INT_ENA** This is the enable bit for UART_TX_DONE_INT. (R/W)

Continued on the next page...

Register 26.5. UART_INT_ENA_REG (0x000C)

Continued from the previous page...

**UART_RS485_PARITY_ERR_INT_ENA** This is the enable bit for UART_RS485_PARITY_ERR_INT. (R/W)

**UART_RS485_FRM_ERR_INT_ENA** This is the enable bit for UART_RS485_PARITY_ERR_INT. (R/W)

**UART_RS485_CLASH_INT_ENA** This is the enable bit for UART_RS485_CLASH_INT. (R/W)

**UART_AT_CMD_CHAR_DET_INT_ENA** This is the enable bit for UART_AT_CMD_CHAR_DET_INT. (R/W)

**UART_WAKEUP_INT_ENA** This is the enable bit for UART_WAKEUP_INT. (R/W)

**Register 26.6. UART_INT_CLR_REG (0x0010)**

| 31 | | | | | | | | | | | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | UART_WAKEUP_INT_CLR | UART_AT_CMD_CHAR_DET_INT_CLR | UART_RS485_CLASH_INT_CLR | UART_RS485_FRM_ERR_INT_CLR | UART_RS485_PARITY_ERR_INT_CLR | UART_TX_DONE_INT_CLR | UART_TX_BRK_IDLE_DONE_INT_CLR | UART_TX_BRK_DONE_INT_CLR | UART_GLITCH_DET_INT_CLR | UART_SW_XOFF_INT_CLR | UART_SW_XON_INT_CLR | UART_RXFIFO_TOUT_INT_CLR | UART_BRK_DET_INT_CLR | UART_CTS_CHG_INT_CLR | UART_DSR_CHG_INT_CLR | UART_RXFIFO_OVF_INT_CLR | UART_FRM_ERR_INT_CLR | UART_PARITY_ERR_INT_CLR | UART_TXFIFO_EMPTY_INT_CLR | UART_RXFIFO_FULL_INT_CLR | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**UART_RXFIFO_FULL_INT_CLR** Set this bit to clear the UART_THE RXFIFO_FULL_INT interrupt. (WT)

**UART_TXFIFO_EMPTY_INT_CLR** Set this bit to clear the UART_TXFIFO_EMPTY_INT interrupt. (WT)

**UART_PARITY_ERR_INT_CLR** Set this bit to clear the UART_PARITY_ERR_INT interrupt. (WT)

**UART_FRM_ERR_INT_CLR** Set this bit to clear the UART_FRM_ERR_INT interrupt. (WT)

**UART_RXFIFO_OVF_INT_CLR** Set this bit to clear the UART_UART_RXFIFO_OVF_INT interrupt. (WT)

**UART_DSR_CHG_INT_CLR** Set this bit to clear the UART_DSR_CHG_INT interrupt. (WT)

**UART_CTS_CHG_INT_CLR** Set this bit to clear the UART_CTS_CHG_INT interrupt. (WT)

**UART_BRK_DET_INT_CLR** Set this bit to clear the UART_BRK_DET_INT interrupt. (WT)

**UART_RXFIFO_TOUT_INT_CLR** Set this bit to clear the UART_RXFIFO_TOUT_INT interrupt. (WT)

**UART_SW_XON_INT_CLR** Set this bit to clear the UART_SW_XON_INT interrupt. (WT)

**UART_SW_XOFF_INT_CLR** Set this bit to clear the UART_SW_XOFF_INT interrupt. (WT)

**UART_GLITCH_DET_INT_CLR** Set this bit to clear the UART_GLITCH_DET_INT interrupt. (WT)

**UART_TX_BRK_DONE_INT_CLR** Set this bit to clear the UART_TX_BRK_DONE_INT interrupt. (WT)

**UART_TX_BRK_IDLE_DONE_INT_CLR** Set this bit to clear the UART_TX_BRK_IDLE_DONE_INT interrupt. (WT)

**UART_TX_DONE_INT_CLR** Set this bit to clear the UART_TX_DONE_INT interrupt. (WT)

**UART_RS485_PARITY_ERR_INT_CLR** Set this bit to clear the UART_RS485_PARITY_ERR_INT interrupt. (WT)

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

### Register 26.6. UART_INT_CLR_REG (0x0010)

Continued from the previous page...

**UART_RS485_FRM_ERR_INT_CLR** Set this bit to clear the UART_RS485_FRM_ERR_INT interrupt. (WT)

**UART_RS485_CLASH_INT_CLR** Set this bit to clear the UART_RS485_CLASH_INT interrupt. (WT)

**UART_AT_CMD_CHAR_DET_INT_CLR** Set this bit to clear the UART_AT_CMD_CHAR_DET_INT interrupt. (WT)

**UART_WAKEUP_INT_CLR** Set this bit to clear the UART_WAKEUP_INT interrupt. (WT)

### Register 26.7. UART_CLKDIV_REG (0x0014)

| (reserved) | UART_CLKDIV_FRAG | (reserved) | UART_CLKDIV | |
|---|---|---|---|---|
| 31 24 | 23 20 | 19 12 | 11 0 | |
| 0 0 0 0 0 0 0 0 | 0x0 | 0 0 0 0 0 0 0 0 | 0x2b6 | Reset |

**UART_CLKDIV** The integral part of the frequency divisor. (R/W)

**UART_CLKDIV_FRAG** The fractional part of the frequency divisor. (R/W)

### Register 26.8. UART_RX_FILT_REG (0x0018)

| (reserved) | UART_GLITCH_FILT_EN | UART_GLITCH_FILT | |
|---|---|---|---|
| 31 9 | 8 | 7 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0x8 | Reset |

**UART_GLITCH_FILT** When input pulse width is lower than this value, the pulse is ignored. (R/W)

**UART_GLITCH_FILT_EN** Set this bit to enable RX signal filter. (R/W)

**Register 26.9. UART_CONF0_REG (0x0020)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | UART_MEM_CLK_EN | UART_AUTOBAUD_EN | UART_ERR_WR_MASK | UART_CLK_EN | UART_DTR_INV | UART_RTS_INV | UART_TXD_INV | UART_DSR_INV | UART_CTS_INV | UART_RXD_INV | UART_TXFIFO_RST | UART_RXFIFO_RST | UART_IRDA_EN | UART_TX_FLOW_EN | UART_LOOPBACK | UART_IRDA_RX_INV | UART_IRDA_TX_INV | UART_IRDA_WCTL | UART_IRDA_TX_EN | UART_IRDA_DPLX | UART_TXD_BRK | UART_SW_DTR | UART_SW_RTS | UART_STOP_BIT_NUM | UART_BIT_NUM | UART_PARITY_EN | UART_PARITY | |
| 0 | 1 | 0 | 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 bit is used to configure the parity check mode. (R/W)

**UART_PARITY_EN** Set this bit to enable UART parity check. (R/W)

**UART_BIT_NUM** This field is used to set the length of data. (R/W)

**UART_STOP_BIT_NUM** This field is used to set the length of stop bit. (R/W)

**UART_SW_RTS** This bit is used to configure the software RTS signal which is used in software flow control. (R/W)

**UART_SW_DTR** This bit 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 the transmitter to send NULL characters 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: The IrDA transmitter's 11th bit is the same as 10th bit; 0: 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 the 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 invert the level value of UART RXD signal. (R/W)

**UART_CTS_INV** Set this bit to invert the level value of UART CTS signal. (R/W)

**UART_DSR_INV** Set this bit to invert the level value of UART DSR signal. (R/W)

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

### Register 26.9. UART_CONF0_REG (0x0020)

Continued from the previous page...

**UART_TXD_INV** Set this bit to invert the level value of UART TXD signal. (R/W)

**UART_RTS_INV** Set this bit to invert the level value of UART RTS signal. (R/W)

**UART_DTR_INV** Set this bit to invert the level value of UART DTR signal. (R/W)

**UART_CLK_EN** 1: Force clock on for register; 0: Support clock only when application writes registers. (R/W)

**UART_ERR_WR_MASK** 1: The receiver stops storing data into FIFO when data is wrong; 0: The receiver stores the data even if the received data is wrong. (R/W)

**UART_AUTOBAUD_EN** This is the enable bit for baud rate detection. (R/W)

**UART_MEM_CLK_EN** The signal to enable UART RAM clock gating. (R/W)

### Register 26.10. UART_CONF1_REG (0x0024)

| (reserved) | UART_RX_TOUT_EN | UART_RX_FLOW_EN | UART_RX_TOUT_FLOW_DIS | UART_DIS_RX_DAT_OVF | UART_TXFIFO_EMPTY_THRHD | UART_RXFIFO_FULL_THRHD | |
|---|---|---|---|---|---|---|---|
| 31 22 | 21 | 20 | 19 | 18 | 17 9 | 8 0 | |
| 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | 0x60 | 0x60 | Reset |

**UART_RXFIFO_FULL_THRHD** An UART_RXFIFO_FULL_INT interrupt is generated when the receiver receives more data than the value of this field. (R/W)

**UART_TXFIFO_EMPTY_THRHD** An UART_TXFIFO_EMPTY_INT interrupt is generated when the number of data bytes in TX FIFO is less than the value of this field. (R/W)

**UART_DIS_RX_DAT_OVF** Disable UART RX data overflow detection. (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. (R/W)

**UART_RX_TOUT_EN** This is the enable bit for UART receiver's timeout function. (R/W)

**Register 26.11. UART_FLOW_CONF_REG (0x0034)**

| (reserved) | UART_SEND_XOFF | UART_SEND_XON | UART_FORCE_XOFF | UART_FORCE_XON | UART_XONOFF_DEL | UART_SW_FLOW_CON_EN | |
|---|---|---|---|---|---|---|---|
| 31 … 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 | Reset |

**UART_SW_FLOW_CON_EN** Set this bit to enable software flow control. When UART receives flow control characters XON or XOFF, which can be configured by UART_XON_CHAR or UART_XOFF_CHAR respectively, UART_SW_XON_INT or UART_SW_XOFF_INT interrupts can be triggered if enabled. (R/W)

**UART_XONOFF_DEL** Set this bit to remove flow control characters from the received data. (R/W)

**UART_FORCE_XON** Set this bit to force the transmitter to send 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 an XON character. This bit is cleared by hardware automatically. (R/W/SS/SC)

**UART_SEND_XOFF** Set this bit to send an XOFF character. This bit is cleared by hardware automatically. (R/W/SS/SC)

**Register 26.12. UART_SLEEP_CONF_REG (0x0038)**

| (reserved) | UART_ACTIVE_THRESHOLD | |
|---|---|---|
| 31 … 10 | 9 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0xf0 | Reset |

**UART_ACTIVE_THRESHOLD** UART is activated from Light-sleep mode when the input RXD edge changes more times than the value of this field plus 3. (R/W)

### Register 26.13. UART_SWFC_CONF0_REG (0x003C)

| (reserved) | UART_XOFF_CHAR | UART_XOFF_THRESHOLD | |
|---|---|---|---|
| 31 … 17 | 16 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x13 | 0xe0 | Reset |

**UART_XOFF_THRESHOLD** When the number of data bytes in RX FIFO is more than the value of this field with UART_SW_FLOW_CON_EN set to 1, the transmitter sends an XOFF character. (R/W)

**UART_XOFF_CHAR** This field stores the XOFF flow control character. (R/W)

### Register 26.14. UART_SWFC_CONF1_REG (0x0040)

| (reserved) | UART_XON_CHAR | UART_XON_THRESHOLD | |
|---|---|---|---|
| 31 … 17 | 16 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x11 | 0x0 | Reset |

**UART_XON_THRESHOLD** When the number of data bytes in RX FIFO is less than the value of this field with UART_SW_FLOW_CON_EN set to 1, the transmitter sends an XON character. (R/W)

**UART_XON_CHAR** This field stores the XON flow control character. (R/W)

### Register 26.15. UART_TXBRK_CONF_REG (0x0044)

| (reserved) | UART_TX_BRK_NUM | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0xa | Reset |

**UART_TX_BRK_NUM** This field 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 26.16. UART_IDLE_CONF_REG (0x0048)**

| (reserved) | UART_TX_IDLE_NUM | UART_RX_IDLE_THRHD | |
|---|---|---|---|
| 31 … 20 | 19 … 10 | 9 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 | 0x100 | 0x100 | Reset |

**UART_RX_IDLE_THRHD** A frame end signal is generated when the receiver takes more time to receive one byte data than the value of this field, in the unit of bit time (the time it takes to transfer one bit). (R/W)

**UART_TX_IDLE_NUM** This field is used to configure the duration time between transfers, in the unit of bit time (the time it takes to transfer one bit). (R/W)

**Register 26.17. UART_RS485_CONF_REG (0x004C)**

| (reserved) | UART_RS485_TX_DLY_NUM | UART_RS485_RX_DLY_NUM | UART_RS485RXBY_TX_EN | UART_RS485TX_RX_EN | UART_DL1_EN | UART_DL0_EN | UART_RS485_EN | |
|---|---|---|---|---|---|---|---|---|
| 31 … 10 | 9 … 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 | Reset |

**UART_RS485_EN** Set this bit to choose RS485 mode. (R/W)

**UART_DL0_EN** Configures whether or not to add a turnaround delay of 1 bit before the start bit.
0: Not add
1: Add
(R/W)

**UART_DL1_EN** Configures whether or not to add a turnaround delay of 1 bit after the stop bit.
0: Not add
1: Add
(R/W)

**UART_RS485TX_RX_EN** Set this bit to enable the receiver could receive data when the transmitter is transmitting data in RS485 mode. (R/W)

**UART_RS485RXBY_TX_EN** 1: enable RS485 transmitter to send data when RS485 receiver line is busy. (R/W)

**UART_RS485_RX_DLY_NUM** This bit is used to delay the receiver's internal data signal. (R/W)

**UART_RS485_TX_DLY_NUM** This field is used to delay the transmitter's internal data signal. (R/W)

### Register 26.18. UART_CLK_CONF_REG (0x0078)

| (reserved) | UART_RX_SCLK_EN | UART_TX_SCLK_EN | UART_RST_CORE | UART_SCLK_EN | UART_SCLK_SEL | UART_SCLK_DIV_NUM | UART_SCLK_DIV_A | UART_SCLK_DIV_B | |
|---|---|---|---|---|---|---|---|---|---|
| 31 – 26 | 25 | 24 | 23 | 22 | 21 – 20 | 19 – 12 | 11 – 6 | 5 – 0 | |
| 0 0 0 0 0 0 | 1 | 1 | 0 | 1 | 3 | 0x1 | 0x0 | 0x0 | Reset |

**UART_SCLK_DIV_B** The denominator of the frequency divisor. (R/W)

**UART_SCLK_DIV_A** The numerator of the frequency divisor. (R/W)

**UART_SCLK_DIV_NUM** The integral part of the frequency divisor. (R/W)

**UART_SCLK_SEL** Selects UART clock source. 1: APB_CLK; 2: RC_FAST_CLK; 3: XTAL_CLK. (R/W)

**UART_SCLK_EN** Set this bit to enable UART TX/RX clock. (R/W)

**UART_RST_CORE** Write 1 and then write 0 to this bit, to reset UART TX/RX. (R/W)

**UART_TX_SCLK_EN** Set this bit to enable UART TX clock. (R/W)

**UART_RX_SCLK_EN** Set this bit to enable UART RX clock. (R/W)

### Register 26.19. UART_STATUS_REG (0x001C)

| UART_TXD | UART_RTSN | UART_DTRN | (reserved) | UART_TXFIFO_CNT | UART_RXD | UART_CTSN | UART_DSRN | (reserved) | UART_RXFIFO_CNT | |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 – 26 | 25 – 16 | 15 | 14 | 13 | 12 – 10 | 9 – 0 | |
| 1 | 1 | 1 | 0 0 0 | 0 | 1 | 1 | 0 | 0 0 0 | 0 | Reset |

**UART_RXFIFO_CNT** Stores the number of valid data bytes in RX FIFO. (RO)

**UART_DSRN** This bit represents the level of the internal UART DSR signal. (RO)

**UART_CTSN** This bit represents the level of the internal UART CTS signal. (RO)

**UART_RXD** This bit represents the level of the internal UART RXD signal. (RO)

**UART_TXFIFO_CNT** Stores the number of data bytes 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 26.20. UART_MEM_TX_STATUS_REG (0x0064)**

| (reserved) | UART_TX_RADDR | (reserved) | UART_APB_TX_WADDR | |
|---|---|---|---|---|
| 31 … 21 | 20 … 11 | 10 | 9 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | 0 | 0x0 | Reset |

**UART_APB_TX_WADDR** This field stores the offset address in TX FIFO when software writes TX FIFO via APB. (RO)

**UART_TX_RADDR** This field stores the offset address in TX FIFO when TX FSM reads data via Tx_FIFO_Ctrl. (RO)

**Register 26.21. UART_MEM_RX_STATUS_REG (0x0068)**

| (reserved) | UART_RX_WADDR | (reserved) | UART_APB_RX_RADDR | |
|---|---|---|---|---|
| 31 … 21 | 20 … 11 | 10 | 9 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 | 0x100 | 0 | 0x100 | Reset |

**UART_APB_RX_RADDR** This field stores the offset address in RX FIFO when software reads data from RX FIFO via APB. UART0 is 0x200. UART1 is 0x280. (RO)

**UART_RX_WADDR** This field stores the offset address in RX FIFO when Rx_FIFO_Ctrl writes RX FIFO. (RO)

**Register 26.22. UART_FSM_STATUS_REG (0x006C)**

| (reserved) | UART_ST_UTX_OUT | UART_ST_URX_OUT | |
|---|---|---|---|
| 31 … 8 | 7 … 4 | 3 … 0 | |
| 0 0 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 |

**UART_ST_URX_OUT** This is the status field of the receiver. (RO)

**UART_ST_UTX_OUT** This is the status field of the transmitter. (RO)

**Register 26.23. UART_LOWPULSE_REG (0x0028)**

| (reserved) | UART_LOWPULSE_MIN_CNT | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0xfff | Reset |

**UART_LOWPULSE_MIN_CNT** This field stores the value of the minimum duration time of the low level pulse, in the unit of APB_CLK cycles. It is used in baud rate detection. (RO)

**Register 26.24. UART_HIGHPULSE_REG (0x002C)**

| (reserved) | UART_HIGHPULSE_MIN_CNT | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0xfff | Reset |

**UART_HIGHPULSE_MIN_CNT** This field stores the value of the maximum duration time for the high level pulse, in the unit of APB_CLK cycles. It is used in baud rate detection. (RO)

**Register 26.25. UART_RXD_CNT_REG (0x0030)**

| (reserved) | UART_RXD_EDGE_CNT | |
|---|---|---|
| 31 … 10 | 9 … 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 |

**UART_RXD_EDGE_CNT** This field stores the count of RXD edge change. It is used in baud rate detection. (RO)

**Register 26.26. UART_POSPULSE_REG (0x0070)**

| (reserved) | UART_POSEDGE_MIN_CNT | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0xfff | Reset |

**UART_POSEDGE_MIN_CNT** This field stores the minimal input clock count between two positive edges. It is used in baud rate detection. (RO)

**Register 26.27. UART_NEGPULSE_REG (0x0074)**

| (reserved) | UART_NEGEDGE_MIN_CNT | |
|---|---|---|
| 31 … 12 | 11 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0xfff | Reset |

**UART_NEGEDGE_MIN_CNT** This field stores the minimal input clock count between two negative edges. It is used in baud rate detection. (RO)

**Register 26.28. UART_AT_CMD_PRECNT_REG (0x0050)**

| (reserved) | UART_PRE_IDLE_NUM | |
|---|---|---|
| 31 … 16 | 15 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x901 | Reset |

**UART_PRE_IDLE_NUM** This field is used to configure the idle duration time before the first AT_CMD is received by the receiver, in the unit of bit time (the time it takes to transfer one bit). (R/W)

### Register 26.29. UART_AT_CMD_POSTCNT_REG (0x0054)

| (reserved) | UART_POST_IDLE_NUM | |
|---|---|---|
| 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 field is used to configure the duration time between the last AT_CMD and the next data byte, in the unit of bit time (the time it takes to transfer one bit). (R/W)

### Register 26.30. UART_AT_CMD_GAPTOUT_REG (0x0058)

| (reserved) | UART_RX_GAP_TOUT | |
|---|---|---|
| 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 field is used to configure the duration time between the AT_CMD characters, in the unit of bit time (the time it takes to transfer one bit). (R/W)

### Register 26.31. UART_AT_CMD_CHAR_REG (0x005C)

| (reserved) | UART_CHAR_NUM | UART_AT_CMD_CHAR | |
|---|---|---|---|
| 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 field is used to configure the content of AT_CMD character. (R/W)

**UART_CHAR_NUM** This field is used to configure the number of continuous AT_CMD characterss received by the receiver. (R/W)

**Register 26.32. UART_DATE_REG (0x007C)**

| UART_DATE | |
|---|---|
| 31 … 0 | |
| 0x2008270 | Reset |

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

**Register 26.33. UART_ID_REG (0x0080)**

| UART_REG_UPDATE | UART_UPDATE_CTRL | UART_ID | |
|---|---|---|---|
| 31 | 30 | 29 … 0 | |
| 0 | 1 | 0x000500 | Reset |

**UART_ID** This field is used to configure the UART_ID. (R/W)

**UART_UPDATE_CTRL** This bit is used to control register synchronization mode. This bit must be cleared before writing 1 to UART_REG_UPDATE to synchronize configured values to UART Core's clock domain. (R/W)

**UART_REG_UPDATE** When this bit is set to 1 by software, registers are synchronized to UART Core's clock domain. This bit is cleared by hardware after synchronization is done. (R/W/SC)

**Register 26.34. UHCI_CONF0_REG (0x0000)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | UHCI_UART_RX_BRK_EOF_EN | UHCI_CLK_EN | UHCI_ENCODE_CRC_EN | UHCI_LEN_EOF_EN | UHCI_UART_IDLE_EOF_EN | UHCI_CRC_REC_EN | UHCI_HEAD_EN | UHCI_SEPER_EN | (reserved) | UHCI_UART1_CE | UHCI_UART0_CE | UHCI_RX_RST | UHCI_TX_RST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | Reset |

**UHCI_TX_RST** Write 1, then write 0 to this bit to reset decode state machine. (R/W)

**UHCI_RX_RST** Write 1, then write 0 to this bit to reset encode state machine. (R/W)

**UHCI_UART0_CE** Set this bit to link up UHCI and UART0. (R/W)

**UHCI_UART1_CE** Set this bit to link up UHCI and UART1. (R/W)

**UHCI_SEPER_EN** Set this bit to separate the data frame using a special character. (R/W)

**UHCI_HEAD_EN** Set this bit to encode the data packet with a formatting header. (R/W)

**UHCI_CRC_REC_EN** Set this bit to enable UHCI to receive the 16 bit CRC. (R/W)

**UHCI_UART_IDLE_EOF_EN** If this bit is set to 1, UHCI will end the payload receiving process when UART has been in idle state. (R/W)

**UHCI_LEN_EOF_EN** If this bit is set to 1, UHCI decoder stops receiving payload data when the number of received data bytes has reached the specified value. The value is payload length indicated by UHCI packet header when UHCI_HEAD_EN is 1 or the value is configuration value when UHCI_HEAD_EN is 0. If this bit is set to 0, UHCI decoder stops receiving payload data when 0xC0 has been received. (R/W)

**UHCI_ENCODE_CRC_EN** Set this bit to enable data integrity check by appending a 16 bit CCITT-CRC to end of the payload. (R/W)

**UHCI_CLK_EN** 1: Force clock on for register; 0: 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 receive process when NULL frame is received by UART. (R/W)

**Register 26.35. UHCI_CONF1_REG (0x0014)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | UHCI_SW_START | UHCI_WAIT_SW_START | (reserved) | UHCI_TX_ACK_NUM_RE | UHCI_TX_CHECK_SUM_RE | UHCI_SAVE_HEAD | UHCI_CRC_DISABLE | UHCI_CHECK_SEQ_EN | UHCI_CHECK_SUM_EN | |
| 0 | 0 | 0 | 0 | 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_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 UHCI 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 acknowledgment when a reliable packet is to be transmitted. (R/W)

**UHCI_WAIT_SW_START** The UHCI der will jump to ST_SW_WAIT status if this bit 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/SC)

**Register 26.36. UHCI_ESCAPE_CONF_REG (0x0020)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | UHCI_RX_13_ESC_EN | UHCI_RX_11_ESC_EN | UHCI_RX_DB_ESC_EN | UHCI_RX_C0_ESC_EN | UHCI_TX_13_ESC_EN | UHCI_TX_11_ESC_EN | UHCI_TX_DB_ESC_EN | UHCI_TX_C0_ESC_EN | |
| 0 | 0 | 0 | 0 | 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 decode character 0xC0 when DMA receives data. (R/W)

**UHCI_TX_DB_ESC_EN** Set this bit to decode character 0xDB when DMA receives data. (R/W)

**UHCI_TX_11_ESC_EN** Set this bit to decode flow control character 0x11 when DMA receives data. (R/W)

**UHCI_TX_13_ESC_EN** Set this bit to decode flow control character 0x13 when DMA receives data. (R/W)

**UHCI_RX_C0_ESC_EN** Set this bit to replace 0xC0 by special characters when DMA sends data. (R/W)

**UHCI_RX_DB_ESC_EN** Set this bit to replace 0xDB by special characters when DMA sends data. (R/W)

**UHCI_RX_11_ESC_EN** Set this bit to replace flow control character 0x11 by special characters when DMA sends data. (R/W)

**UHCI_RX_13_ESC_EN** Set this bit to replace flow control character 0x13 by special characters when DMA sends data. (R/W)

**Register 26.37. UHCI_HUNG_CONF_REG (0x0024)**

| (reserved) | UHCI_RXFIFO_TIMEOUT_ENA | UHCI_RXFIFO_TIMEOUT_SHIFT | UHCI_RXFIFO_TIMEOUT | UHCI_TXFIFO_TIMEOUT_ENA | UHCI_TXFIFO_TIMEOUT_SHIFT | UHCI_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 |

**UHCI_TXFIFO_TIMEOUT** This field stores the timeout value. UHCI will produce the UHCI_TX_HUNG_INT interrupt when DMA takes more time to receive data. (R/W)

**UHCI_TXFIFO_TIMEOUT_SHIFT** This field is used to configure the maximum tick count. (R/W)

**UHCI_TXFIFO_TIMEOUT_ENA** This is the enable bit for TX FIFO receive timeout. (R/W)

**UHCI_RXFIFO_TIMEOUT** This field stores the timeout value. UHCI will produce the UHCI_RX_HUNG_INT interrupt when DMA takes more time to read data from RAM. (R/W)

**UHCI_RXFIFO_TIMEOUT_SHIFT** This field is used to configure the maximum tick count. (R/W)

**UHCI_RXFIFO_TIMEOUT_ENA** This is the enable bit for DMA send timeout. (R/W)

**Register 26.38. UHCI_ACK_NUM_REG (0x0028)**

| (reserved) | UHCI_ACK_NUM_LOAD | UHCI_ACK_NUM | |
|---|---|---|---|
| 31 – 4 | 3 | 2 – 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 | 0x0 | Reset |

**UHCI_ACK_NUM** This is the ACK number used in software flow control. (R/W)

**UHCI_ACK_NUM_LOAD** Set this bit to 1, and the value configured by UHCI_ACK_NUM would be loaded. (WT)

### Register 26.39. UHCI_QUICK_SENT_REG (0x0030)

| (reserved) | UHCI_ALWAYS_SEND_EN | UHCI_ALWAYS_SEND_NUM | UHCI_SINGLE_SEND_EN | UHCI_SINGLE_SEND_NUM | |
|---|---|---|---|---|---|
| 31 … 8 | 7 | 6 … 4 | 3 | 2 … 0 | |
| 0 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 | 0 | 0x0 | Reset |

**UHCI_SINGLE_SEND_NUM** This field is used to specify the single_send mode. (R/W)

**UHCI_SINGLE_SEND_EN** Set this bit to enable single_send mode to send short packets. (R/W/SC)

**UHCI_ALWAYS_SEND_NUM** This field is used to specify the always_send mode. (R/W)

**UHCI_ALWAYS_SEND_EN** Set this bit to enable always_send mode to send short packets. (R/W)

### Register 26.40. UHCI_REG_Q0_WORD0_REG (0x0034)

| UHCI_SEND_Q0_WORD0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**UHCI_SEND_Q0_WORD0** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

### Register 26.41. UHCI_REG_Q0_WORD1_REG (0x0038)

| UHCI_SEND_Q0_WORD1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**UHCI_SEND_Q0_WORD1** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.42. UHCI_REG_Q1_WORD0_REG (0x003C)**

| UHCI_SEND_Q1_WORD0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**UHCI_SEND_Q1_WORD0** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.43. UHCI_REG_Q1_WORD1_REG (0x0040)**

| UHCI_SEND_Q1_WORD1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**UHCI_SEND_Q1_WORD1** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.44. UHCI_REG_Q2_WORD0_REG (0x0044)**

| UHCI_SEND_Q2_WORD0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**UHCI_SEND_Q2_WORD0** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.45. UHCI_REG_Q2_WORD1_REG (0x0048)**

| UHCI_SEND_Q2_WORD1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**UHCI_SEND_Q2_WORD1** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.46. UHCI_REG_Q3_WORD0_REG (0x004C)**

| UHCI_SEND_Q3_WORD0 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

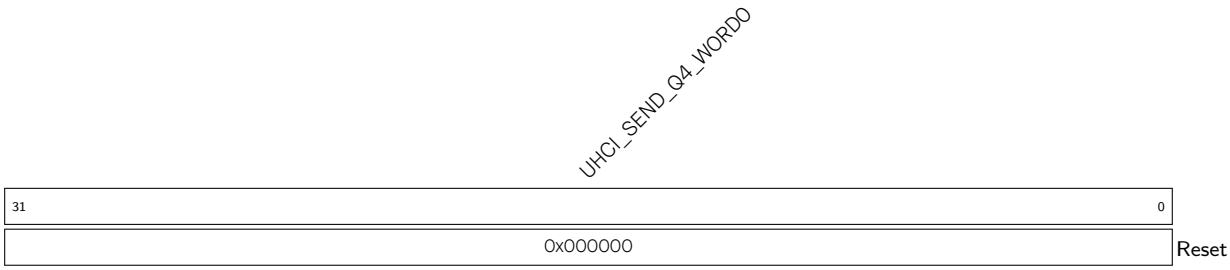
**UHCI_SEND_Q3_WORD0** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.47. UHCI_REG_Q3_WORD1_REG (0x0050)**

| UHCI_SEND_Q3_WORD1 | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

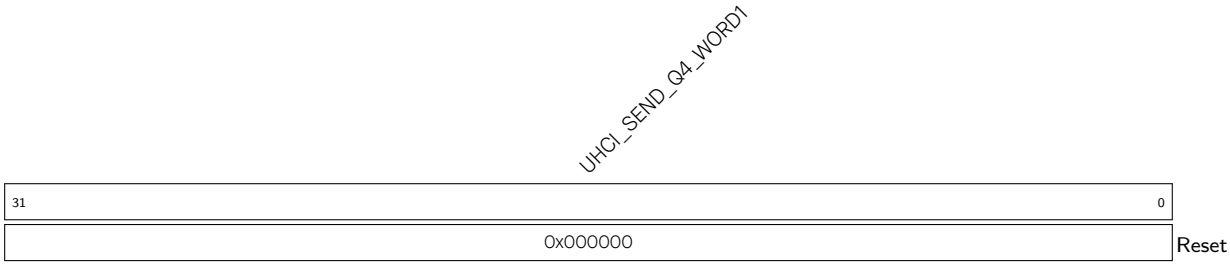
**UHCI_SEND_Q3_WORD1** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

Register 26.48. UHCI_REG_Q4_WORD0_REG (0x0054)



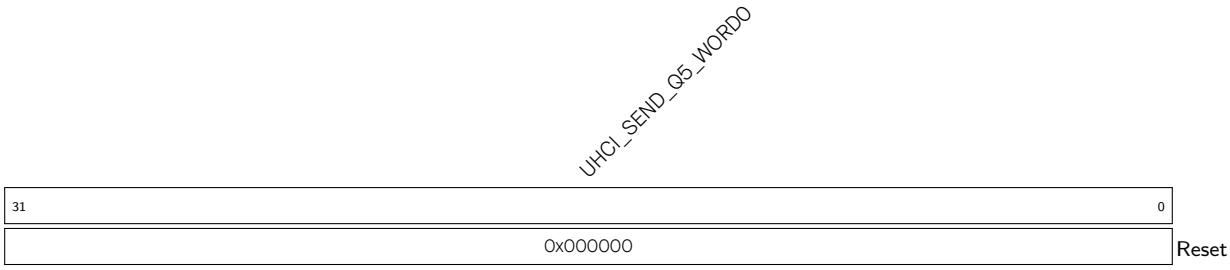

**UHCI_SEND_Q4_WORD0** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

Register 26.49. UHCI_REG_Q4_WORD1_REG (0x0058)




**UHCI_SEND_Q4_WORD1** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

Register 26.50. UHCI_REG_Q5_WORD0_REG (0x005C)




**UHCI_SEND_Q5_WORD0** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.51. UHCI_REG_Q5_WORD1_REG (0x0060)**

| 31 | 0 | |
|---|---|---|
| 0x000000 | | Reset |

**UHCI_SEND_Q5_WORD1** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.52. UHCI_REG_Q6_WORD0_REG (0x0064)**

| 31 | 0 | |
|---|---|---|
| 0x000000 | | Reset |

**UHCI_SEND_Q6_WORD0** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

**Register 26.53. UHCI_REG_Q6_WORD1_REG (0x0068)**

| 31 | 0 | |
|---|---|---|
| 0x000000 | | Reset |

**UHCI_SEND_Q6_WORD1** This register is used as a quick_sent register when mode is specified by UHCI_ALWAYS_SEND_NUM or UHCI_SINGLE_SEND_NUM. (R/W)

### Register 26.54. UHCI_ESC_CONF0_REG (0x006C)

| (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 | 0xdc | 0xdb | 0xc0 | Reset |

**UHCI_SEPER_CHAR** This field is used to define separators to encode data packets. The default value is 0xC0. (R/W)

**UHCI_SEPER_ESC_CHAR0** This field is used to define the first character of SLIP escape sequence. The default value is 0xDB. (R/W)

**UHCI_SEPER_ESC_CHAR1** This field is used to define the second character of SLIP escape sequence. The default value is 0xDC. (R/W)

### Register 26.55. UHCI_ESC_CONF1_REG (0x0070)

| (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 0 0 0 | 0xdd | 0xdb | 0xdb | Reset |

**UHCI_ESC_SEQ0** This field is used to define a character that need to be encoded. The default value is 0xDB that used as the first character of SLIP escape sequence. (R/W)

**UHCI_ESC_SEQ0_CHAR0** This field is used to define the first character of SLIP escape sequence. The default value is 0xDB. (R/W)

**UHCI_ESC_SEQ0_CHAR1** This field is used to define the second character of SLIP escape sequence. The default value is 0xDD. (R/W)

**Register 26.56. UHCI_ESC_CONF2_REG (0x0074)**

| (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 0 0 0 | 0xde | 0xdb | 0x11 | Reset |

**UHCI_ESC_SEQ1** This field is used to define a character that need to be encoded. The default value is 0x11 that used as a flow control character. (R/W)

**UHCI_ESC_SEQ1_CHAR0** This field is used to define the first character of SLIP escape sequence. The default value is 0xDB. (R/W)

**UHCI_ESC_SEQ1_CHAR1** This field is used to define the second character of SLIP escape sequence. The default value is 0xDE. (R/W)

**Register 26.57. UHCI_ESC_CONF3_REG (0x0078)**

| (reserved) | UHCI_ESC_SEQ2_CHAR1 | UHCI_ESC_SEQ2_CHAR0 | UHCI_ESC_SEQ2 | |
|---|---|---|---|---|
| 31 — 24 | 23 — 16 | 15 — 8 | 7 — 0 | |
| 0 0 0 0 0 0 0 0 | 0xdf | 0xdb | 0x13 | Reset |

**UHCI_ESC_SEQ2** This field is used to define a character that need to be decoded. The default value is 0x13 that used as a flow control character. (R/W)

**UHCI_ESC_SEQ2_CHAR0** This field is used to define the first character of SLIP escape sequence. The default value is 0xDB. (R/W)

**UHCI_ESC_SEQ2_CHAR1** This field is used to define the second character of SLIP escape sequence. The default value is 0xDF. (R/W)

**Register 26.58. UHCI_PKT_THRES_REG (0x007C)**

| (reserved) | UHCI_PKT_THRS | |
|---|---|---|
| 31 … 13 | 12 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x80 | Reset |

**UHCI_PKT_THRS** This field is used to configure the maximum value of the packet length when UHCI_HEAD_EN is 0. (R/W)

**Register 26.59. UHCI_INT_RAW_REG (0x0004)**

| (reserved) | UHCI_APP_CTRL1_INT_RAW | UHCI_APP_CTRL0_INT_RAW | UHCI_OUT_EOF_INT_RAW | UHCI_SEND_A_REG_Q_INT_RAW | UHCI_SEND_S_REG_Q_INT_RAW | UHCI_TX_HUNG_INT_RAW | UHCI_RX_HUNG_INT_RAW | UHCI_TX_START_INT_RAW | UHCI_RX_START_INT_RAW | |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 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 | Reset |

**UHCI_RX_START_INT_RAW** This is the interrupt raw bit for UHCI_RX_START_INT interrupt. The interrupt is triggered when a separator has been sent. (R/WTC/SS)

**UHCI_TX_START_INT_RAW** This is the interrupt raw bit for UHCI_TX_START_INT interrupt. The interrupt is triggered when UHCI detects a separator. (R/WTC/SS)

**UHCI_RX_HUNG_INT_RAW** This is the interrupt raw bit for UHCI_RX_HUNG_INT interrupt. The interrupt is triggered when UHCI takes more time to receive data than configure value. (R/WTC/SS)

**UHCI_TX_HUNG_INT_RAW** This is the interrupt raw bit for UHCI_TX_HUNG_INT interrupt. The interrupt is triggered when UHCI takes more time to read data from RAM than the configured value. (R/WTC/SS)

**UHCI_SEND_S_REG_Q_INT_RAW** This is the interrupt raw bit for UHCI_SEND_S_REG_Q_INT interrupt. The interrupt is triggered when UHCI has sent out a short packet using single_send mode. (R/WTC/SS)

**UHCI_SEND_A_REG_Q_INT_RAW** This is the interrupt raw bit for UHCI_SEND_A_REG_Q_INT interrupt. The interrupt is triggered when UHCI has sent out a short packet using always_send mode. (R/WTC/SS)

**UHCI_OUT_EOF_INT_RAW** This is the interrupt raw bit for UHCI_OUT_EOF_INT interrupt. The interrupt is triggered when there are some errors in EOF in the transmit descriptors. (R/WTC/SS)

**UHCI_APP_CTRL0_INT_RAW** This is the interrupt raw bit for UHCI_APP_CTRL0_INT interrupt. The interrupt is triggered when UHCI_APP_CTRL0_IN_SET is set. (R/W)

**UHCI_APP_CTRL1_INT_RAW** This is the interrupt raw bit for UHCI_APP_CTRL1_INT interrupt. The interrupt is triggered when UHCI_APP_CTRL1_IN_SET is set. (R/W)

**Register 26.60. UHCI_INT_ST_REG (0x0008)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | UHCI_APP_CTRL1_INT_ST | UHCI_APP_CTRL0_INT_ST | UHCI_OUTLINK_EOF_ERR_INT_ST | UHCI_SEND_A_REG_Q_INT_ST | UHCI_SEND_S_REG_Q_INT_ST | UHCI_TX_HUNG_INT_ST | UHCI_RX_HUNG_INT_ST | UHCI_TX_START_INT_ST | UHCI_RX_START_INT_ST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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_ST** This is the masked interrupt bit for UHCI_RX_START_INT interrupt when UHCI_RX_START_INT_ENA is set to 1. (RO)

**UHCI_TX_START_INT_ST** This is the masked interrupt bit for UHCI_TX_START_INT interrupt when UHCI_TX_START_INT_ENA is set to 1. (RO)

**UHCI_RX_HUNG_INT_ST** This is the masked interrupt bit for UHCI_RX_HUNG_INT interrupt when UHCI_RX_HUNG_INT_ENA is set to 1. (RO)

**UHCI_TX_HUNG_INT_ST** This is the masked interrupt bit for UHCI_TX_HUNG_INT interrupt when UHCI_TX_HUNG_INT_ENA is set to 1. (RO)

**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_OUTLINK_EOF_ERR_INT_ST** This is the masked interrupt bit for UHCI_OUTLINK_EOF_ERR_INT interrupt when UHCI_OUTLINK_EOF_ERR_INT_ENA is set to 1. (RO)

**UHCI_APP_CTRL0_INT_ST** This is the masked interrupt bit for UHCI_APP_CTRL0_INT interrupt when UHCI_APP_CTRL0_INT_ENA is set to 1. (RO)

**UHCI_APP_CTRL1_INT_ST** This is the masked interrupt bit for UHCI_APP_CTRL1_INT interrupt when UHCI_APP_CTRL1_INT_ENA is set to 1. (RO)

**Register 26.61. UHCI_INT_ENA_REG (0x000C)**

| 31 | | | | | | | | | | | | | | | | | | | | | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | UHCI_APP_CTRL1_INT_ENA | UHCI_APP_CTRL0_INT_ENA | UHCI_OUTLINK_EOF_ERR_INT_ENA | UHCI_SEND_A_REG_Q_INT_ENA | UHCI_SEND_S_REG_Q_INT_ENA | UHCI_TX_HUNG_INT_ENA | UHCI_RX_HUNG_INT_ENA | UHCI_TX_START_INT_ENA | UHCI_RX_START_INT_ENA | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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_SEND_S_REG_Q_INT_ENA** This is the interrupt enable bit for UHCI_SEND_S_REG_Q_INT interrupt. (R/W)

**UHCI_SEND_A_REG_Q_INT_ENA** This is the interrupt enable bit for UHCI_SEND_A_REG_Q_INT interrupt. (R/W)

**UHCI_OUTLINK_EOF_ERR_INT_ENA** This is the interrupt enable bit for UHCI_OUTLINK_EOF_ERR_INT interrupt. (R/W)

**UHCI_APP_CTRL0_INT_ENA** This is the interrupt enable bit for UHCI_APP_CTRL0_INT interrupt. (R/W)

**UHCI_APP_CTRL1_INT_ENA** This is the interrupt enable bit for UHCI_APP_CTRL1_INT interrupt. (R/W)

**Register 26.62. UHCI_INT_CLR_REG (0x0010)**

| 31 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | UHCI_APP_CTRL1_INT_CLR | UHCI_APP_CTRL0_INT_CLR | UHCI_OUTLINK_EOF_ERR_INT_CLR | UHCI_SEND_A_REG_Q_INT_CLR | UHCI_SEND_S_REG_Q_INT_CLR | UHCI_TX_HUNG_INT_CLR | UHCI_RX_HUNG_INT_CLR | UHCI_TX_START_INT_CLR | UHCI_RX_START_INT_CLR | |
| 0 0 0 0 0 0 0 0 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. (WT)

**UHCI_TX_START_INT_CLR** Set this bit to clear UHCI_TX_START_INT interrupt. (WT)

**UHCI_RX_HUNG_INT_CLR** Set this bit to clear UHCI_RX_HUNG_INT interrupt. (WT)

**UHCI_TX_HUNG_INT_CLR** Set this bit to clear UHCI_TX_HUNG_INT interrupt. (WT)

**UHCI_SEND_S_REG_Q_INT_CLR** Set this bit to clear UHCI_SEND_S_REG_Q_INT interrupt. (WT)

**UHCI_SEND_A_REG_Q_INT_CLR** Set this bit to clear UHCI_SEND_A_REG_Q_INT interrupt. (WT)

**UHCI_OUTLINK_EOF_ERR_INT_CLR** Set this bit to clear UHCI_OUTLINK_EOF_ERR_INT interrupt. (WT)

**UHCI_APP_CTRL0_INT_CLR** Set this bit to clear UHCI_APP_CTRL0_INT interrupt. (WT)

**UHCI_APP_CTRL1_INT_CLR** Set this bit to clear UHCI_APP_CTRL1_INT interrupt. (WT)

**Register 26.63. UHCI_STATE0_REG (0x0018)**

| 31 | 6 | 5 | 3 | 2 | 0 | |
|---|---|---|---|---|---|---|
| (reserved) | | UHCI_DECODE_STATE | | UHCI_RX_ERR_CAUSE | | |
| 0 0 0 0 0 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_ERR_CAUSE** This field indicates the error type when DMA has received a packet with error. 3'b001: Checksum error in the HCI packet; 3'b010: Sequence number error in the HCI packet; 3'b011: CRC bit error in the HCI packet; 3'b100: 0xC0 is found but the received the HCI packet is not end; 3'b101: 0xC0 is not found when the HCI packet has been received; 3'b110: CRC check error. (RO)

**UHCI_DECODE_STATE** UHCI decoder status. (RO)

### Register 26.64. UHCI_STATE1_REG (0x001C)

| (reserved) | UHCI_ENCODE_STATE | |
|---|---|---|
| 31 … 3 | 2 … 0 | |
| 0 0 0 0 0 0 0 0 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_ENCODE_STATE** UHCI encoder status. (RO)

### Register 26.65. UHCI_RX_HEAD_REG (0x002C)

| UHCI_RX_HEAD | |
|---|---|
| 31 … 0 | |
| 0x000000 | Reset |

**UHCI_RX_HEAD** This register stores the header of the current received packet. (RO)

### Register 26.66. UHCI_DATE_REG (0x0080)

| UHCI_DATE | |
|---|---|
| 31 … 0 | |
| 0x2007170 | Reset |

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

# Chapter 27

# SPI Controller (SPI)

## 27.1 Overview

The Serial Peripheral Interface (SPI) is a synchronous serial interface useful for communication with external peripherals. The ESP32-C3 chip integrates three SPI controllers:

- SPI0,
- SPI1,
- General Purpose SPI2 (GP-SPI2).

SPI0 and SPI1 controllers are primarily reserved for internal use. This chapter mainly focuses on the GP-SPI2 controller.

## 27.2 Glossary

To better illustrate the functions of GP-SPI2, the following terms are used in this chapter.

| | |
|---|---|
| **Master Mode** | GP-SPI2 acts as an SPI master and initiates SPI transactions. |
| **Slave Mode** | GP-SPI2 acts as an SPI slave and transfers data with its master when its CS is asserted. |
| **MISO** | Master in, slave out, data transmission from a slave to a master. |
| **MOSI** | Master out, slave in, data transmission from a master to a slave |
| **Transaction** | One instance of a master asserting a CS line, transferring data to and from a slave, and de-asserting the CS line. Transactions are atomic, which means they can never be interrupted by another transaction. |
| **SPI Transfer** | The whole process of an SPI master exchanges data with a slave. One SPI transfer consists of one or more SPI transactions. |
| **Single Transfer** | An SPI transfer consists of only one transaction. |
| **CPU-Controlled Transfer** | A data transfer happens between CPU buffer SPI_W0_REG ~ SPI_W15_REG and SPI peripheral. |
| **DMA-Controlled Transfer** | A data transfer happens between DMA and SPI peripheral, controlled by DMA engine. |
| **Configurable Segmented Transfer** | A data transfer controlled by DMA in SPI master mode. Such transfer consists of multiple transactions (segments), and each of transactions can be configured independently. |
| **Slave Segmented Transfer** | A data transfer controlled by DMA in SPI slave mode. Such transfer consists of multiple transactions (segments). |

**Full-duplex** The sending line and receiving line between the master and the slave are independent. Sending data and receiving data happen at the same time.

**Half-duplex** Only one side, the master or the slave, sends data first, and the other side receives data. Sending data and receiving data can not happen at the same time.

**4-line full-duplex** 4-line here means: clock line, CS line, and two data lines. The two data lines can be used to send or receive data simultaneously.

**4-line half-duplex** 4-line here means: clock line, CS line, and two data lines. The two data lines can not be used simultaneously.

**3-line half-duplex** 3-line here means: clock line, CS line, and one data line. The data line is used to transmit or receive data.

**1-bit SPI** In one clock cycle, one bit can be transferred.

**(2-bit) Dual SPI** In one clock cycle, two bits can be transferred.

**Dual Output Read** A data mode of Dual SPI. In one clock cycle, one bit of a command, or one bit of an address, or two bits of data can be transferred.

**Dual I/O Read** Another data mode of Dual SPI. In one clock cycle, one bit of a command, or two bits of an address, or two bits of data can be transferred.

**(4-bit) Quad SPI** In one clock cycle, four bits can be transferred.

**Quad Output Read** A data mode of Quad SPI. In one clock cycle, one bit of a command, or one bit of an address, or four bits of data can be transferred.

**Quad I/O Read** Another data mode of Quad SPI. In one clock cycle, one bit of a command, or four bits of an address, or four bits of data can be transferred.

**QPI** In one clock cycle, four bits of a command, or four bits of an address, or four bits of data can be transferred.

## 27.3 Features

Some of the key features of GP-SPI2 are:

- Master and slave modes
- Half- and full-duplex communications
- CPU- and DMA-controlled transfers
- Various data modes:
  - 1-bit SPI mode
  - 2-bit Dual SPI mode
  - 4-bit Quad SPI mode
  - QPI mode
- Configurable module clock frequency:

  - Master: up to 80 MHz
  - Slave: up to 60 MHz
- Configurable data length:
  - CPU-controlled transfer in master mode or in slave mode: 1 ~ 64 B
  - DMA-controlled single transfer in master mode: 1 ~ 32 KB
  - DMA-controlled configurable segmented transfer in master mode: data length is unlimited
  - DMA-controlled single transfer or segmented transfer in slave mode: data length is unlimited
- Configurable bit read/write order
- Independent interrupts for CPU-controlled transfer and DMA-controlled transfer
- Configurable clock polarity and phase
- Four SPI clock modes: mode 0 ~ mode 3
- Six CS lines in master mode: CS0 ~ CS5
- Able to communicate with SPI devices, such as a sensor, a screen controller, as well as a flash or RAM chip

## 27.4 Architectural Overview

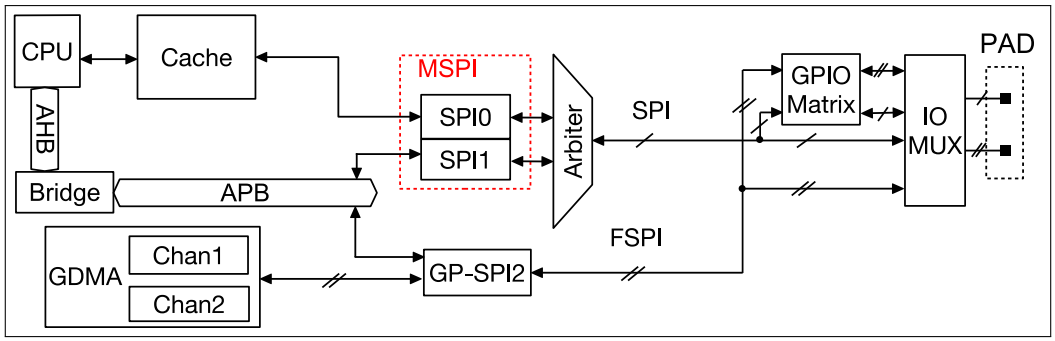



Figure 27.4-1. SPI Module Overview

Figure 27.4-1 shows an overview of SPI module. GP-SPI2 exchanges data with SPI devices by the following ways:

- CPU-controlled transfer: CPU <-> GP-SPI2 <-> SPI devices
- DMA-controlled transfer: GDMA <-> GP-SPI2 <-> SPI devices

The signals for GP-SPI2 are prefixed with "FSPI" (Fast SPI). FSPI bus signals are routed to GPIO pins via either GPIO matrix or IO MUX. For more information, see Chapter *5 IO MUX and GPIO Matrix (GPIO, IO MUX)*.

## 27.5 Functional Description

### 27.5.1 Data Modes

GP-SPI2 can be configured as either a master or a slave to communicate with other SPI devices in the following data modes, see Table 27.5-1.

Table 27.5-1. Data Modes Supported by GP-SPI2

| Supported Mode | | CMD State | Address State | Data State |
|---|---|---|---|---|
| 1-bit SPI | | 1-bit | 1-bit | 1-bit |
| Dual SPI | Dual Output Read | 1-bit | 1-bit | 2-bit |
| | Dual I/O Read | 1-bit | 2-bit | 2-bit |
| Quad SPI | Quad Output Read | 1-bit | 1-bit | 4-bit |
| | Quad I/O Read | 1-bit | 4-bit | 4-bit |
| QPI | | 4-bit | 4-bit | 4-bit |

For the states can be used in

- master mode, see Section 27.5.8.
- slave mode, see Section 27.5.9.

### 27.5.2 FSPI Bus Signal Mapping

The mapping of FSPI bus signals and the functional description of the signals are shown in Table 27.5-2 and in Table 27.5-3, respectively. The signals in one line in Table 27.5-2 corresponds to each other. For example, the signal FSPID is connected to MOSI in GP-SPI2 full-duplex communication, and FSPIQ to MISO. You can take Figure 27.5-6 as an example.

Table 27.5-2. Mapping of FSPI Bus Signals

| Standard SPI Protocol | | Extended SPI Protocol |
|---|---|---|
| Full-Duplex SPI Signal | Half-Duplex SPI Signal | FSPI Bus Signal |
| MOSI | MOSI | FSPID |
| MISO | (MISO) | FSPIQ |
| CS | CS | FSPICS0 ~ 5 |
| CLK | CLK | FSPICLK |
| — | — | FSPIWP |
| — | — | FSPIHD |

Table 27.5-3. Functional Description of FSPI Bus Signals

| FSPI Bus Signal | Function |
|---|---|
| FSPID | MOSI/SIO0 (serial data input and output, bit0) |
| FSPIQ | MISO/SIO1 (serial data input and output, bit1) |
| FSPIWP | SIO2 (serial data input and output, bit2) |
| FSPIHD | SIO3 (serial data input and output, bit3) |

| | |
|---|---|
| FSPICLK | Input and output clock in master/slave mode |
| FSPICS0 | Input and output CS signal in master/slave mode |
| FSPICS1 ~ 5 | Output CS signal in master mode |

Figure 27.5-4 shows the signals used in various SPI modes.

Table 27.5-4. Signals Used in Various SPI Modes

| FSPI Signal | Master Mode | | | | | | Slave Mode | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1-bit SPI | | | 2-bit Dual SPI | 4-bit Quad SPI | QPI | 1-bit SPI | | | 2-bit Dual SPI | 4-bit Quad SPI | QPI |
| | FD[1] | 3-line HD[2] | 4-line HD | | | | FD | 3-line HD | 4-line HD | | | |
| FSPICLK | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| FSPICS0 | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| FSPICS1 | Y | Y | Y | Y | Y | Y | | | | | | |
| FSPICS2 | Y | Y | Y | Y | Y | Y | | | | | | |
| FSPICS3 | Y | Y | Y | Y | Y | Y | | | | | | |
| FSPICS4 | Y | Y | Y | Y | Y | Y | | | | | | |
| FSPICS5 | Y | Y | Y | Y | Y | Y | | | | | | |
| FSPID | Y | Y | (Y)[3] | Y[4] | Y[5] | Y | Y | Y | (Y)[6] | Y[7] | Y[8] | Y |
| FSPIQ | Y | | (Y)[3] | Y[4] | Y[5] | Y | Y | | (Y)[6] | Y[7] | Y[8] | Y |
| FSPIWP | | | | | Y[5] | Y | | | | | Y[8] | Y |
| FSPIHD | | | | | Y[5] | Y | | | | | Y[8] | Y |

[1] FD: full-duplex

[2] HD: half-duplex

[3] Only one of the two signals is used at a time.

[4] The two signals are used in parallel.

[5] The four signals are used in parallel.

[6] Only one of the two signals is used at a time.

[7] The two signals are used in parallel.

[8] The four signals are used in parallel.







GoBack

### 27.5.3 Bit Read/Write Order Control

In master mode:

- The bit order of the command, address and data sent by the GP-SPI2 master is controlled by SPI_WR_BIT_ORDER.
- The bit order of the data received by the master is controlled by SPI_RD_BIT_ORDER.

In slave mode:

- The bit order of the data sent by the GP-SPI2 slave is controlled by SPI_WR_BIT_ORDER.
- The bit order of the command, address and data received by the slave is controlled by SPI_RD_BIT_ORDER.

Table 27.5-5 shows the function of SPI_RD/WR_BIT_ORDER.

Table 27.5-5. Bit Order Control in GP-SPI2 Master and Slave Modes

| Bit Mode | FSPI Bus Data | SPI_RD/WR_BIT_ORDER = 0 (MSB) | SPI_RD/WR_BIT_ORDER = 1 (LSB) |
|---|---|---|---|
| 1-bit mode | FSPID or FSPIQ | B7→B6→B5→B4→B3→B2→B1→B0 | B0→B1→B2→B3→B4→B5→B6→B7 |
| 2-bit mode | FSPIQ | B7→B5→B3→B1 | B1→B3→B5→B7 |
| | FSPID | B6→B4→B2→B0 | B0→B2→B4→B6 |
| 4-bit mode | FSPIHD | B7→B3 | B3→B7 |
| | FSPIWP | B6→B2 | B2→B6 |
| | FSPIQ | B5→B1 | B1→B5 |
| | FSPID | B4→B0 | B0→B4 |

### 27.5.4 Transfer Modes

GP-SPI2 supports the following transfers when working as a master or a slave.

Table 27.5-6. Supported Transfers in Master and Slave Modes

| Mode | | CPU-Controlled Single Transfer | DMA-Controlled Single Transfer | DMA-Controlled Configurable Segmented Transfer | DMA-Controlled Slave Segmented Transfer |
|---|---|---|---|---|---|
| Master | Full-Duplex | Y | Y | Y | – |
| | Half-Duplex | Y | Y | Y | – |
| Slave | Full-Duplex | Y | Y | – | Y |
| | Half-Duplex | Y | Y | – | Y |

The following sections provide detailed information about the transfer modes listed in the table above.

### 27.5.5 CPU-Controlled Data Transfer

GP-SPI2 provides 16 x 32-bit data buffers, i.e., SPI_W0_REG ~ SPI_W15_REG, see Figure 27.5-1. CPU-controlled transfer indicates the transfer, in which the data to send is from GP-SPI2 data buffer and the received data is stored to GP-SPI2 data buffer. In such transfer, every single transaction needs to be triggered by the CPU, after its related registers are configured. For such reason, the CPU-controlled transfer is always

single transfers (consisting of only one transaction). CPU-controlled mode supports full-duplex communication and half-duplex communication.

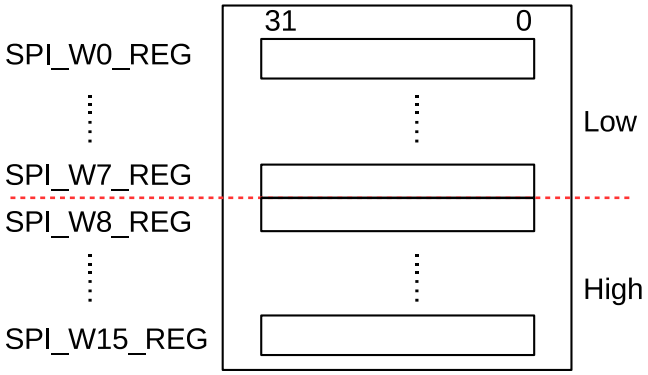



Figure 27.5-1. Data Buffer Used in CPU-Controlled Transfer

#### 27.5.5.1 CPU-Controlled Master Mode

In a CPU-controlled master full-duplex or half-duplex transfer, the RX or TX data is saved to or sent from SPI_W0_REG ~ SPI_W15_REG. The bits SPI_USR_MOSI_HIGHPART and SPI_USR_MISO_HIGHPART control which buffers are used, see the list below.

- TX data
  - When SPI_USR_MOSI_HIGHPART is cleared, i.e., high part mode is disabled, TX data is from SPI_W0_REG ~ SPI_W15_REG and the data address is incremented by 1 on each byte transferred. If the data byte length is larger than 64, the data in SPI_W0_REG[7:0] ~ SPI_W15_REG[31:24] may be sent more than once. For instance, if 66 bytes (byte0 ~ byte65) need to be sent, then the address of byte65 is the result of (65 % 64 = 1), i.e., byte65 is from SPI_W0_REG[15:8], and byte64 is from SPI_W0_REG[7:0]. In this case, the content of SPI_W0_REG[15:0] may be sent more than once.
  - When SPI_USR_MOSI_HIGHPART is set, i.e., high part mode is enabled, TX data is from SPI_W8_REG ~ SPI_W15_REG and the data address is incremented by 1 on each byte transferred. If the data byte length is larger than 32, the data in SPI_W8_REG[7:0] ~ SPI_W15_REG[31:24] may be sent more than once.
- RX data
  - When SPI_USR_MISO_HIGHPART is cleared, i.e., high part mode is disabled, RX data is saved to SPI_W0_REG ~ SPI_W15_REG, and the data address is incremented by 1 on each byte transferred. If the data byte length is larger than 64, the data in SPI_W0_REG[7:0] ~ SPI_W15_REG[31:24] may be overwritten. For instance, when 66 bytes (byte0 ~ byte65) are received, byte65 and byte64 are stored to the addresses of (65 % 64 = 1) and (64 % 64 = 0), i.e., SPI_W0_REG[15:8] and SPI_W0_REG[7:0]. In this case, the content of SPI_W0_REG[15:0] may be overwritten.
  - When SPI_USR_MISO_HIGHPART is set, i.e., high part mode is enabled, the RX data is saved to SPI_W8_REG ~ SPI_W15_REG, and the data address is incremented by 1 on each byte transferred. If the data byte length is larger than 32, the content of SPI_W8_REG ~ SPI_W15_REG may be overwritten.

**Note:**

- TX/RX data address mentioned above both are byte-addressable. Address 0 stands for SPI_W0_REG[7:0], and Address 1 for SPI_W0_REG[15:8], and so on. The largest address is SPI_W15_REG[31:24].
- To avoid any possible error in TX/RX data, such as TX data being sent more than once or RX data being overwritten, please make sure the registers are configured correctly.

#### 27.5.5.2 CPU-Controlled Slave Mode

In a CPU-controlled slave full-duplex or half-duplex transfer, the RX data or TX data is saved to or sent from SPI_W0_REG ~ SPI_W15_REG, which are byte-addressable.

- In full-duplex communication, the address of SPI_W0_REG ~ SPI_W15_REG starts from 0 and is incremented by 1 on each byte transferred. If the data address is larger than 63, the content of SPI_W15_REG[31:24] is overwritten.
- In half-duplex communication, the ADDR value in transmission format is the start address of the RX or TX data, corresponding to the registers SPI_W0_REG ~ SPI_W15_REG. The RX or TX address is incremented by 1 on each byte transferred. If the address is larger than 63 (the highest byte address, i.e. SPI_W15_REG[31:24]), the address of overflowing data is always 63 and only the content of SPI_W15_REG[31:24] is overwritten.

According to your applications, the registers SPI_W0_REG ~ SPI_W15_REG can be used as:

- data buffers only
- data buffers and status buffers
- status buffers only

### 27.5.6 DMA-Controlled Data Transfer

DMA-controlled transfer refers to the transfer, in which GDMA RX module receives data and GDMA TX module sends data. This transfer is supported both in master mode and in slave mode.

A DMA-controlled transfer can be

- a single transfer, consisting of only one transaction. GP-SPI2 supports this transfer both in master and slave modes.
- a configurable segmented transfer, consisting of several transactions (segments). GP-SPI2 supports this transfer only in master mode. For more information, see Section 27.5.8.5.
- a slave segmented transfer, consisting of several transactions (segments). GP-SPI2 supports this transfer only in slave mode. For more information, see Section 27.5.9.3.

A DMA-controlled transfer only needs to be triggered once by CPU. When such transfer is triggered, data is transferred by the GDMA engine from or to the DMA-linked memory, without CPU operation.

DMA-controlled mode supports full-duplex communication, half-duplex communication and functions described in Section 27.5.8 and Section 27.5.9. Meanwhile, the GDMA RX module is independent from the GDMA TX module, which means that there are four kinds of full-duplex communications:

- Data is received in DMA-controlled mode and sent in DMA-controlled mode.

- Data is received in DMA-controlled mode but sent in CPU-controlled mode.
- Data is received in CPU-controlled mode but sent in DMA-controlled mode.
- Data is received in CPU-controlled mode and sent in CPU-controlled mode.

#### 27.5.6.1 GDMA Configuration

- Select a GDMA channel*n*, and configure a GDMA TX/RX descriptor, see Chapter 2 *GDMA Controller (GDMA)*.
- Set the bit GDMA_INLINK_START_CH*n* or GDMA_OUTLINK_START_CH*n* to start GDMA RX/TX engine.
- Before all the GDMA TX buffer is used or the GDMA TX engine is reset, if GDMA_OUTLINK_RESTART_CH*n* is set, a new TX buffer will be added to the end of the last TX buffer in use.
- GDMA RX buffer is linked in the same way as the GDMA TX buffer, by setting GDMA_INLINK_START_CH*n* or GDMA_INLINK_RESTART_CH*n*.
- The TX and RX data lengths are determined by the configured GDMA TX and RX buffer respectively, both of which are 0 ~ 32 KB.
- Initialize GDMA inlink and outlink before GDMA starts. The bits SPI_DMA_RX_ENA and SPI_DMA_TX_ENA in register SPI_DMA_CONF_REG should be set, otherwise the read/write data will be stored to/sent from the registers SPI_W0_REG ~ SPI_W15_REG.

In master mode, if GDMA_IN_SUC_EOF_CH*n*_INT_ENA is set, then the interrupt GDMA_IN_SUC_EOF_CH*n*_INT will be triggered when one single transfer or one configurable segmented transfer is finished.

The only difference between DMA-controlled transfers in master mode and in slave mode is on the GDMA RX control:

- When the bit SPI_RX_EOF_EN is cleared, a GDMA_IN_SUC_EOF_CH*n*_INT interrupt may be generated after the CS is pulled high once:
  - In a slave single transfer, if SPI_DMA_SLV_SEG_TRANS_EN is cleared and GDMA_IN_SUC_EOF_CH*n*_INT _ENA is set, a GDMA_IN_SUC_EOF_CH*n*_INT interrupt will be triggered once the single transfer is done.
  - In a slave segmented transfer, if both SPI_DMA_SLV_SEG_TRANS_EN and GDMA_IN_SUC_EOF_CH*n*_ INT_ENA are set, a GDMA_IN_SUC_EOF_CH*n*_INT interrupt also is triggered once the command (CMD7 or End_SEG_TRANS) is received correctly.
- When the bit SPI_RX_EOF_EN is set, the generation of GDMA_IN_SUC_EOF_CH*n*_INT also depends on the length of transferred data.
  - In a slave single transfer, if SPI_DMA_SLV_SEG_TRANS_EN is cleared and GDMA_IN_SUC_EOF_CH*n*_ INT_ENA is set, a GDMA_IN_SUC_EOF_CH*n*_INT interrupt will be generated once the single transfer is done or the length of GDMA RX received data is equal to (SPI_MS_DATA_BITLEN + 1).
  - In a slave segmented transfer, if SPI_DMA_SLV_SEG_TRANS_EN is set, a GDMA_IN_SUC_EOF_CH*n*_INT interrupt will be generated once the command (CMD7 or

End_SEG_TRANS) is received correctly or the length of GDMA RX received data is equal to (SPI_MS_DATA_BITLEN + 1).

#### 27.5.6.2 GDMA TX/RX Buffer Length Control

It is recommended that the length of configured GDMA TX/RX buffer is equal to the length of real transferred data.

- If the length of configured GDMA TX buffer is shorter than that of real transferred data, the extra data will be the same as the last transferred data. SPI_OUTFIFO_EMPTY_ERR_INT and GDMA_OUT_EOF_CH*n*_INT are triggered.
- If the length of configured GDMA TX buffer is longer than the that of real transferred data, the TX buffer is not fully used, and the remaining buffer is available for following transaction even if a new TX buffer is linked later. Please keep it in mind. Or save the unused data and reset DMA.
- If the length of configured GDMA RX buffer is shorter than that of real transferred data, the extra data will be lost. The interrupts SPI_INFIFO_FULL_ERR_INT and SPI_TRANS_DONE_INT are triggered. But GDMA_IN_SUC_EOF_CH*n*_INT interrupt is not generated.
- If the length of configured GDMA RX buffer is longer than that of real transferred data, the RX buffer is not fully used, and the remaining buffer is discarded. In the following transaction, a new linked buffer will be used directly.

### 27.5.7 Data Flow Control in GP-SPI2 Master and Slave Modes

CPU-controlled and DMA-controlled transfers are supported in GP-SPI2 master and slave modes. CPU-controlled transfer means that data transfers between registers SPI_W0_REG ~ SPI_W15_REG and the SPI device. DMA-controlled transfer means that data transfers between the configured GDMA TX/RX buffer and the SPI device. To select between the two transfer modes, configure SPI_DMA_RX_ENA and SPI_DMA_TX_ENA before the transfer starts.

#### 27.5.7.1 GP-SPI2 Functional Blocks

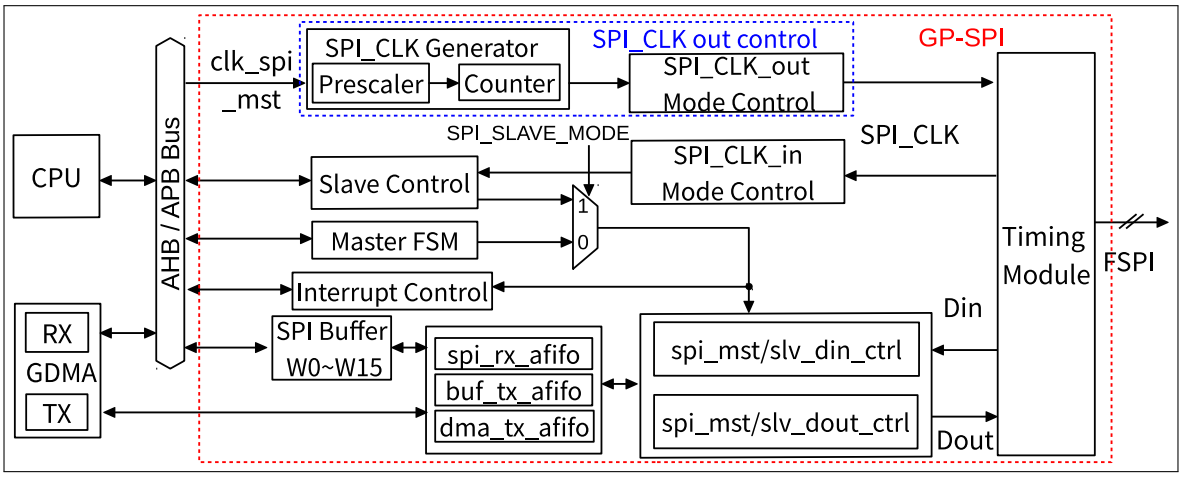



Figure 27.5-2. GP-SPI2 Block Diagram

Figure 27.5-2 shows main functional blocks in GP-SPI2, including:

- Master FSM: all the features, supported in GP-SPI2 master mode, are controlled by this state machine together with register configuration.
- SPI Buffer: SPI_W0_REG ~ SPI_W15_REG, see Figure 27.5-1. The data transferred in CPU-controlled mode is prepared in this buffer.
- Timing Module: capture data on FSPI bus.
- spi_mst/slv_din/dout_ctrl: convert the TX/RX data into bytes.
- spi_rx_afifo: store the received data.
- buf_tx_afifo: store the data to send.
- dma_tx_afifo: store the data from GDMA.
- clk_spi_mst: this clock is the module clock of GP-SPI2 and derived from PLL_CLK. It is used in GP-SPI2 master mode, to generate SPI_CLK signal for data transfer and for slaves.
- SPI_CLK Generator: generate SPI_CLK by dividing clk_spi_mst. The divider is determined by SPI_CLKCNT_N and SPI_CLKDIV_PRE.
- SPI_CLK_out Mode Control: output the SPI_CLK signal for data transfer and for slaves.
- SPI_CLK_in Mode Control: capture the SPI_CLK signal from SPI master when GP-SPI2 works as a slave.

#### 27.5.7.2 Data Flow Control in Master Mode

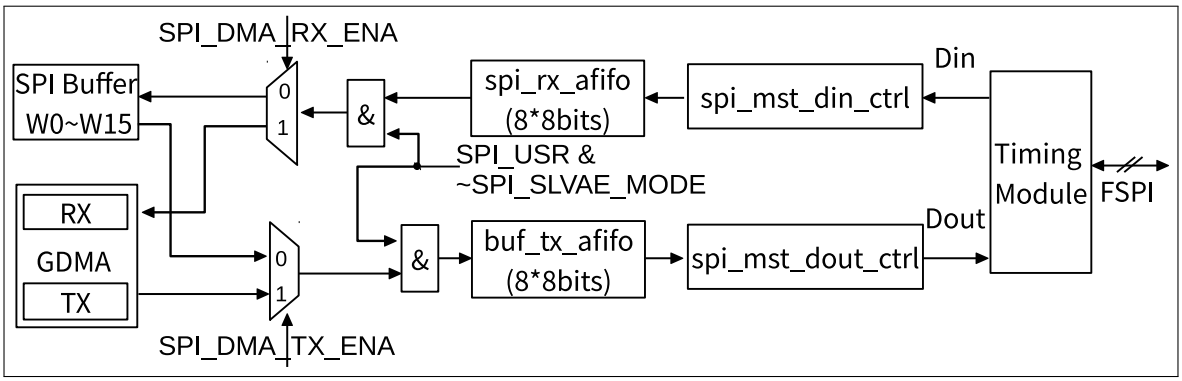



Figure 27.5-3. Data Flow Control in GP-SPI2 Master Mode

Figure 27.5-3 shows the data flow of GP-SPI2 in master mode. Its control logic is as follows:

- RX data: data in FSPI bus is captured by Timing Module, converted in units of bytes by spi_mst_din_ctrl module, and then stored in corresponding addresses according to the transfer modes.
  - CPU-controlled transfer: the data is stored to registers SPI_W0_REG ~ SPI_W15_REG.
  - DMA-controlled transfer: the data is stored to GDMA RX buffer.
- TX data: the TX data is from corresponding addresses according to transfer modes and is saved to buf_tx_afifo.
  - CPU-controlled transfer: TX data is from SPI_W0_REG ~ SPI_W15_REG.
  - DMA-controlled transfer: TX data is from GDMA TX buffer.

The data in buf_tx_afifo is sent out to Timing Module in 1/2/4-bit modes, controlled by GP-SPI2 state machine. The Timing Module can be used for timing compensation. For more information, see Section 27.8.

#### 27.5.7.3 Data Flow Control in Slave Mode

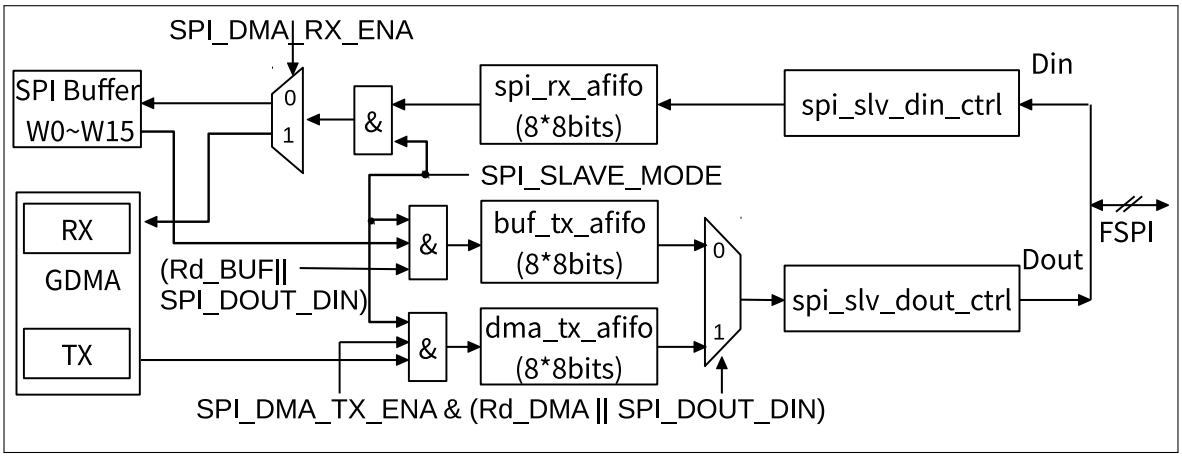



Figure 27.5-4. Data Flow Control in GP-SPI2 Slave Mode

Figure 27.5-4 shows the data flow in GP-SPI2 slave mode. Its control logic is as follows:

- In CPU/DMA-controlled full-duplex/half-duplex modes, when an external SPI master starts the SPI transfer, data on the FSPI bus is captured, converted into unit of bytes by spi_slv_din_ctrl module, and then is stored in spi_rx_afifo.
  - In CPU-controlled full-duplex transfer, the received data in spi_rx_afifo will be later stored into registers SPI_W0_REG ~ SPI_W15_REG, successively.
  - In half-duplex Wr_BUF transfer, when the value of address (SLV_ADDR[7:0]) is received, the received data in spi_rx_afifo will be stored in the related address of registers SPI_W0_REG ~ SPI_W15_REG
  - In DMA-controlled full-duplex transfer or in half-duplex Wr_DMA transfer, the received data in spi_rx_afifo will be stored in the configured GDMA RX buffer.
- In CPU-controlled full-/half-duplex transfer, the data to send is stored in buf_tx_afifo. In DMA-controlled full-/half-duplex transfer, the data to send is stored in dma_tx_afifo. Therefore, Rd_BUF transaction controlled by CPU and Rd_DMA transaction controlled by DMA can be done in one slave segmented transfer. TX data comes from corresponding addresses according the transfer modes.
  - In CPU-controlled full-duplex transfer, when SPI_SLAVE_MODE and SPI_DOUTDIN are set and SPI_DMA
    _TX_ENA is cleared, the data in SPI_W0_REG ~ SPI_W15_REG will be stored into buf_tx_afifo;
  - In CPU-controlled half-duplex transfer, when SPI_SLAVE_MODE is set, SPI_DOUTDIN is cleared, Rd_BUF command and SLV_ADDR[7:0] are received, the data started from the related address of SPI_W0_REG ~ SPI_W15_REG will be stored into buf_tx_afifo;
  - In DMA-controlled full-duplex transfer, when SPI_SLAVE_MODE, SPI_DOUTDIN and SPI_DMA_TX_ENA are set, the data in the configured GDMA TX buffer will be stored into dma_tx_afifo;

- In DMA-controlled half-duplex transfer, when SPI_SLAVE_MODE is set, SPI_DOUTDIN is cleared, and Rd_DMA command is received, the data in the configured GDMA TX buffer will be stored into dma_tx_afifo.

The data in buf_tx_afifo or dma_tx_afifo is sent out by spi_slv_dout_ctrl module in 1/2/4-bit modes.

### 27.5.8 GP-SPI2 Works as a Master

GP-SPI2 can be configured as a SPI master by clearing the bit SPI_SLAVE_MODE in SPI_SLAVE_REG. In this operation mode, GP-SPI2 provides clock signal (the divided clock from GP-SPI2 module clock) and six CS lines (CS0 ~ CS5).

**Note:**

- The length of transferred data must be in unit of bytes, otherwise the extra bits will be lost. The extra bits here means the result of total data bits % 8.
- To transfer bits not in unit of bytes, consider implementing it in CMD state or ADDR state.

#### 27.5.8.1 State Machine

When GP-SPI2 works as a master, the state machine controls its various states during data transfer, including configuration (CONF), preparation (PREP), command (CMD), address (ADDR), dummy (DUMMY), data out (DOUT), and data in (DIN) states. GP-SPI2 is mainly used to access 1/2/4-bit SPI devices, such as flash and external RAM, thus the naming of GP-SPI2 states keeps consistent with the sequence naming of flash and external RAM. The meaning of each state is described as follows and Figure 27.5-5 shows the workflow of GP-SPI2 state machine.

1. IDLE: GP-SPI2 is not active or is in slave mode.
2. CONF: only used in DMA-controlled configurable segmented transfer. Set SPI_USR and SPI_USR_CONF to enable this state. If this state is not enabled, it means the current transfer is a single transfer.
3. PREP: prepare an SPI transaction and control SPI CS setup time. Set SPI_USR and SPI_CS_SETUP to enable this state.
4. CMD: send command sequence. Set SPI_USR and SPI_USR_COMMAND to enable this state.
5. ADDR: send address sequence. Set SPI_USR and SPI_USR_ADDR to enable this state.
6. DUMMY (wait cycle): send dummy sequence. Set SPI_USR and SPI_USR_DUMMY to enable this state.
7. DATA: transfer data.
   - DOUT: send data sequence. Set SPI_USR and SPI_USR_MOSI to enable this state.
   - DIN: receive data sequence. Set SPI_USR and SPI_USR_MISO to enable this state.
8. DONE: control SPI CS hold time. Set SPI_USR to enable this state.

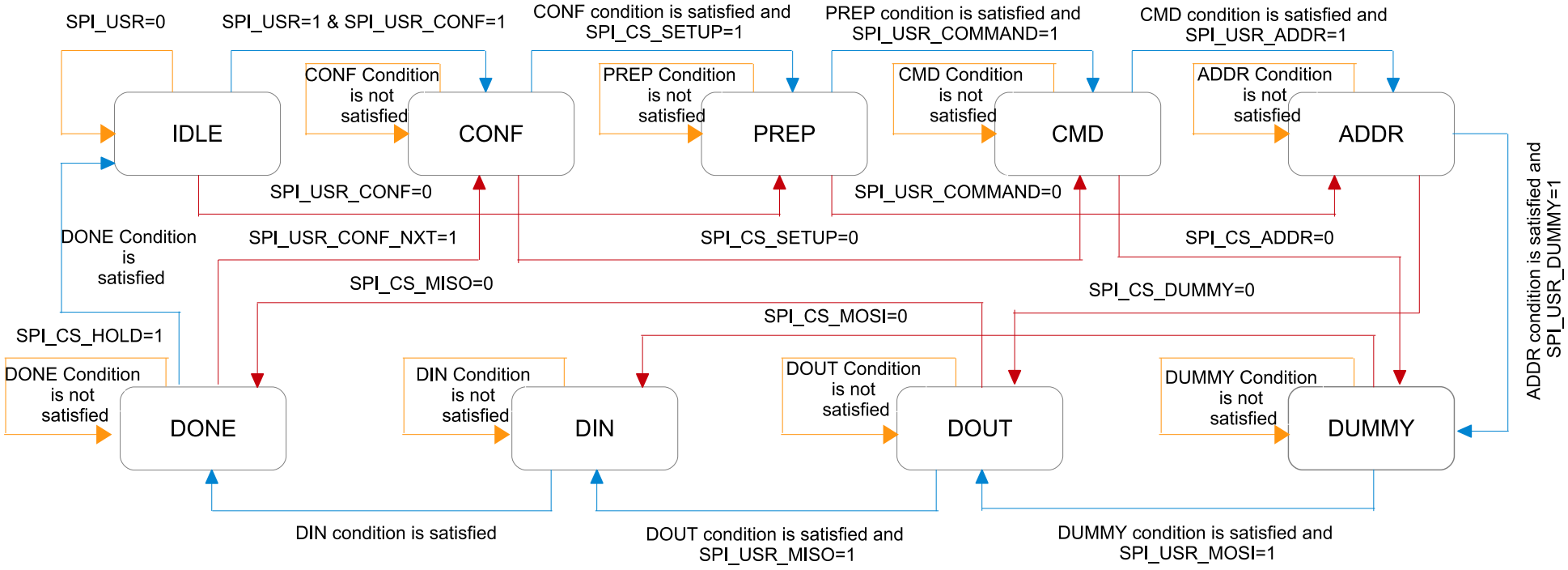



Figure 27.5-5. GP-SPI2 State Machine in Master Mode

Legend to state flow:

- —: indicates corresponding state condition is not satisfied; repeats current state.
- —: corresponding registers are set and conditions are satisfied; goes to next state.
- —: state registers are not set; skips one or more following states, depending on the registers of the following states are set or not.

Explanation to the conditions listed in the figure above:

- CONF condition: gpc[17:0] >= SPI_CONF_BITLEN[17:0]
- PREP condition: gpc[4:0] >= SPI_CS_SETUP_TIME[4:0]
- CMD condition: gpc[3:0] >= SPI_USR_COMMAND_BITLEN[3:0]
- ADDR condition: gpc[4:0] >= SPI_USR_ADDR_BITLEN[4:0]
- DUMMY condition: gpc[7:0] >= SPI_USR_DUMMY_CYCLELEN[7:0]
- DOUT condition: gpc[17:0] >= SPI_MS_DATA_BITLEN[17:0]
- DIN condition: gpc[17:0] >= SPI_MS_DATA_BITLEN[17:0]
- DONE condition: (gpc[4:0] >= SPI_CS_HOLD_TIME[4:0] || SPI_CS_HOLD == 1'b0)

A counter (gpc[17:0]) is used in the state machine to control the cycle length of each state. The states CONF, PREP, CMD, ADDR, DUMMY, DOUT, and DIN can be enabled or disabled independently. The cycle length of each state can also be configured independently.

#### 27.5.8.2 Register Configuration for State and Bit Mode Control

**Introduction**

The registers, related to GP-SPI2 state control, are listed in Table 27.5-7. Users can enable QPI mode for GP-SPI2 by setting the bit SPI_QPI_MODE in register SPI_USER_REG.

Table 27.5-7. Registers Used for State Control in 1/2/4-bit Modes

| State | Control Registers for 1-bit Mode FSPI Bus | Control Registers for 2-bit Mode FSPI Bus | Control Registers for 4-bit Mode FSPI Bus |
|---|---|---|---|
| CMD | SPI_USR_COMMAND_VALUE<br>SPI_USR_COMMAND_BITLEN<br>SPI_USR_COMMAND | SPI_USR_COMMAND_VALUE<br>SPI_USR_COMMAND_BITLEN<br>SPI_FCMD_DUAL<br>SPI_USR_COMMAND | SPI_USR_COMMAND_VALUE<br>SPI_USR_COMMAND_BITLEN<br>SPI_FCMD_QUAD<br>SPI_USR_COMMAND |
| ADDR | SPI_USR_ADDR_VALUE<br>SPI_USR_ADDR_BITLEN<br>SPI_USR_ADDR | SPI_USR_ADDR_VALUE<br>SPI_USR_ADDR_BITLEN<br>SPI_USR_ADDR<br>SPI_FADDR_DUAL | SPI_USR_ADDR_VALUE<br>SPI_USR_ADDR_BITLEN<br>SPI_USR_ADDR<br>SPI_FADDR_QUAD |
| DUMMY | SPI_USR_DUMMY_CYCLELEN<br>SPI_USR_DUMMY | SPI_USR_DUMMY_CYCLELEN<br>SPI_USR_DUMMY | SPI_USR_DUMMY_CYCLELEN<br>SPI_USR_DUMMY |
| DIN | SPI_USR_MISO<br>SPI_MS_DATA_BITLEN | SPI_USR_MISO<br>SPI_MS_DATA_BITLEN<br>SPI_FREAD_DUAL | SPI_USR_MISO<br>SPI_MS_DATA_BITLEN<br>SPI_FREAD_QUAD |

Table 27.5-7. Registers Used for State Control in 1/2/4-bit Modes

| State | Control Registers for 1-bit Mode FSPI Bus | Control Registers for 2-bit Mode FSPI Bus | Control Registers for 4-bit Mode FSPI Bus |
|---|---|---|---|
| DOUT | SPI_USR_MOSI<br>SPI_MS_DATA_BITLEN | SPI_USR_MOSI<br>SPI_MS_DATA_BITLEN<br>SPI_FWRITE_DUAL | SPI_USR_MOSI<br>SPI_MS_DATA_BITLEN<br>SPI_FWRITE_QUAD |

As shown in Table 27.5-7, the registers in each cell should be configured to set the FSPI bus to corresponding bit mode, i.e. the mode shown in the table header, at a specific state (corresponding to the first column).

**Configuration**

For instance, when GP-SPI2 reads data, and

- CMD is in 1-bit mode
- ADDR is in 2-bit mode
- DUMMY is 8 clock cycles
- DIN is in 4-bit mode

The register configuration can be as follows:

1. Configure CMD state related registers.
    - Configure the required command value in SPI_USR_COMMAND_VALUE.
    - Configure command bit length in SPI_USR_COMMAND_BITLEN. SPI_USR_COMMAND_BITLEN = expected bit length - 1.
    - Set SPI_USR_COMMAND.
    - Clear SPI_FCMD_DUAL and SPI_FCMD_QUAD.
2. Configure ADDR state related registers.
    - Configure the required address value in SPI_USR_ADDR_VALUE.
    - Configure address bit length in SPI_USR_ADDR_BITLEN. SPI_USR_ADDR_BITLEN = expected bit length - 1.
    - Set SPI_USR_ADDR and SPI_FADDR_DUAL.
    - Clear SPI_FADDR_QUAD.
3. Configure DUMMY state related registers.
    - Configure DUMMY cycles in SPI_USR_DUMMY_CYCLELEN. SPI_USR_DUMMY_CYCLELEN = expected clock cycles - 1.
    - Set SPI_USR_DUMMY.
4. Configure DIN state related registers.
    - Configure read data bit length in SPI_MS_DATA_BITLEN. SPI_MS_DATA_BITLEN = bit length expected - 1.

- Set SPI_FREAD_QUAD and SPI_USR_MISO.
- Clear SPI_FREAD_DUAL.
- Configure GDMA in DMA-controlled mode. In CPU controlled mode, no action is needed.

5. Clear SPI_USR_MOSI.
6. Set SPI_DMA_AFIFO_RST, SPI_BUF_AFIFO_RST, and SPI_RX_AFIFO_RST to reset these buffers.
7. Set SPI_USR to start GP-SPI2 transfer.

When writing data (DOUT state), SPI_USR_MOSI should be configured instead, while SPI_USR_MISO should be cleared. The output data bit length is the value of SPI_MS_DATA_BITLEN + 1. Output data should be configured in GP-SPI2 data buffer (SPI_W0_REG ~ SPI_W15_REG) in CPU-controlled mode, or GDMA TX buffer in DMA-controlled mode. The data byte order is incremented from LSB (byte 0) to MSB.

Pay special attention to the command value in SPI_USR_COMMAND_VALUE and to address value in SPI_USR_ADDR_VALUE.

The configuration of command value is as follows:

- If SPI_USR_COMMAND_BITLEN < 8, the command value is written to SPI_USR_COMMAND_VALUE[7:0]. Command value is sent as follows.
  - If SPI_WR_BIT_ORDER is set, the lower part of SPI_USR_COMMAND_VALUE[7:0], i.e. SPI_USR_COMMAND_VALUE[SPI_USR_COMMAND_BITLEN:0], is sent first.
  - If SPI_WR_BIT_ORDER is cleared, the higher part of SPI_USR_COMMAND_VALUE[7:0], i.e. SPI_USR_COMMAND_VALUE[7:7 - SPI_USR_COMMAND_BITLEN], is sent first.
- If 7 < SPI_USR_COMMAND_BITLEN < 16, the command value is written to SPI_USR_COMMAND_VALUE[15:0]. Command value is sent as follows.
  - If SPI_WR_BIT_ORDER is set, SPI_USR_COMMAND_VALUE[7:0] is sent first, and then the lower part of SPI_USR_COMMAND_VALUE[15:8], i.e. SPI_USR_COMMAND_VALUE[SPI_USR_COMMAND_BITLEN:8], is sent.
  - If SPI_WR_BIT_ORDER is cleared, SPI_USR_COMMAND_VALUE[7:0] is sent first, and then the higher part of SPI_USR_COMMAND_VALUE[15:8], i.e. SPI_USR_COMMAND_VALUE[15:15 - SPI_USR_COMMAND_BITLEN], is sent.

The configuration of address value is as follows:

- If SPI_USR_ADDR_BITLEN < 8, the address value is written to SPI_USR_ADDR_VALUE[31:24]. Address value is sent as follows.
  - If SPI_WR_BIT_ORDER is set, the lower part of SPI_USR_ADDR_VALUE[31:24], i.e. SPI_USR_ADDR_VALUE[SPI_USR_ADDR_BITLEN + 24:24], is sent first.
  - If SPI_WR_BIT_ORDER is cleared, the higher part of SPI_USR_ADDR_VALUE[31:24], i.e. SPI_USR_ADDR_VALUE[31:31 - SPI_USR_ADDR_BITLEN], is sent first.

- If 7 < SPI_USR_ADDR_BITLEN < 16, the ADDR value is written to SPI_USR_ADDR_VALUE[31:16]. Address value is sent as follows.
  - If SPI_WR_BIT_ORDER is set, SPI_USR_ADDR_VALUE[31:24] is sent first, and then the lower part of SPI_USR_ADDR_VALUE[23:16], i.e. SPI_USR_ADDR_VALUE[SPI_USR_ADDR_BITLEN + 8:16], is sent.
  - If SPI_WR_BIT_ORDER is cleared, SPI_USR_ADDR_VALUE[31:24] is sent first, and then the higher part of SPI_USR_ADDR_VALUE[23:16], i.e. SPI_USR_ADDR_VALUE[23:31 - SPI_USR_ADDR_BITLEN], is sent.
- If 15 < SPI_USR_ADDR_BITLEN < 24, the ADDR value is written to SPI_USR_ADDR_VALUE[31:8]. Address value is sent as follows.
  - If SPI_WR_BIT_ORDER is set, SPI_USR_ADDR_VALUE[31:16] is sent first, and then the lower part of SPI_USR_ADDR_VALUE[15:8], i.e. SPI_USR_ADDR_VALUE[SPI_USR_ADDR_BITLEN - 8:8], is sent.
  - If SPI_WR_BIT_ORDER is cleared, SPI_USR_ADDR_VALUE[31:16] is sent first, and then the higher part of SPI_USR_ADDR_VALUE[15:8], i.e. SPI_USR_ADDR_VALUE[15:31 - SPI_USR_ADDR_BITLEN], is sent.
- If 23 < SPI_USR_ADDR_BITLEN < 32, the ADDR value is written to SPI_USR_ADDR_VALUE[31:0]. Address value is sent as follows.
  - If SPI_WR_BIT_ORDER is set, SPI_USR_ADDR_VALUE[31:8] is sent first, and then the lower part of SPI_USR_ADDR_VALUE[7:0], i.e. SPI_USR_ADDR_VALUE[SPI_USR_ADDR_BITLEN - 24:0], is sent.
  - If SPI_WR_BIT_ORDER is cleared, SPI_USR_ADDR_VALUE[31:8] is sent first, and then the higher part of SPI_USR_ADDR_VALUE[7:0], i.e. SPI_USR_ADDR_VALUE[7:31 - SPI_USR_ADDR_BITLEN], is sent.

#### 27.5.8.3 Full-Duplex Communication (1-bit Mode Only)

**Introduction**

GP-SPI2 supports SPI full-duplex communication. In this mode, SPI master provides CLK and CS signals, exchanging data with SPI slave in 1-bit mode via MOSI (FSPID, sending) and MISO (FSPIQ, receiving) at the same time. To enable this communication mode, set the bit SPI_DOUTDIN in register SPI_USER_REG. Figure 27.5-6 illustrates the connection of GP-SPI2 with its slave in full-duplex communication.

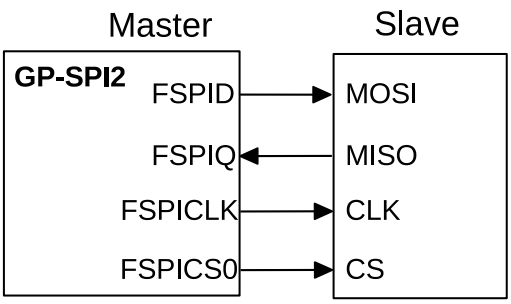



Figure 27.5-6. Full-Duplex Communication Between GP-SPI2 Master and a Slave

In full-duplex communication, the behavior of states CMD, ADDR, DUMMY, DOUT and DIN are configurable. Usually, the states CMD, ADDR and DUMMY are not used in this communication. The bit length of transferred data is configured in SPI_MS_DATA_BITLEN. The actual bit length used in communication equals to (SPI_MS_DATA_BITLEN + 1).

#### Configuration

To start a data transfer, follow the steps below:

- Configure the IO path via IO MUX or GPIO matrix between GP-SPI2 and an external SPI device.
- Configure APB clock (APB_CLK, see Chapter 6 *Reset and Clock*) and module clock (clk_spi_mst) for the GP-SPI2 module.
- Set SPI_DOUTDIN and clear SPI_SLAVE_MODE, to enable full-duplex communication in master mode.
- Configure GP-SPI2 registers listed in Table 27.5-7.
- Configure SPI CS setup time and hold time according to Section 27.6.
- Set the property of FSPICLK according to Section 27.7.
- Prepare data according to the selected transfer mode:
  - In CPU-controlled MOSI mode, prepare data in registers SPI_W0_REG ~ SPI_W15_REG.
  - In DMA-controlled mode,
    * configure SPI_DMA_TX_ENA/SPI_DMA_RX_ENA
    * configure GDMA TX/RX link
    * start GDMA TX/RX engine, as described in Section 27.5.6 and Section 27.5.7.
- Configure interrupts and wait for SPI slave to get ready for transfer.
- Set SPI_DMA_AFIFO_RST, SPI_BUF_AFIFO_RST, and SPI_RX_AFIFO_RST to reset these buffers.
- Set SPI_USR in register SPI_CMD_REG to start the transfer and wait for the configured interrupts.

### 27.5.8.4 Half-Duplex Communication (1/2/4-bit Mode)

#### Introduction

In this mode, GP-SPI2 provides CLK and CS signals. Only one side (SPI master or slave) can send data at a time, while the other side receives the data. To enable this communication mode, clear the bit SPI_DOUTDIN in register SPI_USER_REG. The standard format of SPI half-duplex communication is CMD + [ADDR +] [DUMMY +] [DOUT or DIN]. The states ADDR, DUMMY, DOUT, and DIN are optional, and can be disabled or enabled independently.

As described in Section 27.5.8.2, the properties of GP-SPI2 states: CMD, ADDR, DUMMY, DOUT and DIN, such as cycle length, value, and parallel bus bit mode, can be set independently. For the register configuration, see Table 27.5-7.

The detailed properties of half-duplex GP-SPI2 are as follows:

1. CMD: 0 ~ 16 bits, master output, slave input.
2. ADDR: 0 ~ 32 bits, master output, slave input.
3. DUMMY: 0 ~ 256 FSPICLK cycles, master output, slave input.
4. DOUT: 0 ~ 512 bits (64 B) in CPU-controlled mode and 0 ~ 256 Kbits (32 KB) in DMA-controlled mode, master output, slave input.

5. DIN: 0 ~ 512 bits (64 B) in CPU-controlled mode and 0 ~ 256 Kbits (32 KB) in DMA-controlled mode, master input, slave output.

#### Configuration

The register configuration is as follows:

1. Configure the IO path via IO MUX or GPIO matrix between GP-SPI2 and an external SPI device.
2. Configure APB clock (APB_CLK) and module clock (clk_spi_mst) for the GP-SPI2 module.
3. Clear SPI_DOUTDIN and SPI_SLAVE_MODE, to enable half-duplex communication in master mode.
4. Configure GP-SPI2 registers listed in Table 27.5-7.
5. Configure SPI CS setup time and hold time according to Section 27.6.
6. Set the property of FSPICLK according to Section 27.7.
7. Prepare data according to the selected transfer mode:
   - In CPU-controlled MOSI mode, prepare data in registers SPI_W0_REG ~ SPI_W15_REG.
   - In DMA-controlled mode,
     - configure SPI_DMA_TX_ENA/SPI_DMA_RX_ENA
     - configure GDMA TX/RX link
     - start GDMA TX/RX engine, as described in Section 27.5.6 and Section 27.5.7.
8. Configure interrupts and wait for SPI slave to get ready for transfer.
9. Set SPI_DMA_AFIFO_RST, SPI_BUF_AFIFO_RST, and SPI_RX_AFIFO_RST to reset these buffers.
10. Set SPI_USR in register SPI_CMD_REG to start the transfer and wait for the configured interrupts.

#### Application Example

The following example shows how GP-SPI2 to access flash and external RAM in master half-duplex mode.

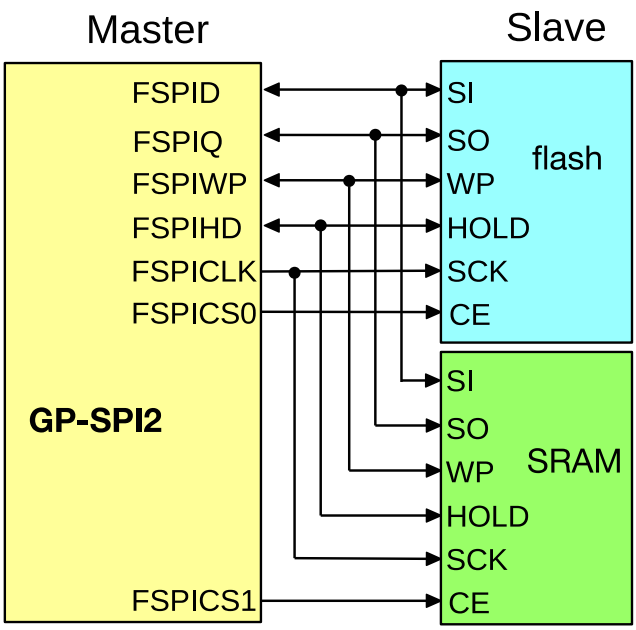



Figure 27.5-7. Connection of GP-SPI2 to Flash and External RAM in 4-bit Mode

Figure 27.5-8 indicates GP-SPI2 Quad I/O Read sequence according to standard flash specification. Other GP-SPI2 command sequences are implemented in accordance with the requirements of SPI slaves.

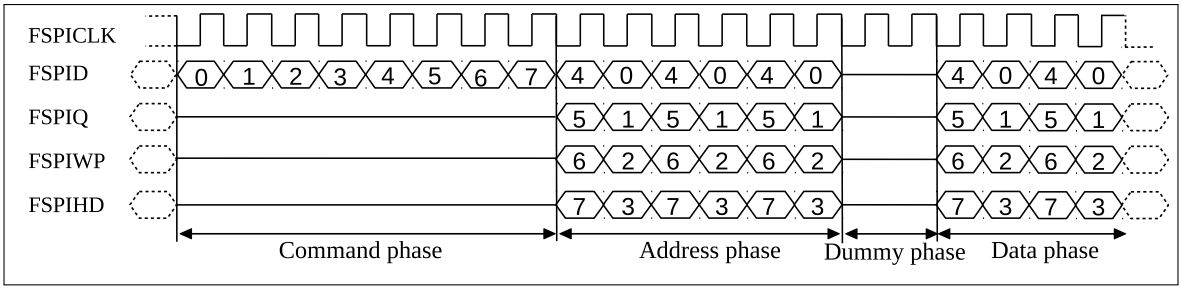



Figure 27.5-8. SPI Quad I/O Read Command Sequence Sent by GP-SPI2 to Flash

### 27.5.8.5 DMA-Controlled Configurable Segmented Transfer

**Note:**
Note that there is no separate section on how to configure a single transfer in master mode, since the CONF state of a configurable segmented transfer can be skipped to implement a single transfer.

#### Introduction

When GP-SPI2 works as a master, it provides a feature named: configurable segmented transfer controlled by DMA.

A DMA-controlled transfer in master mode can be

- a single transfer, consisting of only one transaction;
- or a configurable segmented transfer, consisting of several transactions (segments).

In a configurable segmented transfer, the registers of its each single transaction (segment) are configurable.

This feature enables GP-SPI2 to do as many as transactions (segments) as configured when such transfer is triggered once by CPU. Figure 27.5-9 shows how this feature works.

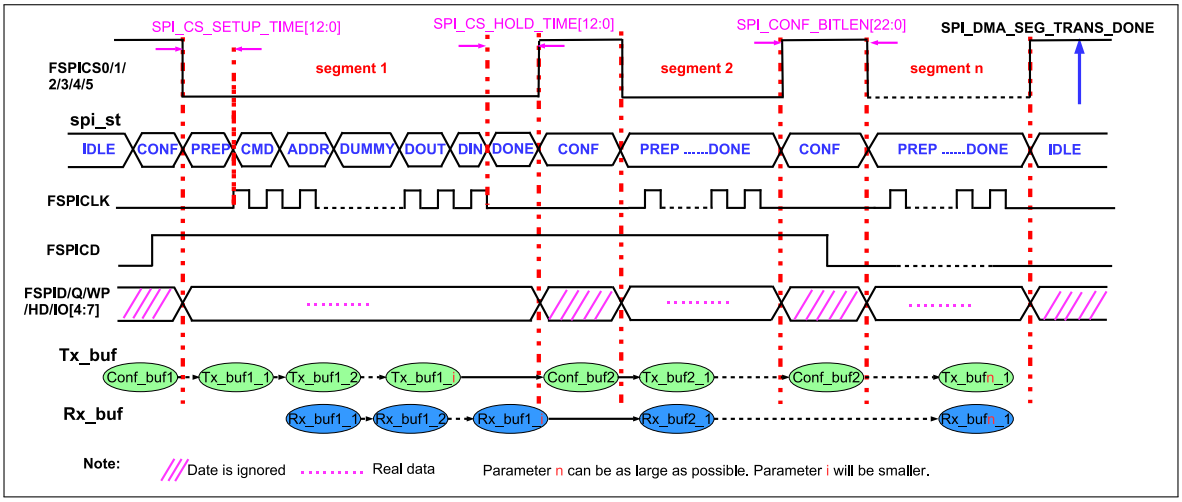



Figure 27.5-9. Configurable Segmented Transfer in DMA-Controlled Master Mode

As shown in Figure 27.5-9, the registers for one transaction (segment *n*) can be reconfigured by GP-SPI2 hardware according to the content in its Conf_buf*n* during a CONF state, before this segment starts.

It's recommended to provide separate GDMA CONF links and CONF buffers (Conf_buf*i* in Figure 27.5-9) for each CONF state. A GDMA TX link is used to connect all the CONF buffers and TX data buffers (Tx_buf*i* in Figure 27.5-9) into a chain. Hence, the behavior of the FSPI bus in each segment can be controlled independently.

For example, in a configurable segmentent transfer, its segment*i*, segment*j*, and segment*k* can be configured to full-duplex, half-duplex MISO, and half-duplex MOSI, respectively. *i*, *j*, and *k* are integer variables, which can be any segment number.

Meanwhile, the state of GP-SPI2, the data length and cycle length of the FSPI bus, and the behavior of the GDMA, can be configured independently for each segment. When this whole DMA-controlled transfer (consisting of several segments) has finished, a GP-SPI2 interrupt, SPI_DMA_SEG_TRANS_DONE_INT, is triggered.

### Configuration

1. Configure the IO path via IO MUX or GPIO matrix between GP-SPI2 and an external SPI device.
2. Configure APB clock (APB_CLK) and module clock (clk_spi_mst) for GP-SPI2 module.
3. Clear SPI_DOUTDIN and SPI_SLAVE_MODE, to enable half-duplex communication in master mode.
4. Configure GP-SPI2 registers listed in Table 27.5-7.
5. Configure SPI CS setup time and hold time according to Section 27.6.
6. Set the property of FSPICLK according to Section 27.7.
7. Prepare descriptors for GDMA CONF buffer and TX data (optional) for each segment. Chain the descriptors of CONF buffer and TX buffers of several segments into one linked list.
8. Similarly, prepare descriptors for RX buffers for each segment and chain them into one linked list.

9. Configure all the needed CONF buffers, TX buffers and RX buffers, respectively for each segment before this DMA-controlled transfer begins.
10. Point GDMA_OUTLINK_ADDR_CH*n* to the head address of the CONF and TX buffer descriptor linked list, and then set GDMA_OUTLINK_START_CH*n* to start the TX GDMA.
11. Clear the bit SPI_RX_EOF_EN in register SPI_DMA_CONF_REG. Point GDMA_INLINK_ADDR_CH*n* to the head address of the RX buffer descriptor linked list, and then set GDMA_INLINK_START_CH*n* to start the RX GDMA.
12. Set SPI_USR_CONF to enable CONF state.
13. Set SPI_DMA_SEG_TRANS_DONE_INT_ENA to enable the SPI_DMA_SEG_TRANS_DONE_INT interrupt. Configure other interrupts if needed according to Section 27.9.
14. Wait for all the slaves to get ready for transfer.
15. Set SPI_DMA_AFIFO_RST, SPI_BUF_AFIFO_RST and SPI_RX_AFIFO_RST, to reset these buffers.
16. Set SPI_USR to start this DMA-controlled transfer.
17. Wait for SPI_DMA_SEG_TRANS_DONE_INT interrupt, which means this transfer has finished and the data has been stored into corresponding memory.

#### Configuration of CONF Buffer and Magic Value

On GP-SPI2, only registers which will change from the last transaction (segment) need to be re-configured to new values in CONF state. The configuration of other registers can be skipped (i.e. kept the same) to save time and chip resources.

The first word in GDMA CONF buffer*i*, called SPI_BIT_MAP_WORD, defines whether each GP-SPI2 register is to be updated or not in segment*i*. The relation of SPI_BIT_MAP_WORD and GP-SPI2 registers to update can be seen in Table 27.5-8 Bitmap (BM) Table. If a bit in the BM table is set to 1, its corresponding register value will be updated in this segment. Otherwise, if some registers should be kept from being changed, the related bits should be set to 0.

Table 27.5-8. GP-SPI2 Master BM Table for CONF State

| BM Bit | Register Name | BM Bit | Register Name |
|---|---|---|---|
| 0 | SPI_ADDR_REG | 7 | SPI_MISC_REG |
| 1 | SPI_CTRL_REG | 8 | SPI_DIN_MODE_REG |
| 2 | SPI_CLOCK_REG | 9 | SPI_DIN_NUM_REG |
| 3 | SPI_USER_REG | 10 | SPI_DOUT_MODE_REG |
| 4 | SPI_USER1_REG | 11 | SPI_DMA_CONF_REG |
| 5 | SPI_USER2_REG | 12 | SPI_DMA_INT_ENA_REG |
| 6 | SPI_MS_DLEN_REG | 13 | SPI_DMA_INT_CLR_REG |

Then new values of all the registers to be modified should be placed right after SPI_BIT_MAP_WORD, in consecutive words in the CONF buffer.

To ensure the correctness of the content in each CONF buffer, the value in SPI_BIT_MAP_WORD[31:28] is used as "magic value", and will be compared with SPI_DMA_SEG_MAGIC_VALUE in the register SPI_SLAVE_REG. The value of SPI_DMA_SEG_MAGIC_VALUE should be configured before this DMA-controlled transfer starts, and can not be changed during these segments.

- If SPI_BIT_MAP_WORD[31:28] == SPI_DMA_SEG_MAGIC_VALUE, this DMA-controlled transfer continues normally; the interrupt SPI_DMA_SEG_TRANS_DONE_INT is triggered at the end of this DMA-controlled transfer.
- If SPI_BIT_MAP_WORD[31:28] != SPI_DMA_SEG_MAGIC_VALUE, GP-SPI2 state (spi_st) goes back to IDLE and the transfer is ended immediately. The interrupt SPI_DMA_SEG_TRANS_DONE_INT is still triggered, with SPI_SEG_MAGIC_ERR_INT_RAW bit set to 1.

#### CONF Buffer Configuration Example

Table 27.5-9 and Table 27.5-10 provide an example to show how to configure a CONF buffer for a transaction (segment *i*) whose SPI_ADDR_REG, SPI_CTRL_REG, SPI_CLOCK_REG, SPI_USER_REG, SPI_USER1_REG need to be updated.

Table 27.5-9. An Example of CONF buffer*i* in Segment*i*

| CONF buffer*i* | Note |
|---|---|
| SPI_BIT_MAP_WORD | The first word in this buffer. Its value is 0xA000001F in this example when the SPI_DMA_SEG_MAGIC_VALUE is set to 0xA. As shown in Table 27.5-10, bits 0, 1, 2, 3, and 4 are set, indicating the following registers will be updated. |
| SPI_ADDR_REG | The second word, stores the new value to SPI_ADDR_REG. |
| SPI_CTRL_REG | The third word, stores the new value to SPI_CTRL_REG. |
| SPI_CLOCK_REG | The fourth word, stores the new value to SPI_CLOCK_REG. |
| SPI_USER_REG | The fifth word, stores the new value to SPI_USER_REG. |
| SPI_USER1_REG | The sixth word, stores the new value to SPI_USER1_REG. |

Table 27.5-10. BM Bit Value v.s. Register to Be Updated in This Example

| BM Bit | Value | Register Name | BM Bit | Value | Register Name |
|---|---|---|---|---|---|
| 0 | 1 | SPI_ADDR_REG | 7 | 0 | SPI_MISC_REG |
| 1 | 1 | SPI_CTRL_REG | 8 | 0 | SPI_DIN_MODE_REG |
| 2 | 1 | SPI_CLOCK_REG | 9 | 0 | SPI_DIN_NUM_REG |
| 3 | 1 | SPI_USER_REG | 10 | 0 | SPI_DOUT_MODE_REG |
| 4 | 1 | SPI_USER1_REG | 11 | 0 | SPI_DMA_CONF_REG |
| 5 | 0 | SPI_USER2_REG | 12 | 0 | SPI_DMA_INT_ENA_REG |
| 6 | 0 | SPI_MS_DLEN_REG | 13 | 0 | SPI_DMA_INT_CLR_REG |

**Notes:**

In a DMA-controlled configurable segmented transfer, please pay special attention to the following bits:

- SPI_USR_CONF: set SPI_USR_CONF before SPI_USR is set, to enable this transfer.
- SPI_USR_CONF_NXT: if segment*i* is not the final transaction of this whole DMA-controlled transfer, its SPI_USR_CONF_NXT should be set to 1.
- SPI_CONF_BITLEN: GP-SPI2 CS setup time and hold time are programmable independently in each segment, see Section 27.6 for detailed configuration. The CS high time in each segment is about:

$$(SPI_CONF_BITLEN + 5) \times T_{APB_CLK}$$

The CS high time in CONF state can be set from 62.5 $\mu s$ to 3.2768 ms when $f_{APB_CLK}$ is 80 MHz. (SPI_CONF_
BITLEN + 5) will overflow from (0x40000 - SPI_CONF_BITLEN - 5) if SPI_CONF_BITLEN is larger than 0x3FFFA.

### 27.5.9 GP-SPI2 Works as a Slave

GP-SPI2 can be used as a slave to communicate with an SPI master. As a slave, GP-SPI2 supports 1-bit SPI, 2-bit dual SPI, 4-bit quad SPI, and QPI modes, with specific communication formats. To enable this mode, set SPI_SLAVE_MODE in register SPI_SLAVE_REG.

The CS signal must be held low during the transmission, and its falling/rising edges indicate the start/end of a single or segmented transmission. The length of transferred data must be in unit of bytes, otherwise the extra bits will be lost. The extra bits here means the result of total bits % 8.

#### 27.5.9.1 Communication Formats

In GP-SPI2 slave mode, SPI full-duplex and half-duplex communications are available. To select from the two communications, configure SPI_DOUTDIN in register SPI_USER_REG.

Full-duplex communication means that input data and output data are transmitted simultaneously throughout the entire transaction. All bits are treated as input or output data, which means no command, address or dummy states are expected. The interrupt SPI_TRANS_DONE_INT is triggered once the transaction ends.

In half-duplex communication, the format is CMD+ADDR+DUMMY+DATA (DIN or DOUT).

- "DIN" means that an SPI master reads data from GP-SPI2.
- "DOUT" means that an SPI master writes data to GP-SPI2.

The detailed properties of each state are as follows:

1. CMD:
    - Indicate the function of SPI slave;
    - One byte from master to slave;
    - Only the values in Table 27.5-11 and Table 27.5-12 are valid;
    - Can be sent in 1-bit SPI mode or 4-bit QPI mode.
2. ADDR:
    - The address for Wr_BUF and Rd_BUF commands in CPU-controlled transfer, or placeholder bits in other transfers and can be defined by application;
    - One byte from master to slave;
    - Can be sent in 1-bit, 2-bit or 4-bit modes (according to the command).
3. DUMMY:
    - It's value is meaningless. SPI slave prepares data in this state;
    - Bit mode of FSPI bus is also meaningless here;

- Last for eight SPI_CLK cycles.

4. DIN or DOUT:
   - Data length can be 0 ~ 64 B in CPU-controlled mode and unlimited in DMA-controlled mode;
   - Can be sent in 1-bit, 2-bit or 4-bit modes according to the CMD value.

> **Note:**
> The states of ADDR and DUMMY can never be omitted in any half-duplex communications.

When a half-duplex transaction is complete, the transferred CMD and ADDR values are latched into SPI_SLV_
LAST_COMMAND and SPI_SLV_LAST_ADDR respectively. The SPI_SLV_CMD_ERR_INT_RAW will be set if the transferred CMD value is not supported by GP-SPI2 slave mode. The SPI_SLV_CMD_ERR_INT_RAW can only be cleared by software.

#### 27.5.9.2 Supported CMD Values in Half-Duplex Communication

In half-duplex communication, the defined values of CMD determine the transfer types. Unsupported CMD values are disregarded, meanwhile the related transfer is ignored and SPI_SLV_CMD_ERR_INT_RAW is set. The transfer format is CMD (8 bits) + ADDR (8 bits) + DUMMY (8 SPI_CLK cycles) + DATA (unit in bytes). The detailed description of CMD[3:0] is as follows:

1. 0x1 (Wr_BUF): CPU-controlled write mode. Master sends data and GP-SPI2 receives data. The data is stored in the related address of SPI_W0_REG ~ SPI_W15_REG.
2. 0x2 (Rd_BUF): CPU-controlled read mode. Master receives the data sent by GP-SPI2. The data comes from the related address of SPI_W0_REG ~ SPI_W15_REG.
3. 0x3 (Wr_DMA): DMA-controlled write mode. Master sends data and GP-SPI2 receives data. The data is stored in GP-SPI2 GDMA RX buffer.
4. 0x4 (Rd_DMA): DMA-controlled read mode. Master receives the data sent by GP-SPI2. The data comes from GP-SPI2 GDMA TX buffer.
5. 0x7 (CMD7): used to generate an SPI_SLV_CMD7_INT interrupt. It can also generate a GDMA_IN_SUC_EOF
   _CH*n*_INT interrupt in a slave segmented transfer when GDMA RX link is used. But it will not end GP-SPI2's slave segmented transfer.
6. 0x8 (CMD8): only used to generate an SPI_SLV_CMD8_INT interrupt, which will not end GP-SPI2's slave segmented transfer.
7. 0x9 (CMD9): only used to generate an SPI_SLV_CMD9_INT interrupt, which will not end GP-SPI2's slave segmented transfer.
8. 0xA (CMDA): only used to generate an SPI_SLV_CMDA_INT interrupt, which will not end GP-SPI2's slave segmented transfer.

The detail function of CMD7, CMD8, CMD9, and CMDA commands is reserved for user definition. These commands can be used as handshake signals, the passwords of some specific functions, the triggers of some user defined actions, and so on.

1/2/4-bit modes in states of CMD, ADDR, DATA are supported, which are determined by value of CMD[7:4]. The DUMMY state is always in 1-bit mode and lasts for eight SPI_CLK cycles. The definition of CMD[7:4] is as follows:

1. 0x0: CMD, ADDR, and DATA states all are in 1-bit mode.
2. 0x1: CMD and ADDR are in 1-bit mode. DATA is in 2-bit mode.
3. 0x2: CMD and ADDR are in 1-bit mode. DATA is in 4-bit mode.
4. 0x5: CMD is in 1-bit mode. ADDR and DATA are in 2-bit mode.
5. 0xA: CMD is in 1-bit mode, ADDR and DATA are in 4-bit mode. Or in QPI mode.

In addition, if the value of CMD[7:0] is 0x05, 0xA5, 0x06, or 0xDD, DUMMY and DATA states are omitted. The definition of CMD[7:0] is as follows:

1. 0x05 (End_SEG_TRANS): master sends 0x05 command to end slave segmented transfer in SPI mode.
2. 0xA5 (End_SEG_TRANS): master sends 0xA5 command to end slave segmented transfer in QPI mode.
3. 0x06 (En_QPI): GP-SPI2 enters QPI mode when receiving the 0x06 command and the bit SPI_QPI_MODE in register SPI_USER_REG is set.
4. 0xDD (Ex_QPI): GP-SPI2 exits QPI mode when receiving the 0xDD command and the bit SPI_QPI_MODE is cleared.

All the GP-SPI2 supported CMD values are listed in Table 27.5-11 and Table 27.5-12. Note that DUMMY state is always in 1-bit mode and lasts for eight SPI_CLK cycles.

**Table 27.5-11. Supported CMD Values in SPI Mode**

| Transfer Type | CMD[7:0] | CMD State | ADDR State | DATA State |
|---|---|---|---|---|
| Wr_BUF | 0x01 | 1-bit mode | 1-bit mode | 1-bit mode |
| | 0x11 | 1-bit mode | 1-bit mode | 2-bit mode |
| | 0x21 | 1-bit mode | 1-bit mode | 4-bit mode |
| | 0x51 | 1-bit mode | 2-bit mode | 2-bit mode |
| | 0xA1 | 1-bit mode | 4-bit mode | 4-bit mode |
| Rd_BUF | 0x02 | 1-bit mode | 1-bit mode | 1-bit mode |
| | 0x12 | 1-bit mode | 1-bit mode | 2-bit mode |
| | 0x22 | 1-bit mode | 1-bit mode | 4-bit mode |
| | 0x52 | 1-bit mode | 2-bit mode | 2-bit mode |
| | 0xA2 | 1-bit mode | 4-bit mode | 4-bit mode |
| Wr_DMA | 0x03 | 1-bit mode | 1-bit mode | 1-bit mode |
| | 0x13 | 1-bit mode | 1-bit mode | 2-bit mode |
| | 0x23 | 1-bit mode | 1-bit mode | 4-bit mode |
| | 0x53 | 1-bit mode | 2-bit mode | 2-bit mode |
| | 0xA3 | 1-bit mode | 4-bit mode | 4-bit mode |
| Rd_DMA | 0x04 | 1-bit mode | 1-bit mode | 1-bit mode |
| | 0x14 | 1-bit mode | 1-bit mode | 2-bit mode |
| | 0x24 | 1-bit mode | 1-bit mode | 4-bit mode |
| | 0x54 | 1-bit mode | 2-bit mode | 2-bit mode |
| | 0xA4 | 1-bit mode | 4-bit mode | 4-bit mode |

Table 27.5-11. Supported CMD Values in SPI Mode

| Transfer Type | CMD[7:0] | CMD State | ADDR State | DATA State |
|---|---|---|---|---|
| CMD7 | 0x07 | 1-bit mode | 1-bit mode | - |
| | 0x17 | 1-bit mode | 1-bit mode | - |
| | 0x27 | 1-bit mode | 1-bit mode | - |
| | 0x57 | 1-bit mode | 2-bit mode | - |
| | 0xA7 | 1-bit mode | 4-bit mode | - |
| CMD8 | 0x08 | 1-bit mode | 1-bit mode | - |
| | 0x18 | 1-bit mode | 1-bit mode | - |
| | 0x28 | 1-bit mode | 1-bit mode | - |
| | 0x58 | 1-bit mode | 2-bit mode | - |
| | 0xA8 | 1-bit mode | 4-bit mode | - |
| CMD9 | 0x09 | 1-bit mode | 1-bit mode | - |
| | 0x19 | 1-bit mode | 1-bit mode | - |
| | 0x29 | 1-bit mode | 1-bit mode | - |
| | 0x59 | 1-bit mode | 2-bit mode | - |
| | 0xA9 | 1-bit mode | 4-bit mode | - |
| CMDA | 0x0A | 1-bit mode | 1-bit mode | - |
| | 0x1A | 1-bit mode | 1-bit mode | - |
| | 0x2A | 1-bit mode | 1-bit mode | - |
| | 0x5A | 1-bit mode | 2-bit mode | - |
| | 0xAA | 1-bit mode | 4-bit mode | - |
| End_SEG_TRANS | 0x05 | 1-bit mode | - | - |
| En_QPI | 0x06 | 1-bit mode | - | - |

Table 27.5-12. Supported CMD Values in QPI Mode

| Transfer Type | CMD[7:0] | CMD State | ADDR State | DATA State |
|---|---|---|---|---|
| Wr_BUF | 0xA1 | 4-bit mode | 4-bit mode | 4-bit mode |
| Rd_BUF | 0xA2 | 4-bit mode | 4-bit mode | 4-bit mode |
| Wr_DMA | 0xA3 | 4-bit mode | 4-bit mode | 4-bit mode |
| Rd_DMA | 0xA4 | 4-bit mode | 4-bit mode | 4-bit mode |
| CMD7 | 0xA7 | 4-bit mode | 4-bit mode | - |
| CMD8 | 0xA8 | 4-bit mode | 4-bit mode | - |
| CMD9 | 0xA9 | 4-bit mode | 4-bit mode | - |
| CMDA | 0xAA | 4-bit mode | 4-bit mode | - |
| End_SEG_TRANS | 0xA5 | 4-bit mode | 4-bit mode | - |
| Ex_QPI | 0xDD | 4-bit mode | 4-bit mode | - |

Master sends 0x06 CMD (En_QPI) to set GP-SPI2 slave to QPI mode and all the states of supported transfer will be in 4-bit mode afterwards. If 0xDD CMD (Ex_QPI) is received, GP-SPI2 slave will be back to SPI mode.

Other transfer types than described in Table 27.5-11 and Table 27.5-12 are ignored. If the transferred data is not in unit of byte, GP-SPI2 can send or receive these extra bits (total bits % 8), however, the correctness of the

data is not guaranteed. But if the CS low time is longer than 2 APB clock (APB_CLK) cycles, SPI_TRANS_DONE_INT will be triggered. For more information on interrupts triggered at the end of transmissions, please refer to Section 27.9.

#### 27.5.9.3 Slave Single Transfer and Slave Segmented Transfer

When GP-SPI2 works as a slave, it supports full-duplex and half-duplex communications controlled by DMA and by CPU. DMA-controlled transfer can be a single transfer, or a slave segmented transfer consisting of several transactions (segments). The CPU-controlled transfer can only be one single transfer, since each CPU-controlled transaction needs to be triggered by CPU.

In a slave segmented transfer, all transfer types listed in Table 27.5-11 and Table 27.5-12 are supported in a single transaction (segment). It means that CPU-controlled transaction and DMA-controlled transaction can be mixed in one slave segmented transfer.

It is recommended that in a slave segmented transfer:

- CPU-controlled transaction is used for handshake communication and short data transfers.
- DMA-controlled transaction is used for large data transfers.

#### 27.5.9.4 Configuration of Slave Single Transfer

In slave mode, GP-SPI2 supports CPU/DMA-controlled full-duplex/half-duplex single transfers. The register configuration procedure is as follows:

1. Configure the IO path via IO MUX or GPIO matrix between GP-SPI2 and an external SPI device.
2. Configure APB clock (APB_CLK).
3. Set the bit SPI_SLAVE_MODE, to enable slave mode.
4. Configure SPI_DOUTDIN:
   - 1: enable full-duplex communication.
   - 0: enable half-duplex communication.
5. Prepare data:
   - if CPU-controlled transfer mode is selected and GP-SPI2 is used to send data, then prepare data in registers SPI_W0_REG ~ SPI_W15_REG.
   - if DMA-controlled transfer mode is selected,
     - configure SPI_DMA_TX_ENA/SPI_DMA_RX_ENA and SPI_RX_EOF_EN.
     - configure GDMA TX/RX link.
     - start GDMA TX/RX engine, as described in Section 27.5.6 and Section 27.5.7.
6. Set SPI_DMA_AFIFO_RST, SPI_BUF_AFIFO_RST, and SPI_RX_AFIFO_RST to reset these buffers.
7. Clear SPI_DMA_SLV_SEG_TRANS_EN in register SPI_DMA_CONF_REG to enable slave single transfer mode.
8. Set SPI_TRANS_DONE_INT_ENA in SPI_DMA_INT_ENA_REG and wait for the interrupt SPI_TRANS_DONE_INT. In DMA-controlled mode, it is recommended to wait for the interrupt

GDMA_IN_SUC_EOF_CH*n*_INT when GDMA RX buffer is used, which means that data has been stored in the related memory. Other interrupts described in Section 27.9 are optional.

#### 27.5.9.5 Configuration of Slave Segmented Transfer in Half-Duplex

GDMA must be used in this mode. The register configuration procedure is as follows:

1. Configure the IO path via IO MUX or GPIO matrix between GP-SPI2 and an external SPI device.
2. Configure APB clock (APB_CLK).
3. Set SPI_SLAVE_MODE to enable slave mode.
4. Clear SPI_DOUTDIN to enable half-duplex communication.
5. Prepare data in registers SPI_W0_REG ~ SPI_W15_REG, if needed.
6. Set SPI_DMA_AFIFO_RST, SPI_BUF_AFIFO_RST and SPI_RX_AFIFO_RST to reset these buffers.
7. Set bits SPI_DMA_RX_ENA and SPI_DMA_TX_ENA. Clear the bit SPI_RX_EOF_EN. Configure GDMA TX/RX link and start GDMA TX/RX engine, as shown in Section 27.5.6 and Section 27.5.7.
8. Set SPI_DMA_SLV_SEG_TRANS_EN in SPI_DMA_CONF_REG to enable slave segmented transfer.
9. Set SPI_DMA_SEG_TRANS_DONE_INT_ENA in SPI_DMA_INT_ENA_REG and wait for the interrupt SPI_DMA_SEG_TRANS_DONE_INT, which means that the segmented transfer has finished and data has been put into the related memory. Other interrupts described in Section 27.9 are optional.

When End_SEG_TRANS (0x05 in SPI mode, 0xA5 in QPI mode) is received by GP-SPI2, this slave segmented transfer is ended and the interrupt SPI_DMA_SEG_TRANS_DONE_INT is triggered.

#### 27.5.9.6 Configuration of Slave Segmented Transfer in Full-Duplex

GDMA must be used in this mode. In such transfer, the data is transferred from and to the GDMA buffer. The interrupt GDMA_IN_SUC_EOF_CH*n*
_INT is triggered when the transfer ends. The configuration procedure is as follows:

1. Configure the IO path via IO MUX or GPIO matrix between GP-SPI2 and an external SPI device.
2. Configure APB clock (APB_CLK).
3. Set SPI_SLAVE_MODE and SPI_DOUTDIN, to enable full-duplex communication in slave mode.
4. Set SPI_DMA_AFIFO_RST, SPI_BUF_AFIFO_RST, and SPI_RX_AFIFO_RST bit, to reset these buffers.
5. Set SPI_DMA_TX_ENA/SPI_DMA_RX_ENA. Configure GDMA TX/RX link and start GDMA TX/RX engine, as shown in Section 27.5.6 and Section 27.5.7.
6. Set the bit SPI_RX_EOF_EN in register SPI_DMA_CONF_REG. Configure SPI_MS_DATA_BITLEN[17:0] in register SPI_MS_DLEN_REG to the byte length of the received DMA data.
7. Set SPI_DMA_SLV_SEG_TRANS_EN in SPI_DMA_CONF_REG to enable slave segmented transfer mode.
8. Set GDMA_IN_SUC_EOF_CH*n*_INT_ENA and wait for the interrupt GDMA_IN_SUC_EOF_CH*n*_INT.

## 27.6 CS Setup Time and Hold Time Control

SPI bus CS (SPI_CS) setup time and hold time are very important to meet the timing requirements of various SPI devices (e.g. flash or PSRAM).

CS setup time is the time between the CS falling edge and the first latch edge of SPI bus CLK (SPI_CLK). The first latch edge for mode 0 and mode 3 is rising edge, and falling edge for mode 2 and mode 4.

CS hold time is the time between the last latch edge of SPI_CLK and the CS rising edge.

In slave mode, the CS setup time and hold time should be longer than 0.5 x T_SPI_CLK, otherwise the SPI transfer may be incorrect. T_SPI_CLK: one cycle of SPI_CLK.

In master mode, set the CS setup time by specifying SPI_CS_SETUP in SPI_USER_REG and SPI_CS_SETUP_TIME in SPI_USER1_REG:

- If SPI_CS_SETUP is cleared, the SPI CS setup time is 0.5 x T_SPI_CLK.
- If SPI_CS_SETUP is set, the SPI CS setup time is (SPI_CS_SETUP_TIME + 1.5) x T_SPI_CLK.

Set the CS hold time by specifying SPI_CS_HOLD in SPI_USER_REG and SPI_CS_HOLD_TIME in SPI_USER1_REG:

- If SPI_CS_HOLD is cleared, the SPI CS hold time is 0.5 x T_SPI_CLK;
- If SPI_CS_HOLD is set, the SPI CS hold time is (SPI_CS_HOLD_TIME + 1.5) x T_SPI_CLK.

Figure 27.6-1 and Figure 27.6-2 show the recommended CS timing and register configuration to access external RAM and flash.

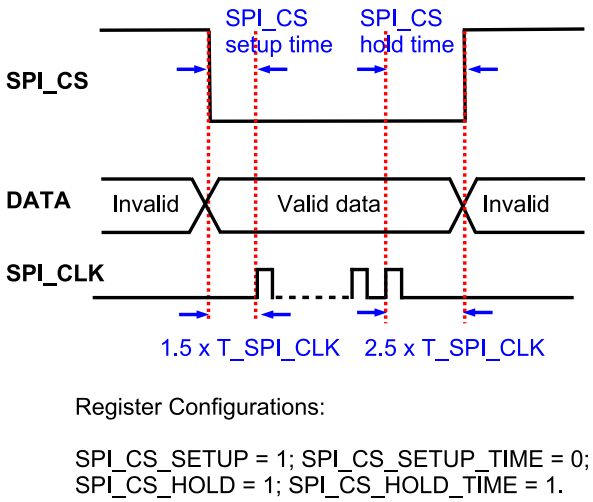



Figure 27.6-1. Recommended CS Timing and Settings When Accessing External RAM

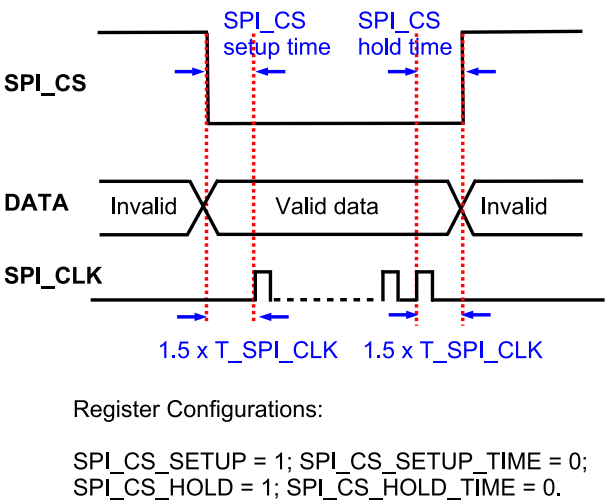



Figure 27.6-2. Recommended CS Timing and Settings When Accessing Flash

## 27.7 GP-SPI2 Clock Control

GP-SPI2 has the following clocks:

- clk_spi_mst: module clock of GP-SPI2, derived from PLL_CLK. Used in GP-SPI2 master mode, to generate SPI_CLK signal for data transfer and for slaves.
- SPI_CLK: output clock in master mode.
- APB_CLK: clock for register configuration.
- clk_hclk: module timing compensation clock of GP-SPI2.

In master mode, the maximum output clock frequency of GP-SPI2 is $f_{\text{clk_spi_mst}}$. To have slower frequencies, the output clock frequency can be divided as follows:

$$f_{\text{SPI_CLK}} = \frac{f_{\text{clk_spi_mst}}}{(\text{SPI_CLKCNT_N} + 1)(\text{SPI_CLKDIV_PRE} + 1)}$$

The divider is configured by SPI_CLKCNT_N and SPI_CLKDIV_PRE in register SPI_CLOCK_REG. When the bit SPI_CLK_EQU_SYSCLK in register SPI_CLOCK_REG is set to 1, the output clock frequency of GP-SPI2 will be $f_{\text{clk_spi_mst}}$. And for other integral clock divisions, SPI_CLK_EQU_SYSCLK should be set to 0.

In slave mode, the supported input clock frequency ($f_{\text{SPI_CLK}}$) of GP-SPI2 is:

- If $f_{\text{APB_CLK}}$ >= 60 MHz, $f_{\text{SPI_CLK}}$ <= 60 MHz;
- If $f_{\text{APB_CLK}}$ < 60 MHz, $f_{\text{SPI_CLK}}$ <= $f_{\text{APB_CLK}}$.

### 27.7.1 Clock Phase and Polarity

There are four clock modes in SPI protocol, modes 0 ~ 3, see Figure 27.7-1 and Figure 27.7-2 (excerpted from SPI protocol):

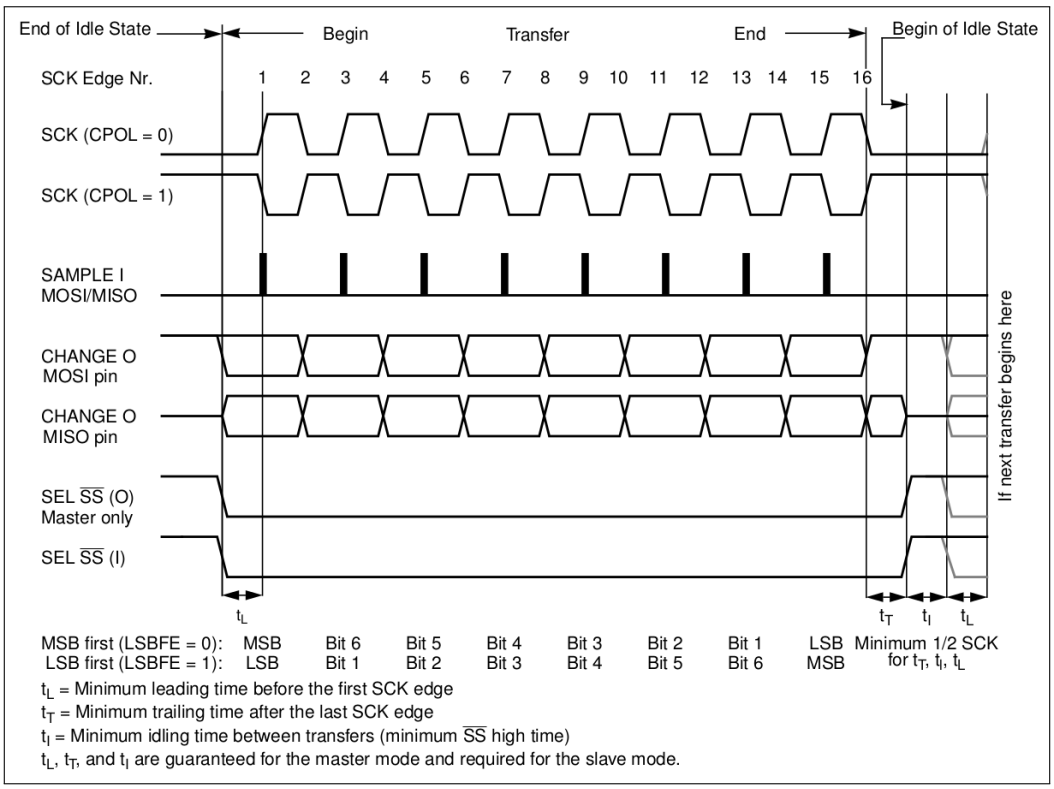



Figure 27.7-1. SPI Clock Mode 0 or 2

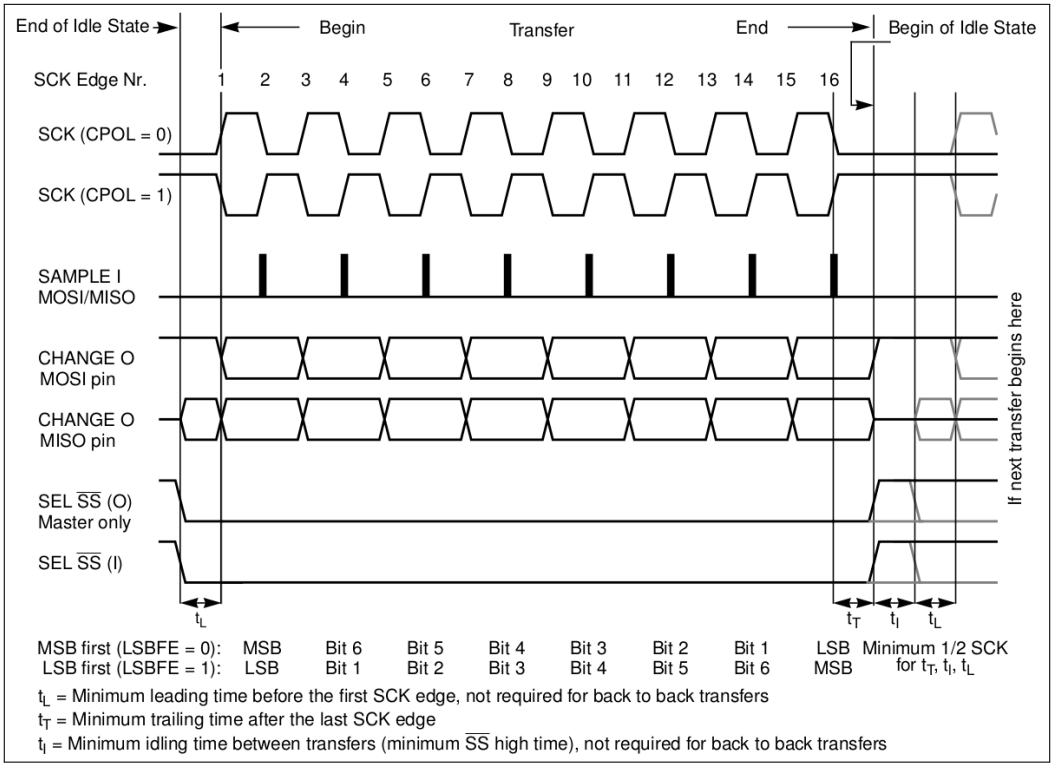



Figure 27.7-2. SPI Clock Mode 1 or 3

1. Mode 0: CPOL = 0, CPHA = 0; SCK is 0 when the SPI is in idle state; data is changed on the negative edge of SCK and sampled on the positive edge. The first data is shifted out before the first negative edge of SCK.

2. Mode 1: CPOL = 0, CPHA = 1; SCK is 0 when the SPI is in idle state; data is changed on the positive edge of SCK and sampled on the negative edge.
3. Mode 2: CPOL = 1, CPHA = 0; SCK is 1 when the SPI is in idle state; data is changed on the positive edge of SCK and sampled on the negative edge. The first data is shifted out before the first positive edge of SCK.
4. Mode 3: CPOL = 1, CPHA = 1; SCK is 1 when the SPI is in idle state; data is changed on the negative edge of SCK and sampled on the positive edge.

### 27.7.2 Clock Control in Master Mode

The four clock modes 0 ~ 3 are supported in GP-SPI2 master mode. The polarity and phase of GP-SPI2 clock are controlled by the bit SPI_CK_IDLE_EDGE in register SPI_MISC_REG and the bit SPI_CK_OUT_EDGE in register SPI_USER_REG. The register configuration for SPI clock modes 0 ~ 3 is provided in Table 27.7-1, and can be changed according to the path delay in the application.

Table 27.7-1. Clock Phase and Polarity Configuration in Master Mode

| Control Bit | Mode 0 | Mode 1 | Mode 2 | Mode 3 |
|---|---|---|---|---|
| SPI_CK_IDLE_EDGE | 0 | 0 | 1 | 1 |
| SPI_CK_OUT_EDGE | 0 | 1 | 1 | 0 |

SPI_CLK_MODE is used to select the number of rising edges of SPI_CLK, when SPI_CS raises high, to be 0, 1, 2 or SPI_CLK always on.

**Note:**
When SPI_CLK_MODE is configured to 1 or 2, the bit SPI_CS_HOLD must be set and the value of SPI_CS_HOLD_TIME should be larger than 1.

### 27.7.3 Clock Control in Slave Mode

GP-SPI2 slave mode also supports clock modes 0 ~ 3. The polarity and phase are configured by the bits SPI_TSCK_I_EDGE and SPI_RSCK_I_EDGE in register SPI_USER_REG. The output edge of data is controlled by SPI_CLK_MODE_13 in register SPI_SLAVE_REG. The detailed register configuration is shown in Table 27.7-2:

Table 27.7-2. Clock Phase and Polarity Configuration in Slave Mode

| Control Bit | Mode 0 | Mode 1 | Mode 2 | Mode 3 |
|---|---|---|---|---|
| SPI_TSCK_I_EDGE | 0 | 1 | 1 | 0 |
| SPI_RSCK_I_EDGE | 0 | 1 | 1 | 0 |
| SPI_CLK_MODE_13 | 0 | 1 | 0 | 1 |

## 27.8 GP-SPI2 Timing Compensation

**Introduction**

The I/O lines are mapped via GPIO Matrix or IO MUX. But there is no timing adjustment in IO MUX. The input data and output data can be delayed for 1 or 2 APB_CLK cycles at the rising or falling edge in GPIO matrix. For detailed register configuration, see Chapter *5 IO MUX and GPIO Matrix (GPIO, IO MUX)*.

Figure 27.8-1 shows the timing compensation control for GP-SPI2 master mode, including the following paths:

- "CLK": the output path of GP-SPI2 bus clock. The clock is sent out by SPI_CLK out control module, passes through GPIO Matrix or IO MUX and then goes to an external SPI device.
- "IN": data input path of GP-SPI2. The input data from an external SPI device passes through GPIO Matrix or IO MUX, then is adjusted by the Timing Module and finally is stored into spi_rx_afifo.
- "OUT": data output path of GP-SPI2. The output data is sent out to the Timing Module, passes through GPIO Matrix or IO MUX and is then captured by an external SPI device.

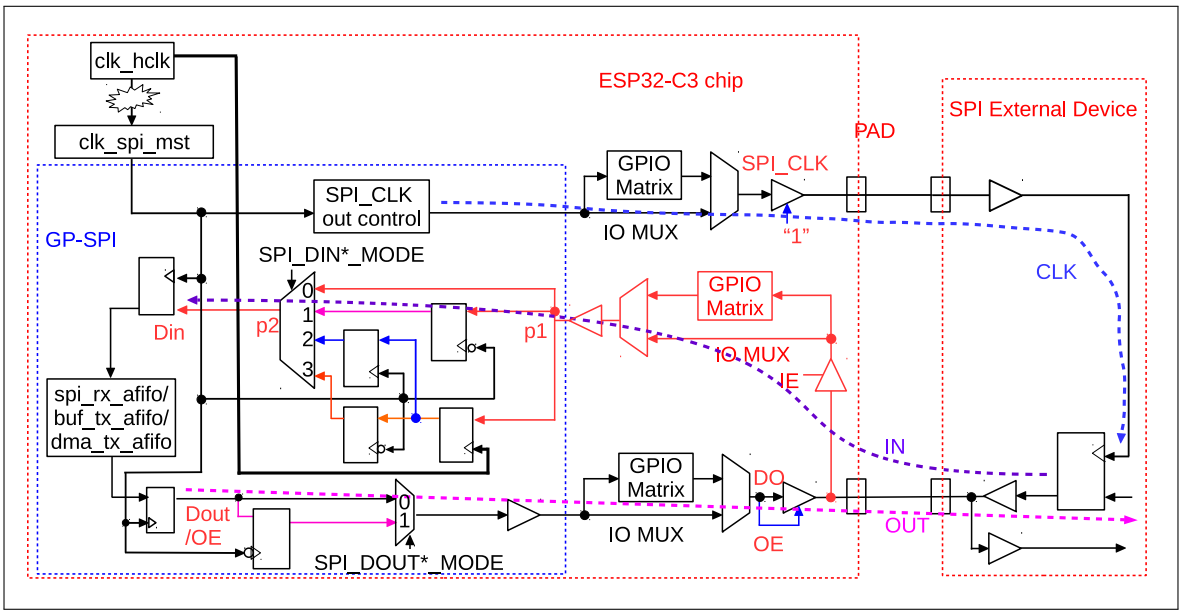



Figure 27.8-1. Timing Compensation Control Diagram in GP-SPI2 Master Mode

Every input and output data is passing through the Timing Module and the module can be used to apply delay in units of $T_{\text{clk_spi_mst}}$ (one cycle of clk_spi_mst) on rising or falling edge.

### Key Registers

- SPI_DIN_MODE_REG: select the latch edge of input data
- SPI_DIN_NUM_REG: select the delay cycles of input data
- SPI_DOUT_MODE_REG: select the latch edge of output data

### Timing Compensation Example

Figure 27.8-2 shows a timing compensation example in GP-SPI2 master mode. Note that DUMMY cycle length is configurable to compensate the delay in I/O lines, so as to enhance the performance of GP-SPI2.

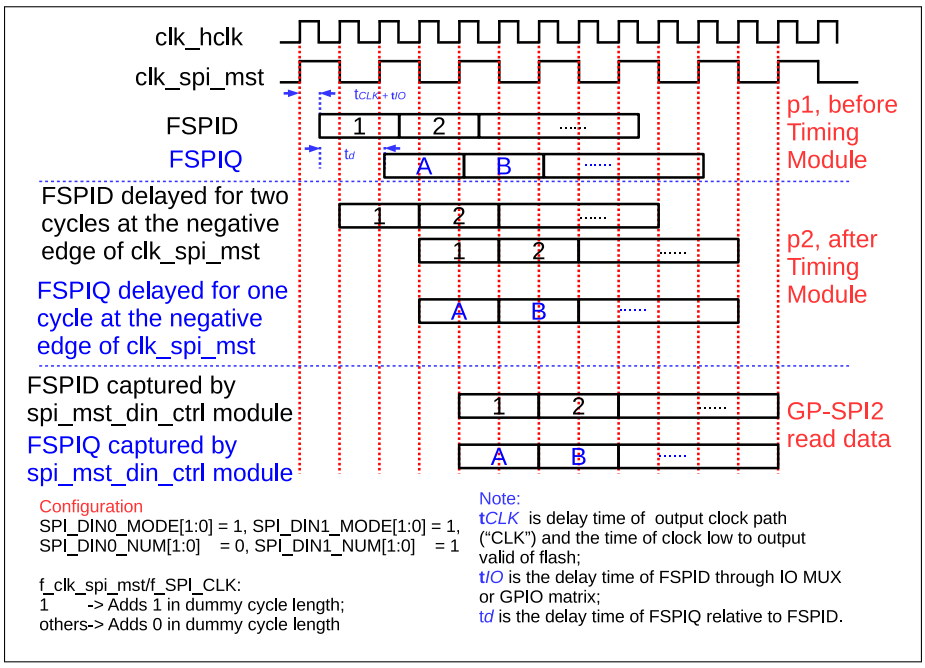



Figure 27.8-2. Timing Compensation Example in GP-SPI2 Master Mode

In Figure 27.8-2, "p1" is the point of input data of Timing Module, "p2" is the point of output data of Timing Module. Since the input data FSPIQ is unaligned to FSPID, the read data of GP-SPI2 will be wrong without the timing compensation.

To get correct read data, follow the the settings below, assuing $f_{clk_spi_mst}$ equals to $f_{SPI_CLK}$:

- Delay FSPID for two cycles at the falling edge of clk_spi_mst.
- Delay FSPIQ for one cycle at the falling edge of clk_spi_mst.
- Add one extra dummy cycle.

In GP-SPI2 slave mode, if the bit SPI_RSCK_DATA_OUT in register SPI_SLAVE_REG is set to 1, the output data is sent at latch edge, which is half an SPI clock cycle earlier. This can be used for slave mode timing compensation.

## 27.9 Interrupts

### Interrupt Summary

GP-SPI2 provides an SPI interface interrupt SPI_INT. When an SPI transfer ends, an interrupt is generated in GP-SPI2. The interrupt may be one or more of the following ones:

- SPI_DMA_INFIFO_FULL_ERR_INT: triggered when GDMA RX FIFO length is shorter than the real transferred data length.
- SPI_DMA_OUTFIFO_EMPTY_ERR_INT: triggered when GDMA TX FIFO length is shorter than the real transferred data length.
- SPI_SLV_EX_QPI_INT: triggered when Ex_QPI is received correctly in GP-SPI2 slave mode and the SPI transfer ends.

- SPI_SLV_EN_QPI_INT: triggered when En_QPI is received correctly in GP-SPI2 slave mode and the SPI transfer ends.
- SPI_SLV_CMD7_INT: triggered when CMD7 is received correctly in GP-SPI2 slave mode and the SPI transfer ends.
- SPI_SLV_CMD8_INT: triggered when CMD8 is received correctly in GP-SPI2 slave mode and the SPI transfer ends.
- SPI_SLV_CMD9_INT: triggered when CMD9 is received correctly in GP-SPI2 slave mode and the SPI transfer ends.
- SPI_SLV_CMDA_INT: triggered when CMDA is received correctly in GP-SPI2 slave mode and the SPI transfer ends.
- SPI_SLV_RD_DMA_DONE_INT: triggered at the end of Rd_DMA transfer in slave mode.
- SPI_SLV_WR_DMA_DONE_INT: triggered at the end of Wr_DMA transfer in slave mode.
- SPI_SLV_RD_BUF_DONE_INT: triggered at the end of Rd_BUF transfer in slave mode.
- SPI_SLV_WR_BUF_DONE_INT: triggered at the end of Wr_BUF transfer in slave mode.
- SPI_TRANS_DONE_INT: triggered at the end of SPI bus transfer in both master and slave modes.
- SPI_DMA_SEG_TRANS_DONE_INT: triggered at the end of End_SEG_TRANS transfer in GP-SPI2 slave segmented transfer mode or at the end of configurable segmented transfer in master mode.
- SPI_SEG_MAGIC_ERR_INT: triggered when a Magic error occurs in CONF buffer during configurable segmented transfer in master mode.
- SPI_MST_RX_AFIFO_WFULL_ERR_INT: triggered by RX AFIFO write-full error in GP-SPI2 master mode.
- SPI_MST_TX_AFIFO_REMPTY_ERR_INT: triggered by TX AFIFO read-empty error in GP-SPI2 master mode.
- SPI_SLV_CMD_ERR_INT: triggered when a received command value is not supported in GP-SPI2 slave mode.
- SPI_APP2_INT: used and triggered by software. It is only used for user defined function.
- SPI_APP1_INT: used and triggered by software. It is only used for user defined function.

#### Interrupts Used in Master and Slave Modes

Table 27.9-1 and Table 27.9-2 show the interrupts used in GP-SPI2 master and slave modes. Set the interrupt enable bit SPI_*_INT_ENA in SPI_DMA_INT_ENA_REG and wait for the SPI_INT interrupt. When the transfer ends, the related interrupt is triggered and should be cleared by software before the next transfer.

Table 27.9-1. GP-SPI2 Master Mode Interrupts

| Transfer Type | Communication Mode | Controlled by | Interrupt |
|---|---|---|---|
| Single Transfer | Full-duplex | DMA | GDMA_IN_SUC_EOF_CH*n*_INT [1] |
| | | CPU | SPI_TRANS_DONE_INT [2] |
| | Half-duplex MOSI Mode | DMA | SPI_TRANS_DONE_INT |
| | | CPU | SPI_TRANS_DONE_INT |
| | Half-duplex MISO Mode | DMA | GDMA_IN_SUC_EOF_CH*n*_INT |

Table 27.9-1. GP-SPI2 Master Mode Interrupts

| Transfer Type | Communication Mode | Controlled by | Interrupt |
|---|---|---|---|
| | | CPU | SPI_TRANS_DONE_INT |
| Configurable Segmented Transfer | Full-duplex | DMA | SPI_DMA_SEG_TRANS_DONE_INT [3] |
| | | CPU | Not supported |
| | Half-duplex MOSI Mode | DMA | SPI_DMA_SEG_TRANS_DONE_INT |
| | | CPU | Not supported |
| | Half-duplex MISO | DMA | SPI_DMA_SEG_TRANS_DONE_INT |
| | | CPU | Not supported |

**Note:**

1. If GDMA_IN_SUC_EOF_CH*n*_INT is triggered, it means all the RX data of GP-SPI2 has been stored in the RX buffer, and the TX data has been transferred to the slave.
2. SPI_TRANS_DONE_INT is triggered when CS is high, which indicates that master has completed the data exchange in SPI_W0_REG ~ SPI_W15_REG with slave in this mode.
3. If SPI_DMA_SEG_TRANS_DONE_INT is triggered, it means that the whole configurable segmented transfer (consisting of several segments) has finished, i.e. the RX data has been stored in the RX buffer completely and all the TX data has been sent out.

Table 27.9-2. GP-SPI2 Slave Mode Interrupts

| Transfer Type | Communication Mode | Controlled by | Interrupt |
|---|---|---|---|
| Single Transfer | Full-duplex | DMA | GDMA_IN_SUC_EOF_CH*n*_INT [1] |
| | | CPU | SPI_TRANS_DONE_INT [2] |
| | Half-duplex MOSI Mode | DMA (Wr_DMA) | GDMA_IN_SUC_EOF_CH*n*_INT[3] |
| | | CPU (Wr_BUF) | SPI_TRANS_DONE_INT[4] |
| | Half-duplex MISO Mode | DMA (Rd_DMA) | SPI_TRANS_DONE_INT[5] |
| | | CPU (Rd_BUF) | SPI_TRANS_DONE_INT[6] |
| Slave Segmented Transfer | Full-duplex | DMA | GDMA_IN_SUC_EOF_CH*n*_INT[7] |
| | | CPU | Not supported[8] |
| | Half-duplex MOSI Mode | DMA (Wr_DMA) | SPI_DMA_SEG_TRANS_DONE_INT[9] |
| | | CPU (Wr_BUF) | Not supported[10] |
| | Half-duplex MISO Mode | DMA (Rd_DMA) | SPI_DMA_SEG_TRANS_DONE_INT[11] |
| | | CPU (Rd_BUF) | Not supported[12] |

**Note:**

1. If GDMA_IN_SUC_EOF_CH*n*_INT is triggered, it means all the RX data has been stored in the RX buffer, and the TX data has been sent to the slave.
2. SPI_TRANS_DONE_INT is triggered when CS is high, which indicates that master has completed the data exchange in SPI_W0_REG ~ SPI_W15_REG with slave in this mode.
3. SPI_SLV_WR_DMA_DONE_INT just means that the transmission on the SPI bus is done, but can not ensure that all the push data has been stored in the RX buffer. For this reason, GDMA_IN_SUC_EOF_CH*n*_INT is recommended.
4. Or wait for SPI_SLV_WR_BUF_DONE_INT.

5. Or wait for SPI_SLV_RD_DMA_DONE_INT.
6. Or wait for SPI_SLV_RD_BUF_DONE_INT.
7. Slave should set the total read data byte length in SPI_MS_DATA_BITLEN before the transfer begins. And set SPI_RX_EOF_EN 0→1 before the end of the interrupt program.
8. Master and slave should define a method to end the segmented transfer, such as via GPIO interrupt and so on.
9. Master sends End_SEG_TRAN to end the segmented transfer or slave sets the total read data byte length in SPI_MS_DATA_BITLEN and waits for GDMA_IN_SUC_EOF_CH*n*_INT.
10. Half-duplex Wr_BUF single transfer can be used in a DMA-controlled segmented transfer.
11. Master sends End_SEG_TRAN to end the segmented transfer.
12. Half-duplex Rd_BUF single transfer can be used in a DMA-controlled segmented transfer.

## 27.10 Register Summary

The addresses in this section are relative to SPI base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

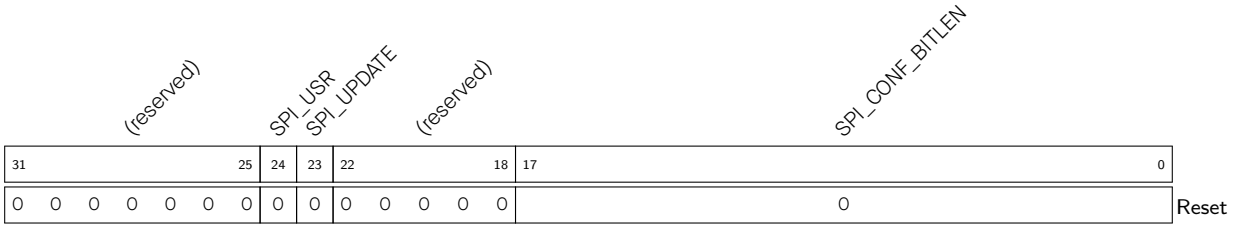
| Name | Description | Address | Access |
|---|---|---|---|
| **User-defined control registers** | | | |
| SPI_CMD_REG | Command control register | 0x0000 | varies |
| SPI_ADDR_REG | Address value register | 0x0004 | R/W |
| SPI_USER_REG | SPI USER control register | 0x0010 | varies |
| SPI_USER1_REG | SPI USER control register 1 | 0x0014 | R/W |
| SPI_USER2_REG | SPI USER control register 2 | 0x0018 | R/W |
| **Control and configuration registers** | | | |
| SPI_CTRL_REG | SPI control register | 0x0008 | R/W |
| SPI_MS_DLEN_REG | SPI data bit length control register | 0x001C | R/W |
| SPI_MISC_REG | SPI MISC register | 0x0020 | R/W |
| SPI_DMA_CONF_REG | SPI DMA control register | 0x0030 | varies |
| SPI_SLAVE_REG | SPI slave control register | 0x00E0 | varies |
| SPI_SLAVE1_REG | SPI slave control register 1 | 0x00E4 | R/W/SS |
| **Clock control registers** | | | |
| SPI_CLOCK_REG | SPI clock control register | 0x000C | R/W |
| SPI_CLK_GATE_REG | SPI module clock and register clock control | 0x00E8 | R/W |
| **Timing registers** | | | |
| SPI_DIN_MODE_REG | SPI input delay mode configuration | 0x0024 | R/W |
| SPI_DIN_NUM_REG | SPI input delay number configuration | 0x0028 | R/W |
| SPI_DOUT_MODE_REG | SPI output delay mode configuration | 0x002C | R/W |
| **Interrupt registers** | | | |
| SPI_DMA_INT_ENA_REG | SPI DMA interrupt enable register | 0x0034 | R/W |
| SPI_DMA_INT_CLR_REG | SPI DMA interrupt clear register | 0x0038 | WT |
| SPI_DMA_INT_RAW_REG | SPI DMA interrupt raw register | 0x003C | varies |
| SPI_DMA_INT_ST_REG | SPI DMA interrupt status register | 0x0040 | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| CPU-controlled data buffer | | | |
| SPI_W0_REG | SPI CPU-controlled buffer 0 | 0x0098 | R/W/SS |
| SPI_W1_REG | SPI CPU-controlled buffer 1 | 0x009C | R/W/SS |
| SPI_W2_REG | SPI CPU-controlled buffer 2 | 0x00A0 | R/W/SS |
| SPI_W3_REG | SPI CPU-controlled buffer 3 | 0x00A4 | R/W/SS |
| SPI_W4_REG | SPI CPU-controlled buffer 4 | 0x00A8 | R/W/SS |
| SPI_W5_REG | SPI CPU-controlled buffer 5 | 0x00AC | R/W/SS |
| SPI_W6_REG | SPI CPU-controlled buffer 6 | 0x00B0 | R/W/SS |
| SPI_W7_REG | SPI CPU-controlled buffer 7 | 0x00B4 | R/W/SS |
| SPI_W8_REG | SPI CPU-controlled buffer 8 | 0x00B8 | R/W/SS |
| SPI_W9_REG | SPI CPU-controlled buffer 9 | 0x00BC | R/W/SS |
| SPI_W10_REG | SPI CPU-controlled buffer 10 | 0x00C0 | R/W/SS |
| SPI_W11_REG | SPI CPU-controlled buffer 11 | 0x00C4 | R/W/SS |
| SPI_W12_REG | SPI CPU-controlled buffer 12 | 0x00C8 | R/W/SS |
| SPI_W13_REG | SPI CPU-controlled buffer 13 | 0x00CC | R/W/SS |
| SPI_W14_REG | SPI CPU-controlled buffer 14 | 0x00D0 | R/W/SS |
| SPI_W15_REG | SPI CPU-controlled buffer 15 | 0x00D4 | R/W/SS |
| Version register | | | |
| SPI_DATE_REG | Version control | 0x00F0 | R/W |

## 27.11 Registers

The addresses in this section are relative to SPI base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 27.1. SPI_CMD_REG (0x0000)**

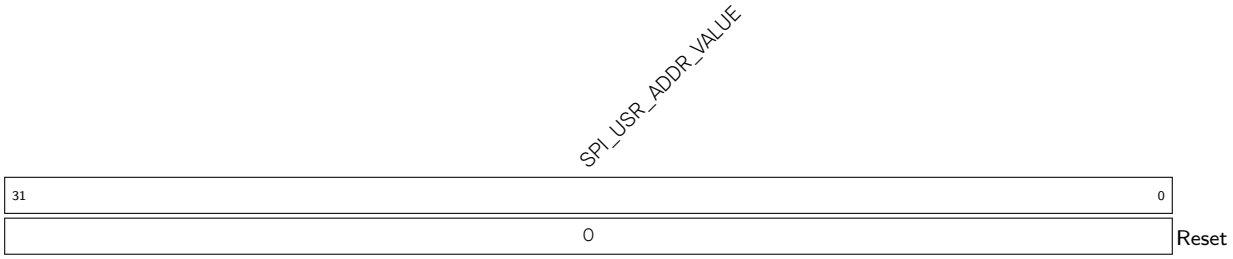



**SPI_CONF_BITLEN** Define the SPI CLK cycles of SPI CONF state. Can be configured in CONF state. (R/W)

**SPI_UPDATE** Set this bit to synchronize SPI registers from APB clock domain into SPI module clock domain. This bit is only used in SPI master mode. (WT)

**SPI_USR** User-defined command enable. An SPI operation will be triggered when the bit is set. The bit will be cleared once the operation is done. 1: enable; 0: disable. Can not be changed by CONF_buf. (R/W/SC)

**Register 27.2. SPI_ADDR_REG (0x0004)**




**SPI_USR_ADDR_VALUE** Address to slave. Can be configured in CONF state. (R/W)

**Register 27.3. SPI_USER_REG (0x0010)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SPI_USR_COMMAND | SPI_USR_ADDR | SPI_USR_DUMMY | SPI_USR_MISO | SPI_USR_MOSI | SPI_USR_DUMMY_IDLE | SPI_USR_MOSI_HIGHPART | SPI_USR_MISO_HIGHPART | (reserved) | | | | | | SPI_SIO | (reserved) | SPI_USR_CONF_NXT | (reserved) | SPI_FWRITE_QUAD | SPI_FWRITE_DUAL | (reserved) | | SPI_CK_OUT_EDGE | SPI_RSCK_I_EDGE | SPI_CS_SETUP | SPI_CS_HOLD | SPI_TSCK_I_EDGE | (reserved) | SPI_QPI_MODE | (reserved) | | | SPI_DOUTDIN | |
| 1 | 0 | 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 | 0 | 0 | 0 | 0 | Reset |

**SPI_DOUTDIN** Set the bit to enable full-duplex communication. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_QPI_MODE** 1: Enable QPI mode. 0: Disable QPI mode. This configuration is applicable when the SPI controller works as master or slave. Can be configured in CONF state. (R/W/SS/SC)

**SPI_TSCK_I_EDGE** In slave mode, this bit can be used to change the polarity of TSCK. 0: TSCK = SPI_CK_I. 1: TSCK = !SPI_CK_I. (R/W)

**SPI_CS_HOLD** Keep SPI CS low when SPI is in DONE state. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_CS_SETUP** Enable SPI CS when SPI is in prepare (PREP) state. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_RSCK_I_EDGE** In slave mode, this bit can be used to change the polarity of RSCK. 0: RSCK = !SPI_CK_I. 1: RSCK = SPI_CK_I. (R/W)

**SPI_CK_OUT_EDGE** This bit together with SPI_CK_IDLE_EDGE is used to control SPI clock mode. Can be configured in CONF state. For more information, see Section 27.7.2. (R/W)

**SPI_FWRITE_DUAL** In write operations, read-data phase is in 2-bit mode. Can be configured in CONF state. (R/W)

**SPI_FWRITE_QUAD** In write operations, read-data phase is in 4-bit mode. Can be configured in CONF state. (R/W)

**SPI_USR_CONF_NXT** Enable the CONF state for the next transaction (segment) in a configurable segmented transfer. Can be configured in CONF state. (R/W)

- If this bit is set, it means this configurable segmented transfer will continue its next transaction (segment).
- If this bit is cleared, it means this transfer will end after the current transaction (segment) is finished. Or this is not a configurable segmented transfer.

**SPI_SIO** Set the bit to enable 3-line half-duplex communication, where MOSI and MISO signals share the same pin. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_USR_MISO_HIGHPART** In read-data phase, only access to high-part of the buffers: SPI_W8_REG ~ SPI_W15_REG. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

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

Register 27.3. SPI_USER_REG (0x0010)

Continued from the previous page...

**SPI_USR_MOSI_HIGHPART** In write-data phase, only access to high-part of the buffers: SPI_W8_REG ~ SPI_W15_REG. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_USR_DUMMY_IDLE** If this bit is set, SPI clock is disabled in DUMMY state. Can be configured in CONF state. (R/W)

**SPI_USR_MOSI** Set this bit to enable the write-data (DOUT) state of an operation. Can be configured in CONF state. (R/W)

**SPI_USR_MISO** Set this bit to enable the read-data (DIN) state of an operation. Can be configured in CONF state. (R/W)

**SPI_USR_DUMMY** Set this bit to enable the DUMMY state of an operation. Can be configured in CONF state. (R/W)

**SPI_USR_ADDR** Set this bit to enable the address (ADDR) state of an operation. Can be configured in CONF state. (R/W)

**SPI_USR_COMMAND** Set this bit to enable the command (CMD) state of an operation. Can be configured in CONF state. (R/W)

**Register 27.4. SPI_USER1_REG (0x0014)**

| SPI_USR_ADDR_BITLEN | SPI_CS_HOLD_TIME | SPI_CS_SETUP_TIME | SPI_MST_WFULL_ERR_END_EN | (reserved) | SPI_USR_DUMMY_CYCLELEN | |
|---|---|---|---|---|---|---|
| 31 27 | 26 22 | 21 17 | 16 | 15 8 | 7 0 | |
| 23 | 0x1 | 0 | 1 | 0 0 0 0 0 0 0 0 | 7 | Reset |

**SPI_USR_DUMMY_CYCLELEN** The length of DUMMY state, in unit of SPI_CLK cycles. This value is (the expected cycle number - 1). Can be configured in CONF state. (R/W)

**SPI_MST_WFULL_ERR_END_EN** 1: SPI transfer is ended when SPI RX AFIFO wfull error occurs in GP-SPI master full-/half-duplex modes. 0: SPI transfer is not ended when SPI RX AFIFO wfull error occurs in GP-SPI master full-/half-duplex modes. (R/W)

**SPI_CS_SETUP_TIME** The length of prepare (PREP) state, in unit of SPI_CLK cycles. This value is equal to the expected cycles - 1. This field is used together with SPI_CS_SETUP. Can be configured in CONF state. (R/W)

**SPI_CS_HOLD_TIME** Delay cycles of CS pin, in units of SPI_CLK cycles. This field is used together with SPI_CS_HOLD. Can be configured in CONF state. (R/W)

**SPI_USR_ADDR_BITLEN** The bit length in address state. This value is (expected bit number - 1). Can be configured in CONF state. (R/W)

**Register 27.5. SPI_USER2_REG (0x0018)**

| SPI_USR_COMMAND_BITLEN | SPI_MST_REMPTY_ERR_END_EN | (reserved) | SPI_USR_COMMAND_VALUE | |
|---|---|---|---|---|
| 31 28 | 27 | 26 16 | 15 0 | |
| 7 | 1 | 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**SPI_USR_COMMAND_VALUE** The value of command. Can be configured in CONF state. (R/W)

**SPI_MST_REMPTY_ERR_END_EN** 1: SPI transfer is ended when SPI TX AFIFO read empty error occurs in GP-SPI master full-/half-duplex modes. 0: SPI transfer is not ended when SPI TX AFIFO read empty error occurs in GP-SPI master full-/half-duplex modes. (R/W)

**SPI_USR_COMMAND_BITLEN** The bit length of command state. This value is (expected bit number - 1). Can be configured in CONF state. (R/W)

**Register 27.6. SPI_CTRL_REG (0x0008)**

| (reserved) | SPI_WR_BIT_ORDER | SPI_RD_BIT_ORDER | (reserved) | SPI_WP_POL | SPI_HOLD_POL | SPI_D_POL | SPI_Q_POL | (reserved) | SPI_FREAD_QUAD | SPI_FREAD_DUAL | (reserved) | SPI_FCMD_QUAD | SPI_FCMD_DUAL | (reserved) | SPI_FADDR_QUAD | SPI_FADDR_DUAL | (reserved) | SPI_DUMMY_OUT | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 27 | 26 | 25 | 24 22 | 21 | 20 | 19 | 18 | 17 16 | 15 | 14 | 13 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 0 | |
| 0 0 0 0 0 | 0 | 0 | 0 0 0 | 1 | 1 | 1 | 1 | 0 0 | 0 | 0 | 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 0 0 | Reset |

**SPI_DUMMY_OUT** Configure the output signal level in DUMMY state. Can be configured in CONF state. (R/W)

**SPI_FADDR_DUAL** Apply 2-bit mode during address (ADDR) state. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_FADDR_QUAD** Apply 4-bit mode during address (ADDR) state. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_FCMD_DUAL** Apply 2-bit mode during command (CMD) state. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_FCMD_QUAD** Apply 4-bit mode during command (CMD) state. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_FREAD_DUAL** In read operations, read-data (DIN) state is in 2-bit mode. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_FREAD_QUAD** In read operations, read-data (DIN) state is in 4-bit mode. 1: enable; 0: disable. Can be configured in CONF state. (R/W)

**SPI_Q_POL** This bit is used to set MISO line polarity. 1: high; 0: low. Can be configured in CONF state. (R/W)

**SPI_D_POL** This bit is used to set MOSI line polarity. 1: high; 0: low. Can be configured in CONF state. (R/W)

**SPI_HOLD_POL** This bit is used to set SPI_HOLD output value when SPI is in idle. 1: output high; 0: output low. Can be configured in CONF state. (R/W)

**SPI_WP_POL** This bit is to set the output value of write-protect signal when SPI is in idle. 1: output high; 0: output low. Can be configured in CONF state. (R/W)

**SPI_RD_BIT_ORDER** In read-data (MISO) state, 1: LSB first; 0: MSB first. Can be configured in CONF state. (R/W)

**SPI_WR_BIT_ORDER** In command (CMD), address (ADDR), and write-data (MOSI) states, 1: LSB first; 0: MSB first. Can be configured in CONF state. (R/W)

**Register 27.7. SPI_MS_DLEN_REG (0x001C)**

| (reserved) | SPI_MS_DATA_BITLEN | |
|---|---|---|
| 31 … 18 | 17 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**SPI_MS_DATA_BITLEN** The value of this field is the configured SPI transmission data bit length in master mode DMA-controlled transfer or CPU-controlled transfer. The value is also the configured bit length in slave mode DMA RX controlled transfer. The register value shall be (bit_num - 1). Can be configured in CONF state. (R/W)

**Register 27.8. SPI_MISC_REG (0x0020)**

| SPI_QUAD_DIN_PIN_SWAP | SPI_CS_KEEP_ACTIVE | SPI_CK_IDLE_EDGE | (reserved) | SPI_SLAVE_CS_POL | (reserved) | SPI_MASTER_CS_POL | SPI_CK_DIS | SPI_CS5_DIS | SPI_CS4_DIS | SPI_CS3_DIS | SPI_CS2_DIS | SPI_CS1_DIS | SPI_CS0_DIS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 24 | 23 | 22 13 | 12 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 | 1 | 1 | 1 | 0 | Reset |

**SPI_CS0_DIS** SPI CS0 pin enable bit. 1: disable CS0, 0: SPI_CS0 signal is from/to CS0 pin. Can be configured in CONF state. (R/W)

**SPI_CS1_DIS** SPI CS1 pin enable bit. 1: disable CS1, 0: SPI_CS1 signal is from/to CS1 pin. Can be configured in CONF state. (R/W)

**SPI_CS2_DIS** SPI CS2 pin enable bit. 1: disable CS2, 0: SPI_CS2 signal is from/to CS2 pin. Can be configured in CONF state. (R/W)

**SPI_CS3_DIS** SPI CS3 pin enable bit. 1: disable CS3, 0: SPI_CS3 signal is from/to CS3 pin. Can be configured in CONF state. (R/W)

**SPI_CS4_DIS** SPI CS4 pin enable bit. 1: disable CS4, 0: SPI_CS4 signal is from/to CS4 pin. Can be configured in CONF state. (R/W)

**SPI_CS5_DIS** SPI CS5 pin enable bit. 1: disable CS5, 0: SPI_CS5 signal is from/to CS5 pin. Can be configured in CONF state. (R/W)

**SPI_CK_DIS** 1: disable SPI_CLK output. 0: enable SPI_CLK output. Can be configured in CONF state. (R/W)

**SPI_MASTER_CS_POL** In master mode, the bits are the polarity of SPI CS line, the value is equivalent to SPI_CS ^ SPI_MASTER_CS_POL. Can be configured in CONF state. (R/W)

**SPI_SLAVE_CS_POL** Configure SPI slave input CS polarity. 1: invert. 0: not change. Can be configured in CONF state. (R/W)

**SPI_CK_IDLE_EDGE** 1: SPI_CLK line is high when GP-SPI2 is in idle. 0: SPI_CLK line is low when GP-SPI2 is in idle. Can be configured in CONF state. (R/W)

**SPI_CS_KEEP_ACTIVE** SPI CS line keeps low when the bit is set. Can be configured in CONF state. (R/W)

**SPI_QUAD_DIN_PIN_SWAP** 1: SPI quad input swap enable. 0: SPI quad input swap disable. Can be configured in CONF state. (R/W)

**Register 27.9. SPI_DMA_CONF_REG (0x0030)**

| 31 | 30 | 29 | 28 | 27 | 26–22 | 21 | 20 | 19 | 18 | 17–0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SPI_DMA_AFIFO_RST | SPI_BUF_AFIFO_RST | SPI_RX_AFIFO_RST | SPI_DMA_TX_ENA | SPI_DMA_RX_ENA | (reserved) | SPI_RX_EOF_EN | SPI_SLV_TX_SEG_TRANS_CLR_EN | SPI_SLV_RX_SEG_TRANS_CLR_EN | SPI_DMA_SLV_SEG_TRANS_EN | (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 | Reset |

**SPI_DMA_SLV_SEG_TRANS_EN** 1: enable DAM-controlled segmented transfer in slave half-duplex mode. 0: disable. (R/W)

**SPI_SLV_RX_SEG_TRANS_CLR_EN** In DMA-controlled half-duplex slave mode, if the size of DMA RX buffer is smaller than the size of the received data, 1: the data in following transfers will not be received. 0: the data in this transfer will not be received, but in the following transfers, if the size of DMA RX buffer is not 0, the data in following transfers will be received, otherwise not. (R/W)

**SPI_SLV_TX_SEG_TRANS_CLR_EN** In DMA-controlled half-duplex slave mode, if the size of DMA TX buffer is smaller than the size of the transmitted data, 1: the data in the following transfers will not be updated, i.e. the old data is transmitted repeatedly. 0: the data in this transfer will not be updated. But in the following transfers, if new data is filled in DMA TX FIFO, new data will be transmitted, otherwise not. (R/W)

**SPI_RX_EOF_EN** 1: In a DAM-controlled transfer, if the bit number of transferred data is equal to (SPI_MS_DATA_BITLEN + 1), then GDMA_IN_SUC_EOF_CH*n*_INT_RAW will be set by hardware. 0: GDMA_IN_SUC_EOF_CH*n*_INT_RAW is set by SPI_TRANS_DONE_INT event in a non-segmented transfer, or by in a SPI_DMA_SEG_TRANS_DONE_INT event in a segmented transfer. (R/W)

**SPI_DMA_RX_ENA** Set this bit to enable SPI DMA controlled receive data mode. (R/W)

**SPI_DMA_TX_ENA** Set this bit to enable SPI DMA controlled send data mode. (R/W)

**SPI_RX_AFIFO_RST** Set this bit to reset spi_rx_afifo as shown in Figure 27.5-3 and in Figure 27.5-4. spi_rx_afifo is used to receive data in SPI master and slave transfer. (WT)

**SPI_BUF_AFIFO_RST** Set this bit to reset buf_tx_afifo as shown in Figure 27.5-3 and in Figure 27.5-4. buf_tx_afifo is used to send data out in CPU-controlled master and slave transfer. (WT)

**SPI_DMA_AFIFO_RST** Set this bit to reset dma_tx_afifo as shown in Figure 27.5-3 and in Figure 27.5-4. dma_tx_afifo is used to send data out in DMA-controlled slave transfer. (WT)

**Register 27.10. SPI_SLAVE_REG (0x00E0)**

| (reserved) | SPI_USR_CONF | SPI_SOFT_RESET | SPI_SLAVE_MODE | SPI_DMA_SEG_MAGIC_VALUE | (reserved) | SPI_SLV_WRBUF_BITLEN_EN | SPI_SLV_RDBUF_BITLEN_EN | SPI_SLV_WRDMA_BITLEN_EN | SPI_SLV_RDDMA_BITLEN_EN | (reserved) | SPI_RSCK_DATA_OUT | SPI_CLK_MODE_13 | SPI_CLK_MODE | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 29 | 28 | 27 | 26 | 25 22 | 21 12 | 11 | 10 | 9 | 8 | 7 4 | 3 | 2 | 1 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 | 0 | 0 | 0 | Reset |

**SPI_CLK_MODE** SPI clock mode control bits. Can be configured in CONF state. (R/W)

- 0: SPI clock is off when CS becomes inactive.
- 1: SPI clock is delayed one cycle after CS becomes inactive.
- 2: SPI clock is delayed two cycles after CS becomes inactive.
- 3: SPI clock is always on.

**SPI_CLK_MODE_13** Configure clock mode. (R/W)

- 1: support SPI clock mode 1 and 3. Output data B[0]/B[7] at the first edge.
- 0: support SPI clock mode 0 and 2. Output data B[1]/B[6] at the first edge.

**SPI_RSCK_DATA_OUT** Save half a cycle when TSCK is the same as RSCK. 1: output data at RSCK posedge. 0: output data at TSCK posedge. (R/W)

**SPI_SLV_RDDMA_BITLEN_EN** If this bit is set, SPI_SLV_DATA_BITLEN is used to store the data bit length of Rd_DMA transfer. (R/W)

**SPI_SLV_WRDMA_BITLEN_EN** If this bit is set, SPI_SLV_DATA_BITLEN is used to store the data bit length of Wr_DMA transfer. (R/W)

**SPI_SLV_RDBUF_BITLEN_EN** If this bit is set, SPI_SLV_DATA_BITLEN is used to store data bit length of Rd_BUF transfer. (R/W)

**SPI_SLV_WRBUF_BITLEN_EN** If this bit is set, SPI_SLV_DATA_BITLEN is used to store data bit length of Wr_BUF transfer. (R/W)

**SPI_DMA_SEG_MAGIC_VALUE** Configure the magic value of BM table in DMA-controlled configurable segmented transfer. (R/W)

**SPI_SLAVE_MODE** Set SPI work mode. 1: slave mode. 0: master mode. (R/W)

**SPI_SOFT_RESET** Software reset enable bit. If this bit is set, the SPI clock line, CS line, and data line are reset. Can be configured in CONF state. (WT)

**SPI_USR_CONF** 1: enable the CONF state of current DMA-controlled configurable segmented transfer, which means the configurable segmented transfer is started. 0: This is not a configurable segmented transfer. (R/W)

Register 27.11. SPI_SLAVE1_REG (0x00E4)

| SPI_SLV_LAST_ADDR | SPI_SLV_LAST_COMMAND | SPI_SLV_DATA_BITLEN | |
|---|---|---|---|
| 31 26 | 25 18 | 17 0 | |
| 0 | 0 | 0 | Reset |

**SPI_SLV_DATA_BITLEN** Configure the transferred data bit length in SPI slave full-/half-duplex modes. (R/W/SS)

**SPI_SLV_LAST_COMMAND** In slave mode, it is the value of command. (R/W/SS)

**SPI_SLV_LAST_ADDR** In slave mode, it is the value of address. (R/W/SS)

Register 27.12. SPI_CLOCK_REG (0x000C)

| SPI_CLK_EQU_SYSCLK | (reserved) | SPI_CLKDIV_PRE | SPI_CLKCNT_N | SPI_CLKCNT_H | SPI_CLKCNT_L | |
|---|---|---|---|---|---|---|
| 31 | 30 22 | 21 18 | 17 12 | 11 6 | 5 0 | |
| 1 | 0 0 0 0 0 0 0 0 0 | 0 | 0x3 | 0x1 | 0x3 | Reset |

**SPI_CLKCNT_L** In master mode, this field must be equal to SPI_CLKCNT_N. In slave mode, it must be 0. Can be configured in CONF state. (R/W)

**SPI_CLKCNT_H** In master mode, this field must be floor((SPI_CLKCNT_N + 1)/2 - 1). floor() here is to down round a number, floor(2.2) = 2. In slave mode, it must be 0. Can be configured in CONF state. (R/W)

**SPI_CLKCNT_N** In master mode, this is the divider of SPI_CLK. So SPI_CLK frequency is $f_{apb_clk}$/(SPI_CLKDIV_PRE + 1)/(SPI_CLKCNT_N + 1). Can be configured in CONF state. (R/W)

**SPI_CLKDIV_PRE** In master mode, this is pre-divider of SPI_CLK. Can be configured in CONF state. (R/W)

**SPI_CLK_EQU_SYSCLK** In master mode, 1: SPI_CLK is eqaul to APB_CLK. 0: SPI_CLK is divided from APB_CLK. Can be configured in CONF state. (R/W)

**Register 27.13. SPI_CLK_GATE_REG (0x00E8)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | SPI_MST_CLK_SEL | SPI_MST_CLK_ACTIVE | SPI_CLK_EN | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**SPI_CLK_EN** Set this bit to enable clock gate. (R/W)

**SPI_MST_CLK_ACTIVE** Set this bit to power on the SPI module clock. (R/W)

**SPI_MST_CLK_SEL** This bit is used to select SPI module clock source in master mode. 1: PLL_F80M_CLK. 0: XTAL_CLK. (R/W)

**Register 27.14. SPI_DIN_MODE_REG (0x0024)**

| (reserved) | SPI_TIMING_HCLK_ACTIVE | (reserved) | SPI_DIN3_MODE | SPI_DIN2_MODE | SPI_DIN1_MODE | SPI_DIN0_MODE | |
|---|---|---|---|---|---|---|---|
| 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 0 | 0 | 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | Reset |

**SPI_DIN0_MODE** Configure the input mode for FSPID signal. Can be configured in CONF state. (R/W)

- 0: input without delay
- 1: Input at the (SPI_DIN0_NUM + 1)th falling edge of clk_spi_mst
- 2: Input at the (SPI_DIN0_NUM + 1)th rising edge of clk_hclk plus one clk_spi_mst rising edge cycle
- 3: Input at the (SPI_DIN0_NUM + 1)th rising edge of clk_hclk plus one clk_spi_mst falling edge cycle

**SPI_DIN1_MODE** Configure the input mode for FSPIQ signal. Can be configured in CONF state. (R/W)

- 0: input without delay
- 1: Input at the (SPI_DIN1_NUM + 1)th falling edge of clk_spi_mst
- 2: Input at the (SPI_DIN1_NUM + 1)th rising edge of clk_hclk plus one clk_spi_mst rising edge cycle
- 3: Input at the (SPI_DIN1_NUM + 1)th rising edge of clk_hclk plus one clk_spi_mst falling edge cycle

**SPI_DIN2_MODE** Configure the input mode for FSPIWP signal. Can be configured in CONF state. (R/W)

- 0: input without delay
- 1: Input at the (SPI_DIN2_NUM + 1)th falling edge of clk_spi_mst
- 2: Input at the (SPI_DIN2_NUM + 1)th rising edge of clk_hclk plus one clk_spi_mst rising edge cycle
- 3: Input at the (SPI_DIN2_NUM + 1)th rising edge of clk_hclk plus one clk_spi_mst falling edge cycle

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

**Register 27.14. SPI_DIN_MODE_REG (0x0024)**

**Continued from the previous page...**

**SPI_DIN3_MODE** Configure the input mode for FSPIHD signal. Can be configured in CONF state. (R/W)

- 0: input without delay
- 1: Input at the (SPI_DIN3_NUM + 1)th falling edge of clk_spi_mst
- 2: Input at the (SPI_DIN3_NUM + 1)th rising edge of clk_hclk plus one clk_spi_mst rising edge cycle
- 3: Input at the (SPI_DIN3_NUM + 1)th rising edge of clk_hclk plus one clk_spi_mst falling edge cycle

**SPI_TIMING_HCLK_ACTIVE** 1: enable HCLK (high-frequency clock) in SPI input timing module. 0: disable HCLK. Can be configured in CONF state. (R/W)

**Register 27.15. SPI_DIN_NUM_REG (0x0028)**

| (reserved) | SPI_DIN3_NUM | SPI_DIN2_NUM | SPI_DIN1_NUM | SPI_DIN0_NUM | |
|---|---|---|---|---|---|
| 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 0 0 0 | 0 | 0 | 0 | 0 | Reset |

**SPI_DIN0_NUM** Configure the delays to input signal FSPID based on the setting of SPI_DIN0_MODE. Can be configured in CONF state. (R/W)

- 0: delayed by 1 clock cycle
- 1: delayed by 2 clock cycles
- 2: delayed by 3 clock cycles
- 3: delayed by 4 clock cycles

**SPI_DIN1_NUM** Configure the delays to input signal FSPIQ based on the setting of SPI_DIN1_MODE. Can be configured in CONF state. (R/W)

- 0: delayed by 1 clock cycle
- 1: delayed by 2 clock cycles
- 2: delayed by 3 clock cycles
- 3: delayed by 4 clock cycles

**SPI_DIN2_NUM** Configure the delays to input signal FSPIWP based on the setting of SPI_DIN2_MODE. Can be configured in CONF state. (R/W)

- 0: delayed by 1 clock cycle
- 1: delayed by 2 clock cycles
- 2: delayed by 3 clock cycles
- 3: delayed by 4 clock cycles

**SPI_DIN3_NUM** Configure the delays to input signal FSPIHD based on the setting of SPI_DIN3_MODE. Can be configured in CONF state. (R/W)

- 0: delayed by 1 clock cycle
- 1: delayed by 2 clock cycles
- 2: delayed by 3 clock cycles
- 3: delayed by 4 clock cycles

**Register 27.16. SPI_DOUT_MODE_REG (0x002C)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | SPI_DOUT3_MODE | SPI_DOUT2_MODE | SPI_DOUT1_MODE | SPI_DOUT0_MODE | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**SPI_DOUT0_MODE** Configure the output mode for FSPID signal. Can be configured in CONF state. (R/W)

- 0: output without delay
- 1: output with a delay of a SPI module clock cycle at its falling edge

**SPI_DOUT1_MODE** Configure the output mode for FSPIQ signal. Can be configured in CONF state. (R/W)

- 0: output without delay
- 1: output with a delay of a SPI module clock cycle at its falling edge

**SPI_DOUT2_MODE** Configure the output mode for FSPIWP signal. Can be configured in CONF state. (R/W)

- 0: output without delay
- 1: output with a delay of a SPI module clock cycle at its falling edge

**SPI_DOUT3_MODE** Configure the output mode for FSPIHD signal. Can be configured in CONF state. (R/W)

- 0: output without delay
- 1: output with a delay of a SPI module clock cycle at its falling edge

**Register 27.17. SPI_DMA_INT_ENA_REG (0x0034)**

| 31–21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | SPI_APP1_INT_ENA | SPI_APP2_INT_ENA | SPI_MST_TX_AFIFO_REMPTY_ERR_INT_ENA | SPI_MST_RX_AFIFO_WFULL_ERR_INT_ENA | SPI_SLV_CMD_ERR_INT_ENA | (reserved) | SPI_SEG_MAGIC_ERR_INT_ENA | SPI_DMA_SEG_TRANS_DONE_INT_ENA | SPI_TRANS_DONE_INT_ENA | SPI_SLV_WR_BUF_DONE_INT_ENA | SPI_SLV_RD_BUF_DONE_INT_ENA | SPI_SLV_WR_DMA_DONE_INT_ENA | SPI_SLV_RD_DMA_DONE_INT_ENA | SPI_SLV_CMDA_INT_ENA | SPI_SLV_CMD9_INT_ENA | SPI_SLV_CMD8_INT_ENA | SPI_SLV_CMD7_INT_ENA | SPI_SLV_EN_QPI_INT_ENA | SPI_SLV_EX_QPI_INT_ENA | SPI_DMA_OUTFIFO_EMPTY_ERR_INT_ENA | SPI_DMA_INFIFO_FULL_ERR_INT_ENA | |
| 0 0 0 0 0 0 0 0 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 |

**SPI_DMA_INFIFO_FULL_ERR_INT_ENA** The enable bit for SPI_DMA_INFIFO_FULL_ERR_INT interrupt. (R/W)

**SPI_DMA_OUTFIFO_EMPTY_ERR_INT_ENA** The enable bit for SPI_DMA_OUTFIFO_EMPTY_ERR_INT interrupt. (R/W)

**SPI_SLV_EX_QPI_INT_ENA** The enable bit for SPI_SLV_EX_QPI_INT interrupt. (R/W)

**SPI_SLV_EN_QPI_INT_ENA** The enable bit for SPI_SLV_EN_QPI_INT interrupt. (R/W)

**SPI_SLV_CMD7_INT_ENA** The enable bit for SPI_SLV_CMD7_INT interrupt. (R/W)

**SPI_SLV_CMD8_INT_ENA** The enable bit for SPI_SLV_CMD8_INT interrupt. (R/W)

**SPI_SLV_CMD9_INT_ENA** The enable bit for SPI_SLV_CMD9_INT interrupt. (R/W)

**SPI_SLV_CMDA_INT_ENA** The enable bit for SPI_SLV_CMDA_INT interrupt. (R/W)

**SPI_SLV_RD_DMA_DONE_INT_ENA** The enable bit for SPI_SLV_RD_DMA_DONE_INT interrupt. (R/W)

**SPI_SLV_WR_DMA_DONE_INT_ENA** The enable bit for SPI_SLV_WR_DMA_DONE_INT interrupt. (R/W)

**SPI_SLV_RD_BUF_DONE_INT_ENA** The enable bit for SPI_SLV_RD_BUF_DONE_INT interrupt. (R/W)

**SPI_SLV_WR_BUF_DONE_INT_ENA** The enable bit for SPI_SLV_WR_BUF_DONE_INT interrupt. (R/W)

**SPI_TRANS_DONE_INT_ENA** The enable bit for SPI_TRANS_DONE_INT interrupt. (R/W)

**SPI_DMA_SEG_TRANS_DONE_INT_ENA** The enable bit for SPI_DMA_SEG_TRANS_DONE_INT interrupt. (R/W)

**SPI_SEG_MAGIC_ERR_INT_ENA** The enable bit for SPI_SEG_MAGIC_ERR_INT interrupt. (R/W)

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

Register 27.17. SPI_DMA_INT_ENA_REG (0x0034)

Continued from the previous page...

**SPI_SLV_CMD_ERR_INT_ENA** The enable bit for SPI_SLV_CMD_ERR_INT interrupt. (R/W)

**SPI_MST_RX_AFIFO_WFULL_ERR_INT_ENA** The enable bit for SPI_MST_RX_AFIFO_WFULL_ERR_INT interrupt. (R/W)

**SPI_MST_TX_AFIFO_REMPTY_ERR_INT_ENA** The enable bit for SPI_MST_TX_AFIFO_REMPTY_ERR_INT interrupt. (R/W)

**SPI_APP2_INT_ENA** The enable bit for SPI_APP2_INT interrupt. (R/W)

**SPI_APP1_INT_ENA** The enable bit for SPI_APP1_INT interrupt. (R/W)

**Register 27.18. SPI_DMA_INT_CLR_REG (0x0038)**

| 31–21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | SPI_APP1_INT_CLR | SPI_APP2_INT_CLR | SPI_MST_TX_AFIFO_REMPTY_ERR_INT_CLR | SPI_MST_RX_AFIFO_WFULL_ERR_INT_CLR | SPI_SLV_CMD_ERR_INT_CLR | (reserved) | SPI_SEG_MAGIC_ERR_INT_CLR | SPI_DMA_SEG_TRANS_DONE_INT_CLR | SPI_TRANS_DONE_INT_CLR | SPI_SLV_WR_BUF_DONE_INT_CLR | SPI_SLV_RD_BUF_DONE_INT_CLR | SPI_SLV_WR_DMA_DONE_INT_CLR | SPI_SLV_RD_DMA_DONE_INT_CLR | SPI_SLV_CMDA_INT_CLR | SPI_SLV_CMD9_INT_CLR | SPI_SLV_CMD8_INT_CLR | SPI_SLV_CMD7_INT_CLR | SPI_SLV_EN_QPI_INT_CLR | SPI_SLV_EX_QPI_INT_CLR | SPI_DMA_OUTFIFO_EMPTY_ERR_INT_CLR | SPI_DMA_INFIFO_FULL_ERR_INT_CLR | |
| 0 0 0 0 0 0 0 0 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 |

**SPI_DMA_INFIFO_FULL_ERR_INT_CLR** The clear bit for SPI_DMA_INFIFO_FULL_ERR_INT interrupt. (WT)

**SPI_DMA_OUTFIFO_EMPTY_ERR_INT_CLR** The clear bit for SPI_DMA_OUTFIFO_EMPTY_ERR_INT interrupt. (WT)

**SPI_SLV_EX_QPI_INT_CLR** The clear bit for SPI_SLV_EX_QPI_INT interrupt. (WT)

**SPI_SLV_EN_QPI_INT_CLR** The clear bit for SPI_SLV_EN_QPI_INT interrupt. (WT)

**SPI_SLV_CMD7_INT_CLR** The clear bit for SPI_SLV_CMD7_INT interrupt. (WT)

**SPI_SLV_CMD8_INT_CLR** The clear bit for SPI_SLV_CMD8_INT interrupt. (WT)

**SPI_SLV_CMD9_INT_CLR** The clear bit for SPI_SLV_CMD9_INT interrupt. (WT)

**SPI_SLV_CMDA_INT_CLR** The clear bit for SPI_SLV_CMDA_INT interrupt. (WT)

**SPI_SLV_RD_DMA_DONE_INT_CLR** The clear bit for SPI_SLV_RD_DMA_DONE_INT interrupt. (WT)

**SPI_SLV_WR_DMA_DONE_INT_CLR** The clear bit for SPI_SLV_WR_DMA_DONE_INT interrupt. (WT)

**SPI_SLV_RD_BUF_DONE_INT_CLR** The clear bit for SPI_SLV_RD_BUF_DONE_INT interrupt. (WT)

**SPI_SLV_WR_BUF_DONE_INT_CLR** The clear bit for SPI_SLV_WR_BUF_DONE_INT interrupt. (WT)

**SPI_TRANS_DONE_INT_CLR** The clear bit for SPI_TRANS_DONE_INT interrupt. (WT)

**SPI_DMA_SEG_TRANS_DONE_INT_CLR** The clear bit for SPI_DMA_SEG_TRANS_DONE_INT interrupt. (WT)

**SPI_SEG_MAGIC_ERR_INT_CLR** The clear bit for SPI_SEG_MAGIC_ERR_INT interrupt. (WT)

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

Register 27.18. SPI_DMA_INT_CLR_REG (0x0038)

Continued from the previous page...

**SPI_SLV_CMD_ERR_INT_CLR** The clear bit for SPI_SLV_CMD_ERR_INT interrupt. (WT)

**SPI_MST_RX_AFIFO_WFULL_ERR_INT_CLR** The clear bit for SPI_MST_RX_AFIFO_WFULL_ERR_INT interrupt. (WT)

**SPI_MST_TX_AFIFO_REMPTY_ERR_INT_CLR** The clear bit for SPI_MST_TX_AFIFO_REMPTY_ERR_INT interrupt. (WT)

**SPI_APP2_INT_CLR** The clear bit for SPI_APP2_INT interrupt. (WT)

**SPI_APP1_INT_CLR** The clear bit for SPI_APP1_INT interrupt. (WT)

**Register 27.19. SPI_DMA_INT_RAW_REG (0x003C)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | SPI_APP1_INT_RAW | SPI_APP2_INT_RAW | SPI_MST_TX_AFIFO_REMPTY_ERR_INT_RAW | SPI_MST_RX_AFIFO_WFULL_ERR_INT_RAW | SPI_SLV_CMD_ERR_INT_RAW | (reserved) | SPI_SEG_MAGIC_ERR_INT_RAW | SPI_DMA_SEG_TRANS_DONE_INT_RAW | SPI_TRANS_DONE_INT_RAW | SPI_SLV_WR_BUF_DONE_INT_RAW | SPI_SLV_RD_BUF_DONE_INT_RAW | SPI_SLV_WR_DMA_DONE_INT_RAW | SPI_SLV_RD_DMA_DONE_INT_RAW | SPI_SLV_CMDA_INT_RAW | SPI_SLV_CMD9_INT_RAW | SPI_SLV_CMD8_INT_RAW | SPI_SLV_CMD7_INT_RAW | SPI_SLV_EN_QPI_INT_RAW | SPI_SLV_EX_QPI_INT_RAW | SPI_DMA_OUTFIFO_EMPTY_ERR_INT_RAW | SPI_DMA_INFIFO_FULL_ERR_INT_RAW | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**SPI_DMA_INFIFO_FULL_ERR_INT_RAW** The raw bit for SPI_DMA_INFIFO_FULL_ERR_INT interrupt. (R/W/WTC/SS)

**SPI_DMA_OUTFIFO_EMPTY_ERR_INT_RAW** The raw bit for SPI_DMA_OUTFIFO_EMPTY_ERR_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_EX_QPI_INT_RAW** The raw bit for SPI_SLV_EX_QPI_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_EN_QPI_INT_RAW** The raw bit for SPI_SLV_EN_QPI_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_CMD7_INT_RAW** The raw bit for SPI_SLV_CMD7_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_CMD8_INT_RAW** The raw bit for SPI_SLV_CMD8_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_CMD9_INT_RAW** The raw bit for SPI_SLV_CMD9_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_CMDA_INT_RAW** The raw bit for SPI_SLV_CMDA_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_RD_DMA_DONE_INT_RAW** The raw bit for SPI_SLV_RD_DMA_DONE_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_WR_DMA_DONE_INT_RAW** The raw bit for SPI_SLV_WR_DMA_DONE_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_RD_BUF_DONE_INT_RAW** The raw bit for SPI_SLV_RD_BUF_DONE_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_WR_BUF_DONE_INT_RAW** The raw bit for SPI_SLV_WR_BUF_DONE_INT interrupt. (R/W/WTC/SS)

**SPI_TRANS_DONE_INT_RAW** The raw bit for SPI_TRANS_DONE_INT interrupt. (R/W/WTC/SS)

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

Register 27.19. SPI_DMA_INT_RAW_REG (0x003C)

Continued from the previous page...

**SPI_DMA_SEG_TRANS_DONE_INT_RAW** The raw bit for SPI_DMA_SEG_TRANS_DONE_INT interrupt. (R/W/WTC/SS)

**SPI_SEG_MAGIC_ERR_INT_RAW** The raw bit for SPI_SEG_MAGIC_ERR_INT interrupt. (R/W/WTC/SS)

**SPI_SLV_CMD_ERR_INT_RAW** The raw bit for SPI_SLV_CMD_ERR_INT interrupt. (R/W/WTC/SS)

**SPI_MST_RX_AFIFO_WFULL_ERR_INT_RAW** The raw bit for SPI_MST_RX_AFIFO_WFULL_ERR_INT interrupt. (R/W/WTC/SS)

**SPI_MST_TX_AFIFO_REMPTY_ERR_INT_RAW** The raw bit for SPI_MST_TX_AFIFO_REMPTY_ERR_INT interrupt. (R/W/WTC/SS)

**SPI_APP2_INT_RAW** The raw bit for SPI_APP2_INT interrupt. The value is only controlled by application. (R/W/WTC)

**SPI_APP1_INT_RAW** The raw bit for SPI_APP1_INT interrupt. The value is only controlled by application. (R/W/WTC)

**Register 27.20. SPI_DMA_INT_ST_REG (0x0040)**

| 31–21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | SPI_APP1_INT_ST | SPI_APP2_INT_ST | SPI_MST_TX_AFIFO_REMPTY_ERR_INT_ST | SPI_MST_RX_AFIFO_WFULL_ERR_INT_ST | SPI_SLV_CMD_ERR_INT_ST | (reserved) | SPI_SEG_MAGIC_ERR_INT_ST | SPI_DMA_SEG_TRANS_DONE_INT_ST | SPI_TRANS_DONE_INT_ST | SPI_SLV_WR_BUF_DONE_INT_ST | SPI_SLV_RD_BUF_DONE_INT_ST | SPI_SLV_WR_DMA_DONE_INT_ST | SPI_SLV_RD_DMA_DONE_INT_ST | SPI_SLV_CMDA_INT_ST | SPI_SLV_CMD9_INT_ST | SPI_SLV_CMD8_INT_ST | SPI_SLV_CMD7_INT_ST | SPI_SLV_EN_QPI_INT_ST | SPI_SLV_EX_QPI_INT_ST | SPI_DMA_OUTFIFO_EMPTY_ERR_INT_ST | SPI_DMA_INFIFO_FULL_ERR_INT_ST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**SPI_DMA_INFIFO_FULL_ERR_INT_ST** The status bit for SPI_DMA_INFIFO_FULL_ERR_INT interrupt. (RO)

**SPI_DMA_OUTFIFO_EMPTY_ERR_INT_ST** The status bit for SPI_DMA_OUTFIFO_EMPTY_ERR_INT interrupt. (RO)

**SPI_SLV_EX_QPI_INT_ST** The status bit for SPI_SLV_EX_QPI_INT interrupt. (RO)

**SPI_SLV_EN_QPI_INT_ST** The status bit for SPI_SLV_EN_QPI_INT interrupt. (RO)

**SPI_SLV_CMD7_INT_ST** The status bit for SPI_SLV_CMD7_INT interrupt. (RO)

**SPI_SLV_CMD8_INT_ST** The status bit for SPI_SLV_CMD8_INT interrupt. (RO)

**SPI_SLV_CMD9_INT_ST** The status bit for SPI_SLV_CMD9_INT interrupt. (RO)

**SPI_SLV_CMDA_INT_ST** The status bit for SPI_SLV_CMDA_INT interrupt. (RO)

**SPI_SLV_RD_DMA_DONE_INT_ST** The status bit for SPI_SLV_RD_DMA_DONE_INT interrupt. (RO)

**SPI_SLV_WR_DMA_DONE_INT_ST** The status bit for SPI_SLV_WR_DMA_DONE_INT interrupt. (RO)

**SPI_SLV_RD_BUF_DONE_INT_ST** The status bit for SPI_SLV_RD_BUF_DONE_INT interrupt. (RO)

**SPI_SLV_WR_BUF_DONE_INT_ST** The status bit for SPI_SLV_WR_BUF_DONE_INT interrupt. (RO)

**SPI_TRANS_DONE_INT_ST** The status bit for SPI_TRANS_DONE_INT interrupt. (RO)

**SPI_DMA_SEG_TRANS_DONE_INT_ST** The status bit for SPI_DMA_SEG_TRANS_DONE_INT interrupt. (RO)

**SPI_SEG_MAGIC_ERR_INT_ST** The status bit for SPI_SEG_MAGIC_ERR_INT interrupt. (RO)

**SPI_SLV_CMD_ERR_INT_ST** The status bit for SPI_SLV_CMD_ERR_INT interrupt. (RO)

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

Register 27.20. SPI_DMA_INT_ST_REG (0x0040)

Continued from the previous page...

**SPI_MST_RX_AFIFO_WFULL_ERR_INT_ST** The status bit for SPI_MST_RX_AFIFO_WFULL_ERR_INT interrupt. (RO)

**SPI_MST_TX_AFIFO_REMPTY_ERR_INT_ST** The status bit for SPI_MST_TX_AFIFO_REMPTY_ERR_INT interrupt. (RO)

**SPI_APP2_INT_ST** The status bit for SPI_APP2_INT interrupt. (RO)

**SPI_APP1_INT_ST** The status bit for SPI_APP1_INT interrupt. (RO)

Register 27.21. SPI_W0_REG (0x0098)

| SPI_BUF0 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF0** 32-bit data buffer 0. (R/W/SS)

Register 27.22. SPI_W1_REG (0x009C)

| SPI_BUF1 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF1** 32-bit data buffer 1. (R/W/SS)

Register 27.23. SPI_W2_REG (0x00A0)

| SPI_BUF2 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF2** 32-bit data buffer 2. (R/W/SS)

### Register 27.24. SPI_W3_REG (0x00A4)

| SPI_BUF3 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF3** 32-bit data buffer 3. (R/W/SS)

### Register 27.25. SPI_W4_REG (0x00A8)

| SPI_BUF4 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF4** 32-bit data buffer 4. (R/W/SS)

### Register 27.26. SPI_W5_REG (0x00AC)

| SPI_BUF5 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF5** 32-bit data buffer 5. (R/W/SS)

### Register 27.27. SPI_W6_REG (0x00B0)

| SPI_BUF6 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF6** 32-bit data buffer 6. (R/W/SS)

### Register 27.28. SPI_W7_REG (0x00B4)

| SPI_BUF7 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF7** 32-bit data buffer 7. (R/W/SS)

### Register 27.29. SPI_W8_REG (0x00B8)

| SPI_BUF8 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF8** 32-bit data buffer 8. (R/W/SS)

### Register 27.30. SPI_W9_REG (0x00BC)

| SPI_BUF9 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF9** 32-bit data buffer 9. (R/W/SS)

### Register 27.31. SPI_W10_REG (0x00C0)

| SPI_BUF10 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF10** 32-bit data buffer 10. (R/W/SS)

### Register 27.32. SPI_W11_REG (0x00C4)

| SPI_BUF11 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF11** 32-bit data buffer 11. (R/W/SS)

### Register 27.33. SPI_W12_REG (0x00C8)

| SPI_BUF12 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF12** 32-bit data buffer 12. (R/W/SS)

### Register 27.34. SPI_W13_REG (0x00CC)

| SPI_BUF13 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF13** 32-bit data buffer 13. (R/W/SS)

### Register 27.35. SPI_W14_REG (0x00D0)

| SPI_BUF14 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF14** 32-bit data buffer 14. (R/W/SS)

### Register 27.36. SPI_W15_REG (0x00D4)

| SPI_BUF15 | |
|---|---|
| 31 … 0 | |
| 0 | Reset |

**SPI_BUF15** 32-bit data buffer 15. (R/W/SS)

### Register 27.37. SPI_DATE_REG (0x00F0)

| (reserved) | SPI_DATE | |
|---|---|---|
| 31 … 28 | 27 … 0 | |
| 0 0 0 0 | 0x2007220 | Reset |

**SPI_DATE** Version control register. (R/W)

# Chapter 28

# I2C Controller (I2C)

The I2C (Inter-Integrated Circuit) bus allows ESP32-C3 to communicate with multiple external devices. These external devices can share one bus.

## 28.1 Overview

The I2C bus has two lines, namely a serial data line (SDA) and a serial clock line (SCL). Both SDA and SCL lines are open-drain. The I2C bus can be connected to a single or multiple master devices and a single or multiple slave devices. However, only one master device can access a slave at a time via the bus.

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 transmit a 7-bit address followed by a read/write ($R/\overline{W}$) bit. If the address of an I2C 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 RSTART condition, a slave address and a $R/\overline{W}$ bit before changing its operation. The RSTART condition is used to change the transfer direction and the mode of the devices (master mode or slave mode).

## 28.2 Features

The I2C controller has the following features:

- Master mode and slave mode
- Communication between multiple slaves
- Standard mode (100 Kbit/s)
- Fast mode (400 Kbit/s)
- 7-bit addressing and 10-bit addressing
- Continuous data transfer achieved by pulling SCL low
- Programmable digital noise filtering
- Double addressing mode, which uses slave address and slave memory or register address

## 28.3 I2C Architecture

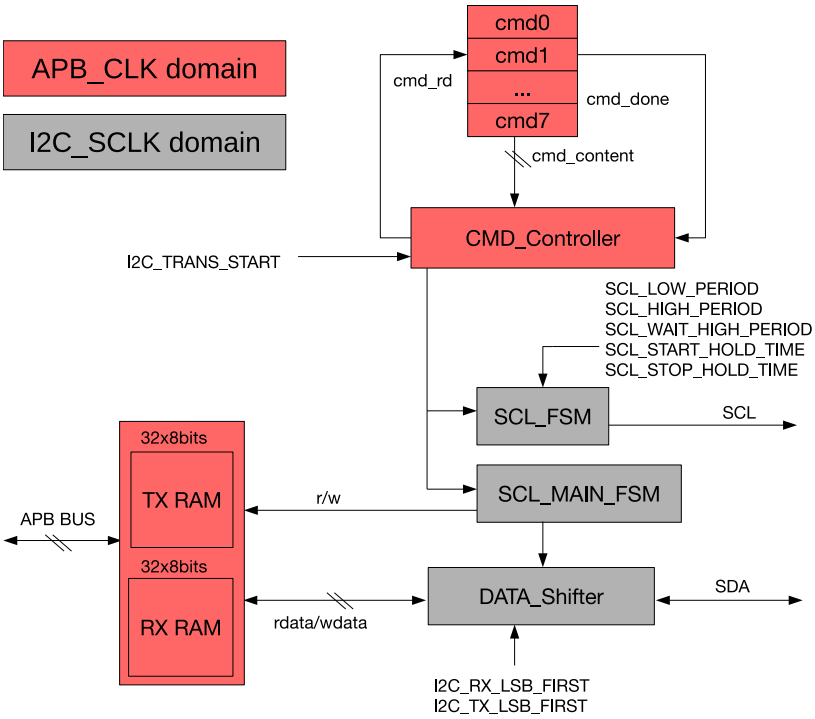



Figure 28.3-1. I2C Master Architecture

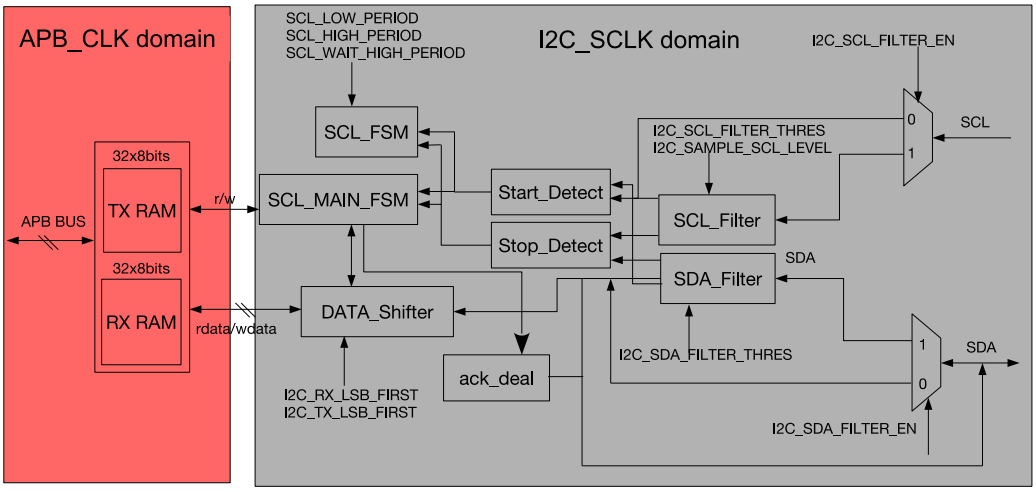



Figure 28.3-2. I2C Slave Architecture

The I2C controller runs either in master mode or slave mode, which is determined by I2C_MS_MODE. Figure 28.3-1 shows the architecture of a master, while Figure 28.3-2 shows that of a slave. The I2C controller has the following main parts:

- transmit and receive memory (TX/RX RAM)
- command controller (CMD_Controller)
- SCL clock controller (SCL_FSM)

- SDA data controller (SCL_MAIN_FSM)
- serial/parallel data converter (DATA_Shifter)
- filter for SCL (SCL_Filter)
- filter for SDA (SDA_Filter)

Besides, the I2C controller also has a clock module which generates I2C clocks, and a synchronization module which synchronizes the APB bus and the I2C controller.

The clock module is used to select clock sources, turn on and off clocks, and divide clocks. SCL_Filter and SDA_Filter remove noises on SCL input signals and SDA input signals respectively. The synchronization module synchronizes signal transfer between different clock domains.

Figure 28.3-3 and Figure 28.3-4 are the timing diagram and corresponding parameters of the I2C protocol. SCL_FSM generates the timing sequence conforming to the I2C protocol.

SCL_MAIN_FSM controls the execution of I2C commands and the sequence of the SDA line. CMD_Controller is used for an I2C master to generate (R)START, STOP, WRITE, READ and END commands. TX RAM and RX RAM store data to be transmitted and data received respectively. DATA_Shifter shifts data between serial and parallel form.

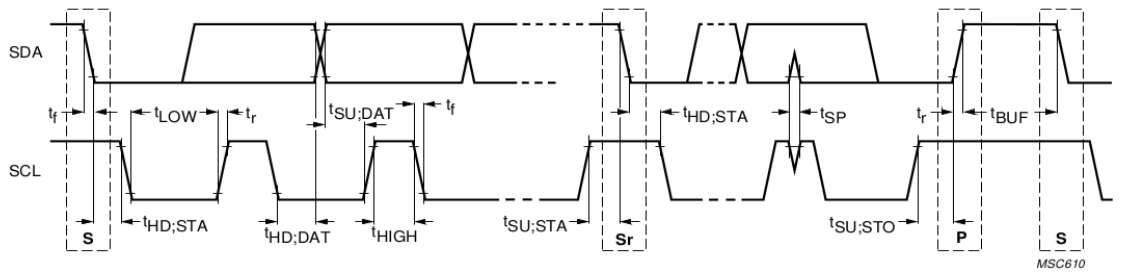



Fig.31 Definition of timing for F/S-mode devices on the I²C-bus.

**Figure 28.3-3. I2C Protocol Timing (Cited from Fig.31 in The I2C-bus specification Version 2.1)**

| PARAMETER | SYMBOL | STANDARD-MODE | | FAST-MODE | | UNIT |
|---|---|---|---|---|---|---|
| | | MIN. | MAX. | MIN. | MAX. | |
| SCL clock frequency | $f_{SCL}$ | 0 | 100 | 0 | 400 | kHz |
| Hold time (repeated) START condition. After this period, the first clock pulse is generated | $t_{HD;STA}$ | 4.0 | – | 0.6 | – | µs |
| LOW period of the SCL clock | $t_{LOW}$ | 4.7 | – | 1.3 | – | µs |
| HIGH period of the SCL clock | $t_{HIGH}$ | 4.0 | – | 0.6 | – | µs |
| Set-up time for a repeated START condition | $t_{SU;STA}$ | 4.7 | – | 0.6 | – | µs |
| Data hold time:<br>for CBUS compatible masters (see NOTE, Section 10.1.3)<br>for I²C-bus devices | $t_{HD;DAT}$ | <br>5.0<br>0(2) | <br>–<br>3.45(3) | <br>–<br>0(2) | <br>–<br>0.9(3) | <br>µs<br>µs |
| Data set-up time | $t_{SU;DAT}$ | 250 | – | 100(4) | – | ns |
| Rise time of both SDA and SCL signals | $t_r$ | – | 1000 | 20 + 0.1$C_b$(5) | 300 | ns |
| Fall time of both SDA and SCL signals | $t_f$ | – | 300 | 20 + 0.1$C_b$(5) | 300 | ns |
| Set-up time for STOP condition | $t_{SU;STO}$ | 4.0 | – | 0.6 | – | µs |
| Bus free time between a STOP and START condition | $t_{BUF}$ | 4.7 | – | 1.3 | – | µs |

Figure 28.3-4. I2C Timing Parameters (Cited from Table 5 in The I2C-bus specification Version 2.1)

## 28.4 Functional Description

Note that operations may differ between the I2C controller in ESP32-C3 and other masters or slaves on the bus. Please refer to datasheets of individual I2C devices for specific information.

### 28.4.1 Clock Configuration

Registers, TX RAM, and RX RAM are configured and accessed in the APB_CLK clock domain, whose frequency is 1 ∼ 80 MHz. The main logic of the I2C controller, including SCL_FSM, SCL_MAIN_FSM, SCL_FILTER, SDA_FILTER, and DATA_SHIFTER, are in the I2C_SCLK clock domain.

You can choose the clock source for I2C_SCLK from XTAL_CLK or RC_FAST_CLK via I2C_SCLK_SEL. When I2C_SCLK_SEL is cleared, the clock source is XTAL_CLK. When I2C_SCLK_SEL is set, the clock source is RC_FAST_CLK. The clock source is enabled by configuring I2C_SCLK_ACTIVE as high level, and then passes through a fractional divider to generate I2C_SCLK according to the following equation:

$$Divisor = I2C_SCLK_DIV_NUM + 1 + \frac{I2C_SCLK_DIV_A}{I2C_SCLK_DIV_B}$$

The frequency of XTAL_CLK is 40 MHz, while the frequency of RC_FAST_CLK is 17.5 MHz. Limited by timing parameters, the derived clock I2C_SCLK should operate at a frequency 20 timers larger than SCL's frequency.

### 28.4.2 SCL and SDA Noise Filtering

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.

Take SCL_Filter as an example. When enabled, 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 I2C_SCLK clock cycles. Given that only valid input signals can pass through the filter, SCL_Filter can remove glitches whose pulse width is shorter than I2C_SCL_FILTER_THRES I2C_SCLK clock cycles, while SDA_Filter can remove glitches whose pulse width is shorter than I2C_SDA_FILTER_THRES I2C_SCLK clock cycles.

### 28.4.3 SCL Clock Stretching

The I2C controller in slave mode (i.e. slave) can hold the SCL line low in exchange for more time to process data. This function called clock stretching is enabled by setting the I2C_SLAVE_SCL_STRETCH_EN bit. The time period to release the SCL line from stretching is configured by setting the I2C_STRETCH_PROTECT_NUM field, in order to avoid timing sequence errors. The slave will hold the SCL line low when one of the following four events occurs:

1. Address match: The address of the slave matches the address sent by the master via the SDA line, and the $R/\overline{W}$ bit is 1.
2. RAM being full: RX RAM of the slave is full. Note that when the slave receives less than 32 bytes, it is not necessary to enable clock stretching; when the slave receives 32 bytes or more, you may interrupt data transmission to wrapped around RAM via the FIFO threshold, or enable clock stretching for more time to process data. When clock stretching is enabled, I2C_RX_FULL_ACK_LEVEL must be cleared, otherwise there will be unpredictable consequences.
3. RAM being empty: The slave is sending data, but its TX RAM is empty.
4. Sending an ACK: If I2C_SLAVE_BYTE_ACK_CTL_EN is set, the slave pulls SCL low when sending an ACK bit. At this stage, software validates data and configures I2C_SLAVE_BYTE_ACK_LVL to control the level of the ACK bit. Note that when RX RAM of the slave is full, the level of the ACK bit to be sent is determined by I2C_RX_FULL_ACK_LEVEL, instead of I2C_SLAVE_BYTE_ACK_LVL. In this case, I2C_RX_FULL_ACK_LEVEL should also be cleared to ensure proper functioning of clock stretching.

After SCL has been stretched low, the cause of stretching can be read from the I2C_STRETCH_CAUSE bit. Clock stretching is disabled by setting the I2C_SLAVE_SCL_STRETCH_CLR bit.

### 28.4.4 Generating SCL Pulses in Idle State

Usually when the I2C bus is idle, the SCL line is held high. The I2C controller in ESP32-C3 can be programmed to generate SCL pulses in idle state. This function only works when the I2C controller is configured as master. If the I2C_SCL_RST_SLV_EN bit is set, hardware will send I2C_SCL_RST_SLV_NUM SCL pulses. When software reads 0 in I2C_SCL_RST_SLV_EN, set I2C_CONF_UPGATE to stop this function.

### 28.4.5 Synchronization

I2C registers are configured in APB_CLK domain, whereas the I2C controller is configured in asynchronous I2C_SCLK domain. Therefore, before being used by the I2C controller, register values should be synchronized by first writing configuration registers and then writing 1 to I2C_CONF_UPGATE. Registers that need synchronization are listed in Table 28.4-1.

Table 28.4-1. I2C Synchronous Registers

| Register | Parameter | Address |
|---|---|---|
| I2C_CTR_REG | I2C_SLV_TX_AUTO_START_EN | 0x0004 |
| | I2C_ADDR_10BIT_RW_CHECK_EN | |
| | I2C_ADDR_BROADCASTING_EN | |
| | I2C_SDA_FORCE_OUT | |
| | I2C_SCL_FORCE_OUT | |
| | I2C_SAMPLE_SCL_LEVEL | |
| | I2C_RX_FULL_ACK_LEVEL | |
| | I2C_MS_MODE | |
| | I2C_TX_LSB_FIRST | |
| | I2C_RX_LSB_FIRST | |
| | I2C_ARBITRATION_EN | |
| I2C_TO_REG | I2C_TIME_OUT_EN | 0x000C |
| | I2C_TIME_OUT_VALUE | |
| I2C_SLAVE_ADDR_REG | I2C_ADDR_10BIT_EN | 0x0010 |
| | I2C_SLAVE_ADDR | |
| I2C_FIFO_CONF_REG | I2C_FIFO_ADDR_CFG_EN | 0x0018 |
| I2C_SCL_SP_CONF_REG | I2C_SDA_PD_EN | 0x0080 |
| | I2C_SCL_PD_EN | |
| | I2C_SCL_RST_SLV_NUM | |
| | I2C_SCL_RST_SLV_EN | |
| I2C_SCL_STRETCH_CONF_REG | I2C_SLAVE_BYTE_ACK_CTL_EN | 0x0084 |
| | I2C_SLAVE_BYTE_ACK_LVL | |
| | I2C_SLAVE_SCL_STRETCH_EN | |
| | I2C_STRETCH_PROTECT_NUM | |
| I2C_SCL_LOW_PERIOD_REG | I2C_SCL_LOW_PERIOD | 0x0000 |
| I2C_SCL_HIGH_PERIOD_REG | I2C_WAIT_HIGH_PERIOD | 0x0038 |
| | I2C_HIGH_PERIOD | |
| I2C_SDA_HOLD_REG | I2C_SDA_HOLD_TIME | 0x0030 |
| I2C_SDA_SAMPLE_REG | I2C_SDA_SAMPLE_TIME | 0x0034 |
| I2C_SCL_START_HOLD_REG | I2C_SCL_START_HOLD_TIME | 0x0040 |
| I2C_SCL_RSTART_SETUP_REG | I2C_SCL_RSTART_SETUP_TIME | 0x0044 |
| I2C_SCL_STOP_HOLD_REG | I2C_SCL_STOP_HOLD_TIME | 0x0048 |
| I2C_SCL_STOP_SETUP_REG | I2C_SCL_STOP_SETUP_TIME | 0x004C |
| I2C_SCL_ST_TIME_OUT_REG | I2C_SCL_ST_TO_I2C | 0x0078 |
| I2C_SCL_MAIN_ST_TIME_OUT_REG | I2C_SCL_MAIN_ST_TO_I2C | 0x007C |
| I2C_FILTER_CFG_REG | I2C_SCL_FILTER_EN | 0x0050 |
| | I2C_SCL_FILTER_THRES | |
| | I2C_SDA_FILTER_EN | |
| | I2C_SDA_FILTER_THRES | |

### 28.4.6 Open-Drain Output

SCL and SDA output drivers must be configured as open drain. There are two ways to achieve this:

1. Set I2C_SCL_FORCE_OUT and I2C_SDA_FORCE_OUT, and configure GPIO_PIN*n*_PAD_DRIVER 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 longer, determined together by the pull-up resistor and the line capacitance. The output duty cycle of I2C is limited by the SDA and SCL line's pull-up speed, mainly SCL's speed.

In addition, when I2C_SCL_FORCE_OUT and I2C_SCL_PD_EN are set to 1, SCL can be forced low; when I2C_SDA_FORCE_OUT and I2C_SDA_PD_EN are set to 1, SDA can be forced low.

### 28.4.7 Timing Parameter Configuration

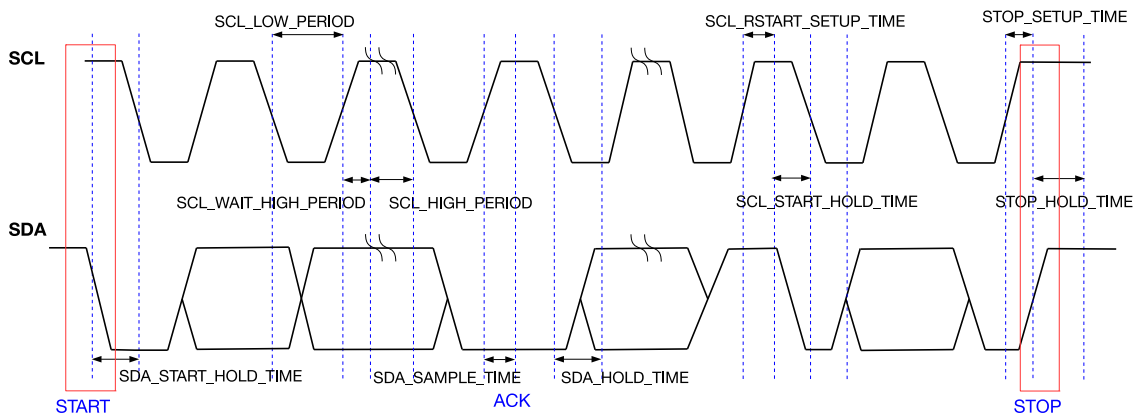



Figure 28.4-1. I2C Timing Diagram

Figure 28.4-1 shows the timing diagram of an I2C master. This figure 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. Timing parameters are calculated as follows in I2C_SCLK clock cycles:

1. $t_{LOW} = (I2C_SCL_LOW_PERIOD + 1) \cdot T_{I2C_SCLK}$
2. $t_{HIGH} = (I2C_SCL_HIGH_PERIOD + 1) \cdot T_{I2C_SCLK}$
3. $t_{SU:STA} = (I2C_SCL_RSTART_SETUP_TIME + 1) \cdot T_{I2C_SCLK}$
4. $t_{HD:STA} = (I2C_SCL_START_HOLD_TIME + 1) \cdot T_{I2C_SCLK}$
5. $t_r = (I2C_SCL_WAIT_HIGH_PERIOD + 1) \cdot T_{I2C_SCLK}$
6. $t_{SU:STO} = (I2C_SCL_STOP_SETUP_TIME + 1) \cdot T_{I2C_SCLK}$
7. $t_{BUF} = (I2C_SCL_STOP_HOLD_TIME + 1) \cdot T_{I2C_SCLK}$
8. $t_{HD:DAT} = (I2C_SDA_HOLD_TIME + 1) \cdot T_{I2C_SCLK}$
9. $t_{SU:DAT} = (I2C_SCL_LOW_PERIOD - I2C_SDA_HOLD_TIME) \cdot T_{I2C_SCLK}$

Timing registers below are divided into two groups, depending on the mode in which these registers are active:

- Master mode only:

1. I2C_SCL_START_HOLD_TIME: Specifies the interval between pulling SDA low and pulling SCL low when the master generates a START condition. This interval is (I2C_SCL_START_HOLD_TIME +1) in I2C_SCLK cycles. This register is active only when the I2C controller works in master mode.
2. I2C_SCL_LOW_PERIOD: Specifies the low period of SCL. This period lasts (I2C_SCL_LOW_PERIOD + 1) in I2C_SCLK cycles. However, it could be extended when SCL is pulled low by peripheral devices or by an END command executed by the I2C controller, or when the clock is stretched. This register is active only when the I2C controller works in master mode.
3. I2C_SCL_WAIT_HIGH_PERIOD: Specifies time for SCL to go high in I2C_SCLK cycles. Please make sure that SCL could be pulled high within this time period. Otherwise, the high period of SCL may be incorrect. This register is active only when the I2C controller works in master mode.
4. I2C_SCL_HIGH_PERIOD: Specifies the high period of SCL in I2C_SCLK cycles. This register is active only when the I2C controller works in master mode. When SCL goes high within (I2C_SCL_WAIT_HIGH_PERIOD + 1) in I2C_SCLK cycles, its frequency is:

$$f_{scl} = \frac{f_{\text{I2C_SCLK}}}{\text{I2C_SCL_LOW_PERIOD} + \text{I2C_SCL_HIGH_PERIOD} + \text{I2C_SCL_WAIT_HIGH_PERIOD} + 3}$$

- Master mode and slave mode:
  1. 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, so as to correctly sample the level of SCL. This register is active both in master mode and slave mode.
  2. 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.

Timing parameters limits corresponding register configuration.

1. $\frac{f_{I2C_SCLK}}{f_{SCL}} > 20$
2. $3 \times f_{I2C_SCLK} \leq (I2C_SDA_HOLD_TIME - 4) \times f_{APB_CLK}$
3. I2C_SDA_HOLD_TIME + I2C_SCL_START_HOLD_TIME > SDA_FILTER_THRES + 3
4. I2C_SCL_WAIT_HIGH_PERIOD < I2C_SDA_SAMPLE_TIME < I2C_SCL_HIGH_PERIOD
5. I2C_SDA_SAMPLE_TIME < I2C_SCL_WAIT_HIGH_PERIOD + I2C_SCL_START_HOLD_TIME + I2C_SCL_RSTART_SETUP_TIME
6. I2C_STRETCH_PROTECT_NUM + I2C_SDA_HOLD_TIME > I2C_SCL_LOW_PERIOD

### 28.4.8 Timeout Control

The I2C controller has three types of timeout control, namely timeout control for SCL_FSM, for SCL_MAIN_FSM, and for the SCL line. The first two are always enabled, while the third is configurable.

When SCL_FSM remains unchanged for more than $2^{I2C_SCL_ST_TO_I2C}$ clock cycles, an I2C_SCL_ST_TO_INT interrupt is triggered, and then SCL_FSM goes to idle state. The value of I2C_SCL_ST_TO_I2C should be less than or equal to 22, which means SCL_FSM could remain unchanged for $2^{22}$ I2C_SCLK clock cycles at most before the interrupt is generated.

When SCL_MAIN_FSM remains unchanged for more than $2^{I2C_SCL_MAIN_ST_TO_I2C}$ clock cycles, an

I2C_SCL_MAIN_ST_TO_INT interrupt is triggered, and then SCL_MAIN_FSM goes to idle state. The value of I2C_SCL_MAIN_ST_TO_I2C should be less than or equal to 22, which means SCL_MAIN_FSM could remain unchanged for $2^{22}$ I2C_SCLK clock cycles at most before the interrupt is generated.

Timeout control for SCL is enabled by setting I2C_TIME_OUT_EN. When the level of SCL remains unchanged for more than $2^{I2C_TIME_OUT_VALUE}$ clock cycles, an I2C_TIME_OUT_INT interrupt is triggered, and then the I2C bus goes to idle state.

### 28.4.9 Command Configuration

When the I2C controller works in master mode, CMD_Controller reads commands from 8 sequential command registers and controls SCL_FSM and SCL_MAIN_FSM accordingly.

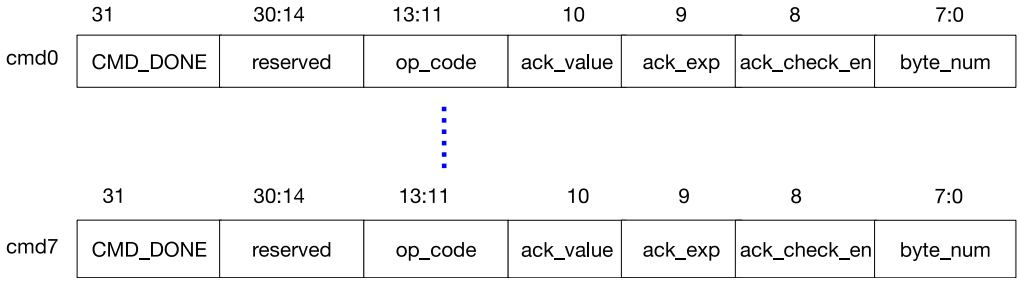



**Figure 28.4-2. Structure of I2C Command Registers**

Command registers, whose structure is illustrated in Figure 28.4-2, are active only when the I2C controller works in master mode. Fields of command registers are:

1. CMD_DONE: Indicates that a command has been executed. After each command has been executed, the CMD_DONE bit in the corresponding command register is set to 1 by hardware. By reading this bit, software can tell if the command has been executed. When writing new commands, this bit must be cleared by software.
2. op_code: Indicates the command. The I2C controller supports five commands:
   - RSTART: op_code = 6. The I2C controller sends a START bit or a RSTART bit defined by the I2C protocol.
   - WRITE: op_code = 1. The I2C controller sends a slave address, a register address (only in double addressing mode) and data to the slave.
   - READ: op_code = 3. The I2C controller reads data from the slave.
   - STOP: op_code = 2. The I2C controller sends a STOP bit defined by the I2C protocol. This code also indicates that the command sequence has been executed, and the CMD_Controller stops reading commands. After restarted by software, the CMD_Controller resumes reading commands from command register 0.
   - END: op_code = 4. The I2C controller pulls the SCL line down and suspends I2C communication. This code also indicates that the command sequence has completed, and the CMD_Controller stops executing commands. Once software refreshes data in command registers and the RAM, the CMD_Controller can be restarted to execute commands from command register 0 again.

3. ack_value: Used to configure the level of the ACK bit sent by the I2C controller during a read operation. This bit is ignored in RSTART, STOP, END and WRITE conditions.
4. ack_exp: Used to configure the level of the ACK bit expected by the I2C controller during a write operation. This bit is ignored during RSTART, STOP, END and READ conditions.
5. ack_check_en: Used to enable the I2C controller during a write operation to check whether the ACK 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, there must be a STOP or an END command in one command sequence.

A complete data transfer on the I2C bus should be initiated by a START and terminated by a STOP. The transfer process may be completed using multiple sequences, separated by END commands. Each sequence may differ in the direction of data transfer, clock frequency, slave addresses, data length, etc. This allows efficient use of available peripheral RAM and also achieves more flexible I2C communication.

### 28.4.10 TX/RX RAM Data Storage

Both TX RAM and RX RAM are 32 × 8 bits, and can be accessed in FIFO or non-FIFO mode. If I2C_NONFIFO_EN bit is cleared, both RAMs are accessed in FIFO mode; if I2C_NONFIFO_EN bit is set, both RAMs are accessed in non-FIFO mode.

TX RAM stores data that the I2C controller needs to send. During communication, when the I2C controller needs to send data (except acknowledgement bits), it reads data from TX RAM and sends them sequentially via SDA. When the I2C controller works in master mode, all data must be stored in TX RAM in the order they will be sent to slaves. The data stored in TX RAM include slave addresses, read/write bits, register addresses (only in double addressing mode) and data to be sent. When the I2C controller works in slave mode, TX RAM only stores data to be sent.

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 the fixed address I2C_DATA_REG, with addresses for writing in TX RAM incremented automatically by hardware. In non-FIFO mode, the CPU accesses TX RAM directly via address fields (I2C Base Address + 0x100) ~(I2C Base Address + 0x17C). Each byte in TX RAM occupies an entire word in the address space. Therefore, the address of the first byte is I2C Base Address + 0x100, the second byte is I2C Base Address + 0x104, the third byte is I2C Base Address + 0x108, and so on. The CPU can only read TX RAM via direct addresses. Addresses for reading TX RAM are the same with addresses for writing TX RAM.

RX RAM stores data the I2C controller receives during communication. When the I2C controller works in slave mode, neither slave addresses sent by the master nor register addresses (only in double addressing mode) will be stored into RX RAM. Values of RX RAM can be read by software after I2C communication completes.

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 the fixed address I2C_DATA_REG, with addresses for reading RX RAM incremented automatically by hardware. In non-FIFO mode, the CPU accesses TX RAM directly

via address fields (I2C Base Address + 0x180) ~(I2C Base Address + 0x1FC). Each byte in RX RAM occupies an entire word in the address space. Therefore, the address of the first byte is I2C Base Address + 0x180, the second byte is I2C Base Address + 0x184, the third byte is I2C Base Address + 0x188 and so on.

In FIFO mode, TX RAM of a master may wrap around to send data larger than 32 bytes. Set I2C_FIFO_PRT_EN. If the size of data to be sent is smaller than I2C_TXFIFO_WM_THRHD (master), an I2C_TXFIFO_WM_INT (master) interrupt is generated. After receiving the interrupt, software continues writing to I2C_DATA_REG (master). Please ensure that software writes to or refreshes TX RAM before the master sends data, otherwise it may result in unpredictable consequences.

In FIFO mode, RX RAM of a slave may also wrap around to receive data larger than 32 bytes. Set I2C_FIFO_PRT_EN and clear I2C_RX_FULL_ACK_LEVEL. If data already received (to be overwritten) is larger than I2C_RXFIFO_WM_THRHD (slave), an I2C_RXFIFO_WM_INT (slave) interrupt is generated. After receiving the interrupt, software continues reading from I2C_DATA_REG (slave).

### 28.4.11 Data Conversion

DATA_Shifter is used for serial/parallel conversion, converting byte data in TX RAM to an outgoing serial bitstream or an incoming serial bitstream to byte data in RX RAM. I2C_RX_LSB_FIRST and I2C_TX_LSB_FIRST can be used to select LSB- or MSB-first storage and transmission of data.

### 28.4.12 Addressing Mode

Besides 7-bit addressing, the ESP32-C3 I2C controller also supports 10-bit addressing and double addressing. 10-bit addressing can be mixed with 7-bit addressing.

Define the slave address as SLV_ADDR. In 7-bit addressing mode, the slave address is SLV_ADDR[6:0]; in 10-bit addressing mode, the slave address is SLV_ADDR[9:0].

In 7-bit addressing mode, the master only needs to send one byte of address, which comprises SLV_ADDR[6:0] and a $R/\overline{W}$ bit. In 7-bit addressing mode, there is a special case called general call addressing (broadcast). It is enabled by setting I2C_ADDR_BROADCASTING_EN in a slave. When the slave receives the general call address (0x00) from the master and the $R/\overline{W}$ bit followed is 0, it responds to the master regardless of its own address.

In 10-bit addressing mode, the master needs to send two bytes of address. The first byte is slave_addr_first_7bits followed by a $R/\overline{W}$ bit, and slave_addr_first_7bits should be configured as (0x78 | SLV_ADDR[9:8]). The second byte is slave_addr_second_byte, which should be configured as SLV_ADDR[7:0]. The slave can enable 10-bit addressing by configuring I2C_ADDR_10BIT_EN. I2C_SLAVE_ADDR is used to configure I2C slave address. Specifically, 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.

When working in slave mode, the I2C controller supports double addressing, where the first address is the address of an I2C slave, and the second one is the slave's memory address. When using double addressing, RAM must be accessed in non-FIFO mode. Double addressing is enabled by setting I2C_FIFO_ADDR_CFG_EN.

### 28.4.13 $R/\overline{W}$ Bit Check in 10-bit Addressing Mode

In 10-bit addressing mode, when I2C_ADDR_10BIT_RW_CHECK_EN is set to 1, the I2C controller performs a check on the first byte, which consists of slave_addr_first_7bits and a $R/\overline{W}$ bit. When the $R/\overline{W}$ bit does not indicate a WRITE operation, i.e. not in line with the I2C protocol, the data transfer ends. If the check feature is not enabled, when the $R/\overline{W}$ bit does not indicate a WRITE, the data transfer still continues, but transfer failure may occur.

### 28.4.14 To Start the I2C Controller

To start the I2C controller in master mode, after configuring the controller to master mode and command registers, write 1 to I2C_TRANS_START in order that the master starts to parse and execute command sequences. The master always executes a command sequence starting from command register 0 to a STOP or an END at the end. To execute another command sequence starting from command register 0, refresh commands by writing 1 again to I2C_TRANS_START.

To start the I2C controller in slave mode, there are two ways:

- Set I2C_SLV_TX_AUTO_START_EN, and the slave starts automatic transfer upon an address match;
- Clear I2C_SLV_TX_AUTO_START_EN, and always set I2C_TRANS_START before transfer.

## 28.5 Programming Example

This sections provides programming examples for typical communication scenarios. ESP32-C3 has one I2C controller. For the convenience of description, I2C masters and slaves in all subsequent figures are ESP32-C3 I2C controllers. I2C master is referred to as $\text{I2C}_{\text{master}}$, and I2C slave is referred to as $\text{I2C}_{\text{slave}}$.

### 28.5.1 $\text{I2C}_{\text{master}}$ Writes to $\text{I2C}_{\text{slave}}$ with a 7-bit Address in One Command Sequence

#### 28.5.1.1 Introduction

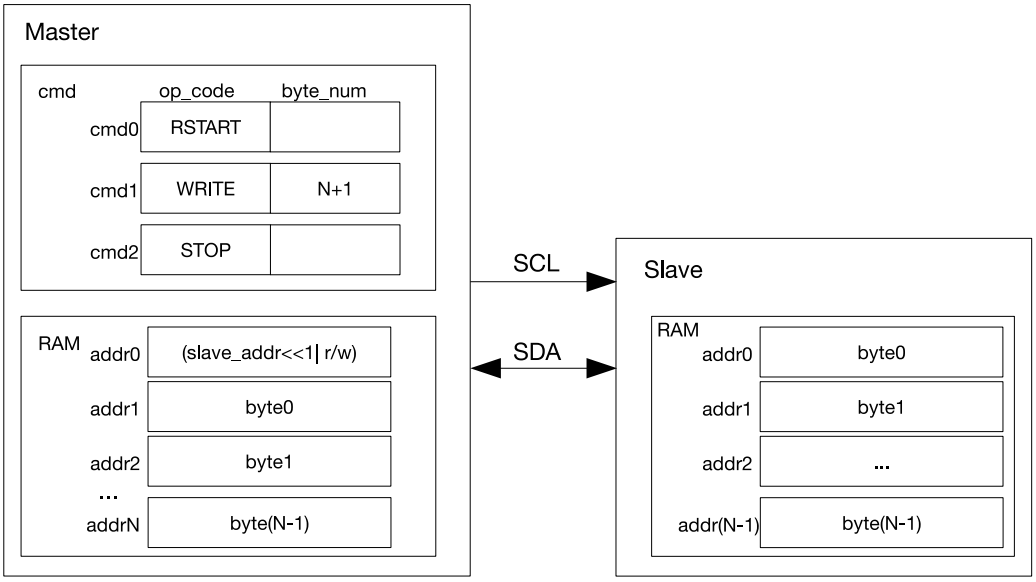



Figure 28.5-1. I2C$_{master}$ Writing to I2C$_{slave}$ with a 7-bit Address

Figure 28.5-1 shows how I2C$_{master}$ writes N bytes of data to I2C$_{slave}$'s RAM using 7-bit addressing. As shown in figure 28.5-1 , the first byte in the RAM of I2C$_{master}$ is a 7-bit I2C$_{slave}$ address followed by a $R/\overline{W}$ bit. When the $R/\overline{W}$ bit is 0, 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, I2C$_{master}$ enables the controller and initiates data transfer by setting the I2C_TRANS_START bit. The controller has four steps to take:

1. Wait for SCL to go high, to avoid SCL being 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 I2C$_{slave}$ in the same order. The first byte is the address of I2C$_{slave}$.
4. Send a STOP. Once the I2C$_{master}$ transfers a STOP bit, an I2C_TRANS_COMPLETE_INT interrupt is generated.

#### 28.5.1.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
3. Configure command registers of I2C$_{master}$.

| Command register | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | ack_value | ack_exp | 1 | N+1 |
| I2C_COMMAND2 (master) | STOP | — | — | — | — |

4. Write $I2C_{slave}$ address and data to be sent to TX RAM of $I2C_{master}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
5. Write address of $I2C_{slave}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register.
6. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
7. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start transfer.
8. $I2C_{slave}$ compares the slave address sent by $I2C_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $I2C_{master}$'s WRITE command is 1, $I2C_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, $I2C_{master}$ does not check ACK value and take $I2C_{slave}$ as a matching slave by default.
   - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $I2C_{master}$ continues data transfer.
   - Not match: If the received ACK value does not match ack_exp, $I2C_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
9. $I2C_{master}$ sends data, and checks ACK value or not according to ack_check_en (master).
10. If data to be sent (N) is larger than 32 bytes, TX RAM of $I2C_{master}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.
11. If data to be received (N) is larger than 32 bytes, RX RAM of $I2C_{slave}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.

    If data to be received (N) is larger than 32 bytes, the other way is to enable clock stretching by setting the I2C_SLAVE_SCL_STRETCH_EN (slave), and clearing I2C_RX_FULL_ACK_LEVEL. When RX RAM is full, an I2C_SLAVE_STRETCH_INT (slave) interrupt is generated. In this way, $I2C_{slave}$ can hold SCL low, in exchange for more time to read data. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
12. After data transfer completes, $I2C_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

### 28.5.2 $I2C_{master}$ Writes to $I2C_{slave}$ with a 10-bit Address in One Command Sequence

#### 28.5.2.1 Introduction

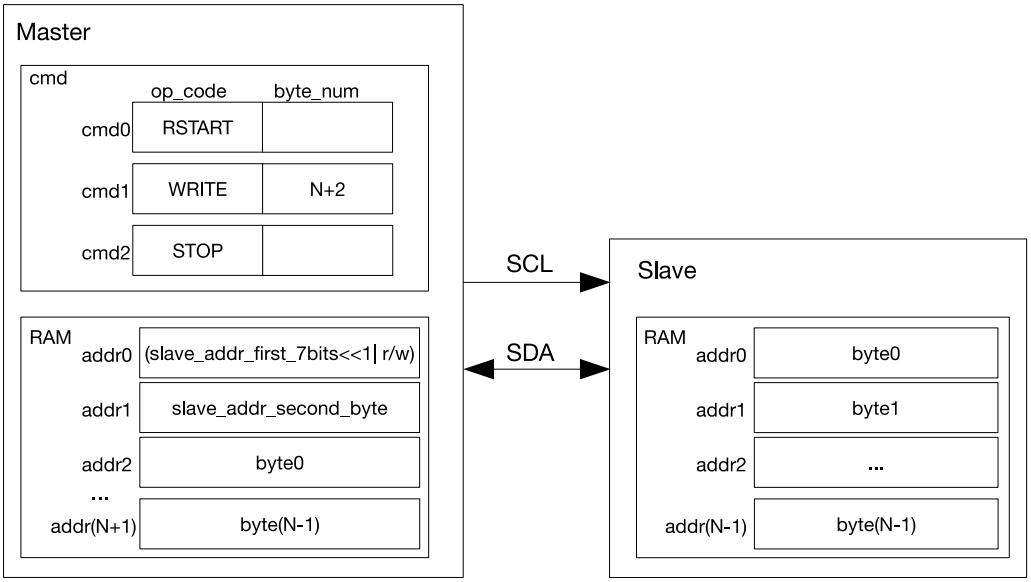



Figure 28.5-2. $I2C_{master}$ Writing to a Slave with a 10-bit Address

Figure 28.5-2 shows how $I2C_{master}$ writes N bytes of data using 10-bit addressing to an I2C slave. The configuration and transfer process is similar to what is described in 28.5.1, except that a 10-bit $I2C_{slave}$ address is formed from two bytes. Since a 10-bit $I2C_{slave}$ address has one more byte than a 7-bit $I2C_{slave}$ address, byte_num and length of data in TX RAM increase by 1 accordingly.

#### 28.5.2.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
3. Configure command registers of $I2C_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | ack_value | ack_exp | 1 | N+2 |
| I2C_COMMAND2 (master) | STOP | — | — | — | — |

4. Configure I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) as $I2C_{slave}$'s 10-bit address, and set I2C_ADDR_10BIT_EN (slave) to 1 to enable 10-bit addressing.
5. Write $I2C_{slave}$ address and data to be sent to TX RAM of $I2C_{master}$. The first byte of $I2C_{slave}$ address comprises ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a $R/\overline{W}$ bit. The second byte of $I2C_{slave}$ address is I2C_SLAVE_ADDR[7:0]. These two bytes are followed by data to be sent in FIFO or non-FIFO mode.
6. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
7. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start transfer.

8. $I2C_{slave}$ compares the slave address sent by $I2C_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $I2C_{master}$'s WRITE command is 1, $I2C_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, $I2C_{master}$ does not check ACK value and take $I2C_{slave}$ as matching slave by default.
   - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $I2C_{master}$ continues data transfer.
   - Not match: If the received ACK value does not match ack_exp, $I2C_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
9. $I2C_{master}$ sends data, and checks ACK value or not according to ack_check_en (master).
10. If data to be sent is larger than 32 bytes, TX RAM of $I2C_{master}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.
11. If data to be received is larger than 32 bytes, RX RAM of $I2C_{slave}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.

    If data to be received is larger than 32 bytes, the other way is to enable clock stretching by setting I2C_SLAVE_SCL_STRETCH_EN (slave), and clearing I2C_RX_FULL_ACK_LEVEL to 0. When RX RAM is full, an I2C_SLAVE_STRETCH_INT (slave) interrupt is generated. In this way, $I2C_{slave}$ can hold SCL low, in exchange for more time to read data. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
12. After data transfer completes, $I2C_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.5.3 $I2C_{master}$ Writes to $I2C_{slave}$ with Two 7-bit Addresses in One Command Sequence

### 28.5.3.1 Introduction

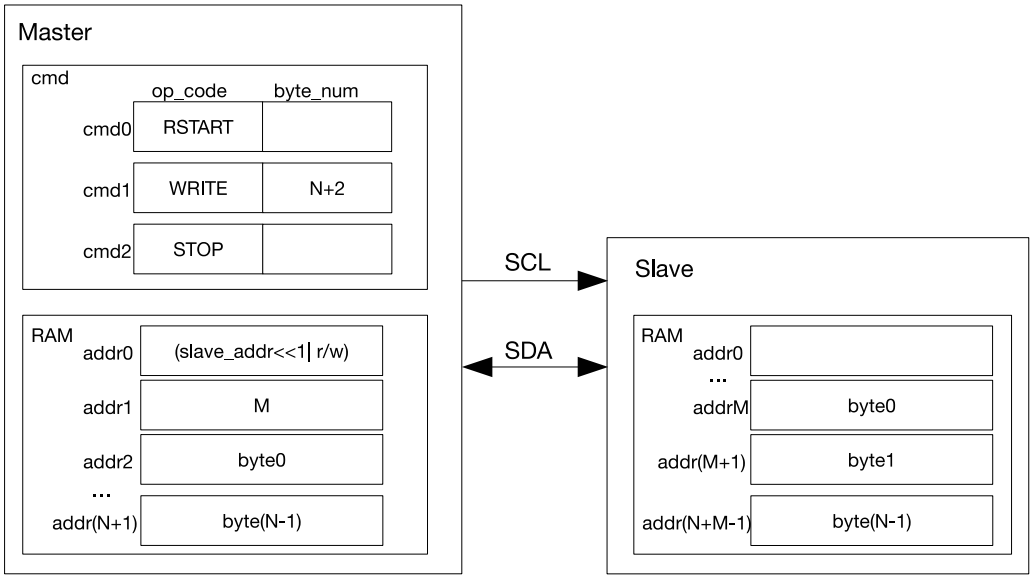



Figure 28.5-3. $I2C_{master}$ Writing to $I2C_{slave}$ with Two 7-bit Addresses

Figure 28.5-3 shows how $I2C_{master}$ writes N bytes of data to $I2C_{slave}$'s RAM using 7-bit double addressing. The configuration and transfer process is similar to what is described in Section 28.5.1, except that in 7-bit double addressing mode $I2C_{master}$ sends two 7-bit addresses. The first address is the address of an I2C slave, and the second one is $I2C_{slave}$'s memory address (i.e. addrM in Figure 28.5-3). When using double addressing, RAM must be accessed in non-FIFO mode. The I2C slave put received byte0 ~ byte(N-1) into its RAM in an order staring from addrM. The RAM is overwritten every 32 bytes.

### 28.5.3.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. Set I2C_FIFO_ADDR_CFG_EN (slave) to 1 to enable double addressing mode.
3. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
4. Configure command registers of $I2C_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | ack_value | ack_exp | 1 | N+2 |
| I2C_COMMAND2 (master) | STOP | — | — | — | — |

5. Write $I2C_{slave}$ address and data to be sent to TX RAM of $I2C_{master}$ in FIFO or non-FIFO mode.
6. Write address of $I2C_{slave}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register.

7. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
8. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start transfer.
9. $I2C_{slave}$ compares the slave address sent by $I2C_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $I2C_{master}$'s WRITE command is 1, $I2C_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, $I2C_{master}$ does not check ACK value and take $I2C_{slave}$ as matching slave by default.
   - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $I2C_{master}$ continues data transfer.
   - Not match: If the received ACK value does not match ack_exp, $I2C_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
10. $I2C_{slave}$ receives the RX RAM address sent by $I2C_{master}$ and adds the offset.
11. $I2C_{master}$ sends data, and checks ACK value or not according to ack_check_en (master).
12. If data to be sent is larger than 32 bytes, TX RAM of $I2C_{master}$ may wrap around in FIFO mode. For details, please refer to Section 28.4.10.
13. If data to be received is larger than 32 bytes, you may enable clock stretching by setting I2C_SLAVE_SCL_STRETCH_EN (slave), and clearing I2C_RX_FULL_ACK_LEVEL to 0. When RX RAM is full, an I2C_SLAVE_STRETCH_INT (slave) interrupt is generated. In this way, $I2C_{slave}$ can hold SCL low, in exchange for more time to read data. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
14. After data transfer completes, $I2C_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.5.4 $I2C_{master}$ Writes to $I2C_{slave}$ with a 7-bit Address in Multiple Command Sequences

### 28.5.4.1 Introduction

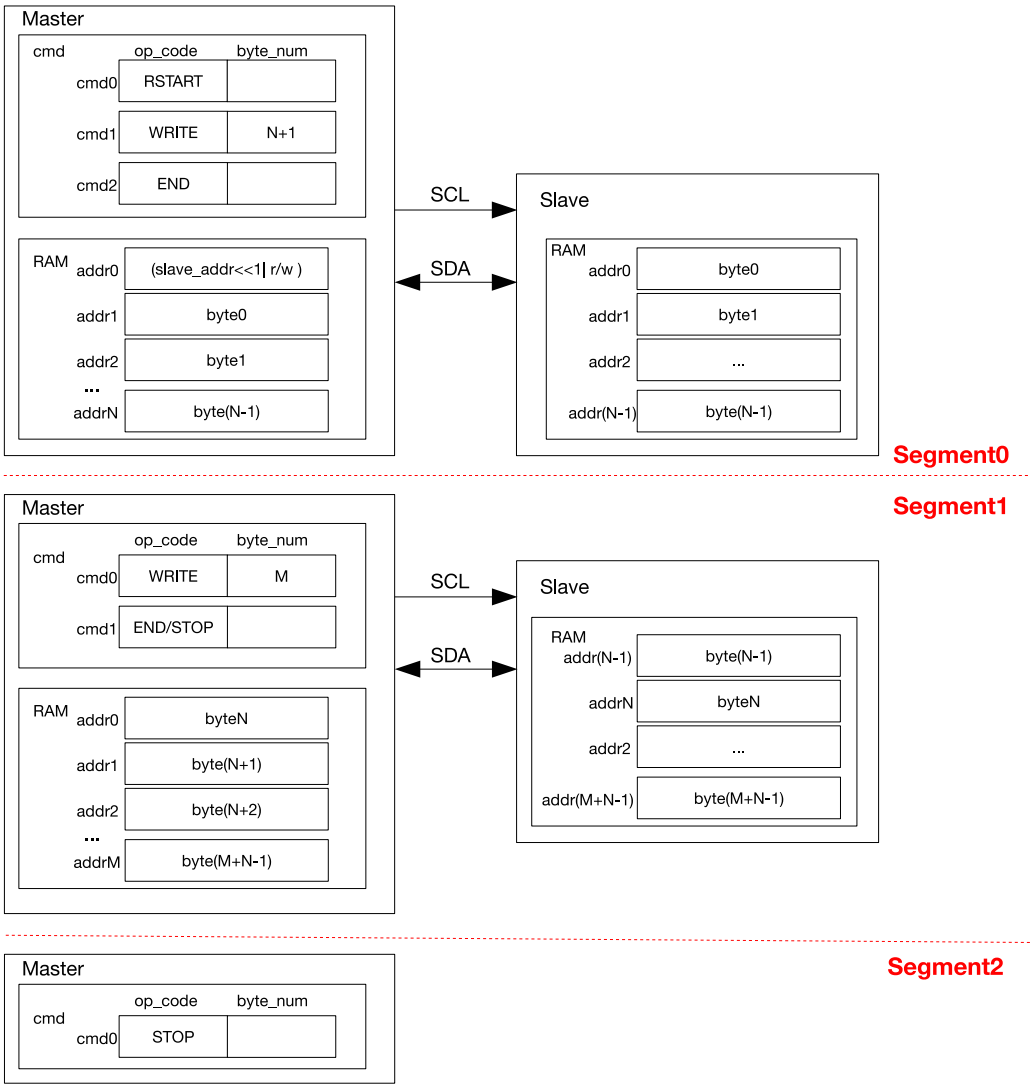



Figure 28.5-4. $I2C_{master}$ Writing to $I2C_{slave}$ with a 7-bit Address in Multiple Sequences

Given that the I2C Controller RAM holds only 32 bytes, when data are too large to be processed even by the wrapped RAM, it is advised to transmit them in multiple command sequences. At the end of every command sequence is an END command. When the controller executes this END command to pull SCL low, software refreshes command sequence registers and the RAM for next the transfer.

Figure 28.5-4 shows how $I2C_{master}$ writes to an I2C slave in two or three segments as an example. For the first segment, the CMD_Controller registers are configured as shown in Segment0. Once data in $I2C_{master}$'s RAM is ready and I2C_TRANS_START is set, $I2C_{master}$ initiates data transfer. After executing the END command, $I2C_{master}$ turns off the SCL clock and pulls 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, 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 $I2C_{slave}$ in two segments. $I2C_{master}$ resumes data transfer after I2C_TRANS_START is set, and terminates the transfer by sending a STOP bit.

For the third segment, after the second data transfer finishes and an I2C_END_DETECT_INT is detected, the CMD_Controller registers of $I2C_{master}$ are configured as shown in Segment2. Once I2C_TRANS_START is set, $I2C_{master}$ generates a STOP bit and terminates the transfer.

Note that other $I2C_{master}$s will not transact on the bus between two segments. The bus is only released after a STOP signal is sent. The I2C controller can be reset by setting I2C_FSM_RST field at any time. This field will later be cleared automatically by hardware.

#### 28.5.4.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
3. Configure command registers of $I2C_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | ack_value | ack_exp | 1 | N+1 |
| I2C_COMMAND2 (master) | END | — | — | — | — |

4. Write $I2C_{slave}$ address and data to be sent to TX RAM of $I2C_{master}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
5. Write address of $I2C_{slave}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register
6. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
7. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start transfer.
8. $I2C_{slave}$ compares the slave address sent by $I2C_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $I2C_{master}$'s WRITE command is 1, $I2C_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, $I2C_{master}$ does not check ACK value and take $I2C_{slave}$ as matching slave by default.
   - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $I2C_{master}$ continues data transfer.
   - Not match: If the received ACK value does not match ack_exp, $I2C_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
9. $I2C_{master}$ sends data, and checks ACK value or not according to ack_check_en (master).
10. After the I2C_END_DETECT_INT (master) interrupt is generated, set I2C_END_DETECT_INT_CLR (master) to 1 to clear this interrupt.
11. Update $I2C_{master}$'s command registers.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | WRITE | ack_value | ack_exp | 1 | M |
| I2C_COMMAND1 (master) | END/STOP | — | — | — | — |

12. Write M bytes of data to be sent to TX RAM of $I2C_{master}$ in FIFO or non-FIFO mode.
13. Write 1 to I2C_TRANS_START (master) bit to start transfer and repeat step 9.
14. If the command is a STOP, I2C stops transfer and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.
15. If the command is an END, repeat step 10.
16. Update $I2C_{master}$'s command registers.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND1 (master) | STOP | — | — | — | — |

17. Write 1 to I2C_TRANS_START (master) bit to start transfer.
18. $I2C_{master}$ executes the STOP command and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.5.5 $I2C_{master}$ Reads $I2C_{slave}$ with a 7-bit Address in One Command Sequence

### 28.5.5.1 Introduction

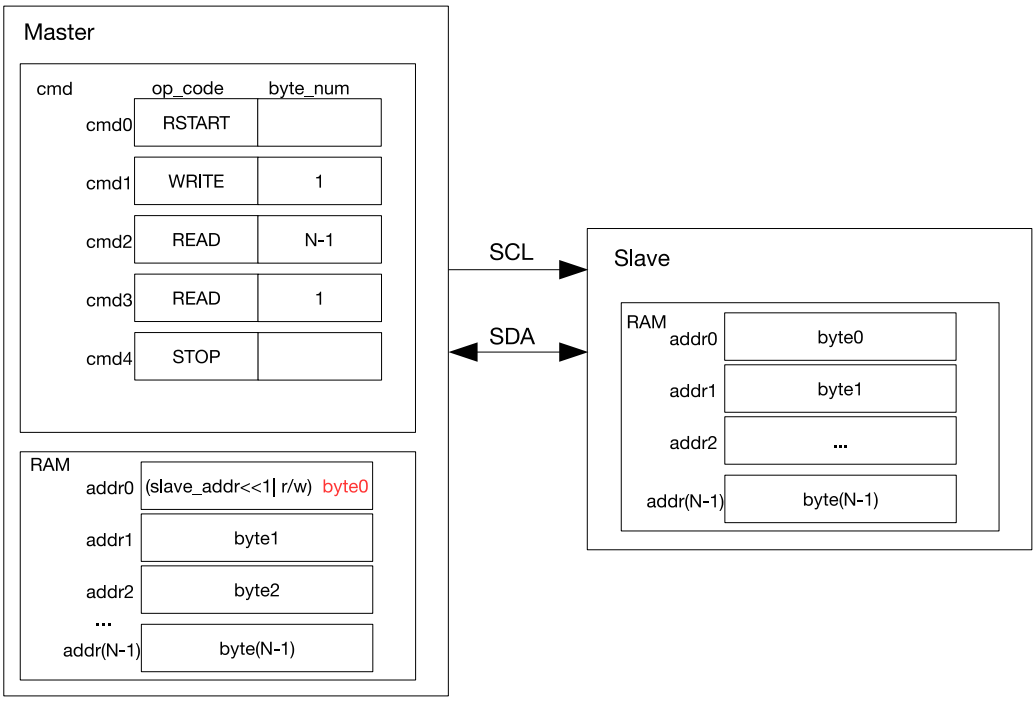



Figure 28.5-5. $I2C_{master}$ Reading $I2C_{slave}$ with a 7-bit Address

Figure 28.5-5 shows how $I2C_{master}$ reads N bytes of data from an I2C slave using 7-bit addressing. cmd1 is a WRITE command, and when this command is executed $I2C_{master}$ sends $I2C_{slave}$ address. The byte sent comprises a 7-bit $I2C_{slave}$ address and a $R/\overline{W}$ bit. When the $R/\overline{W}$ bit is 1, it indicates a READ operation. If the address of an I2C slave matches the sent address, this matching slave starts sending data to $I2C_{master}$. $I2C_{master}$ generates acknowledgements according to ack_value defined in the READ command upon receiving a byte.

As illustrated in Figure 28.5-5, $I2C_{master}$ 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. $I2C_{master}$ writes received data into the controller RAM from addr0, whose original content (a $I2C_{slave}$ address and a $R/\overline{W}$ bit) is overwritten by byte0 marked red in Figure 28.5-5.

#### 28.5.5.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. We recommend setting I2C_SLAVE_SCL_STRETCH_EN (slave) to 1, so that SCL can be held low for more processing time when $I2C_{slave}$ needs to send data. If this bit is not set, software should write data to be sent to $I2C_{slave}$'s TX RAM before $I2C_{master}$ initiates transfer. Configuration below is applicable to scenario where I2C_SLAVE_SCL_STRETCH_EN (slave) is 1.
3. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
4. Configure command registers of $I2C_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_en | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | 0 | 0 | 1 | 1 |
| I2C_COMMAND2 (master) | READ | 0 | 0 | 1 | N-1 |
| I2C_COMMAND3 (master) | READ | 1 | 0 | 1 | 1 |
| I2C_COMMAND4 (master) | STOP | — | — | — | — |

5. Write $I2C_{slave}$ address to TX RAM of $I2C_{master}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
6. Write address of $I2C_{slave}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register.
7. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
8. Write 1 to I2C_TRANS_START (master) bit to start $I2C_{master}$'s transfer.
9. Start $I2C_{slave}$'s transfer according to Section 28.4.14.
10. $I2C_{slave}$ compares the slave address sent by $I2C_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $I2C_{master}$'s WRITE command is 1, $I2C_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, $I2C_{master}$ does not check ACK value and take $I2C_{slave}$ as matching slave by default.
    - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $I2C_{master}$ continues data transfer.

- Not match: If the received ACK value does not match ack_exp, $I2C_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.

11. After I2C_SLAVE_STRETCH_INT (slave) is generated, the I2C_STRETCH_CAUSE bit is 0. The $I2C_{slave}$ address matches the address sent over SDA, and $I2C_{slave}$ needs to send data.
12. Write data to be sent to TX RAM of $I2C_{slave}$ according to Section 28.4.10.
13. Set I2C_SLAVE_SCL_STRETCH_CLR (slave) to 1 to release SCL.
14. $I2C_{slave}$ sends data, and $I2C_{master}$ checks ACK value or not according to ack_check_en (master) in the READ command.
15. If data to be read by $I2C_{master}$ is larger than 32 bytes, an I2C_SLAVE_STRETCH_INT (slave) interrupt will be generated when TX RAM of $I2C_{slave}$ becomes empty. In this way, $I2C_{slave}$ can hold SCL low, so that software has more time to pad data in TX RAM of $I2C_{slave}$ and read data in RX RAM of $I2C_{master}$. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
16. After $I2C_{master}$ has received the last byte of data, set ack_value (master) to 1. $I2C_{slave}$ will stop transfer once receiving the I2C_NACK_INT interrupt.
17. After data transfer completes, $I2C_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.5.6 $I2C_{master}$ Reads $I2C_{slave}$ with a 10-bit Address in One Command Sequence

### 28.5.6.1 Introduction

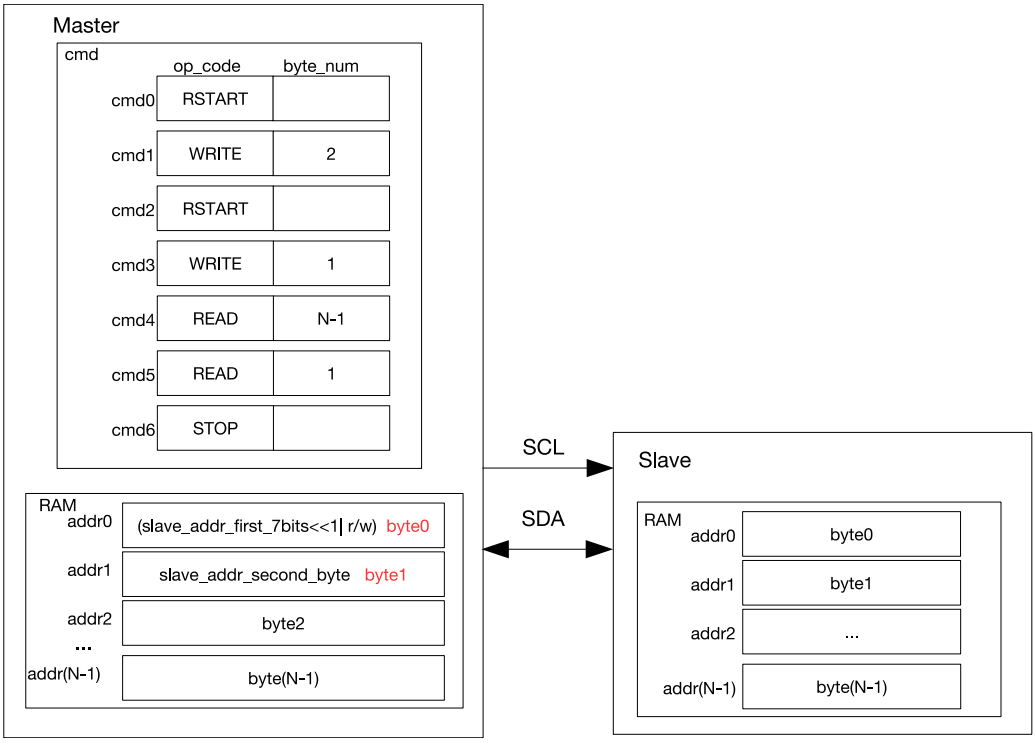



Figure 28.5-6. $I2C_{master}$ Reading $I2C_{slave}$ with a 10-bit Address

Figure 28.5-6 shows how $I2C_{master}$ reads data from an I2C slave using 10-bit addressing. Unlike 7-bit addressing, in 10-bit addressing the WRITE command of the $I2C_{master}$ is formed from two bytes, and correspondingly TX RAM of this master stores a 10-bit address of two bytes. The $R/\overline{W}$ bit in the first byte is 0, which indicates a WRITE operation. After a RSTART condition, $I2C_{master}$ sends the first byte of address again to read data from $I2C_{slave}$, but the $R/\overline{W}$ bit is 1, which indicates a READ operation. The two address bytes can be configured as described in Section 28.5.2.

#### 28.5.6.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. We recommend setting I2C_SLAVE_SCL_STRETCH_EN (slave) to 1, so that SCL can be held low for more processing time when $I2C_{slave}$ needs to send data. If this bit is not set, software should write data to be sent to $I2C_{slave}$'s TX RAM before $I2C_{master}$ initiates transfer. Configuration below is applicable to scenario where I2C_SLAVE_SCL_STRETCH_EN (slave) is 1.
3. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
4. Configure command registers of $I2C_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | 0 | 0 | 1 | 2 |
| I2C_COMMAND2 (master) | RSTART | — | — | — | — |
| I2C_COMMAND3 (master) | WRITE | 0 | 0 | 1 | 1 |
| I2C_COMMAND4 (master) | READ | 0 | 0 | 1 | N-1 |
| I2C_COMMAND5 (master) | READ | 1 | 0 | 1 | 1 |
| I2C_COMMAND6 (master) | STOP | — | — | — | — |

5. Configure I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) as $I2C_{slave}$'s 10-bit address, and set
   I2C_ADDR_10BIT_EN (slave) to 1 to enable 10-bit addressing.
6. Write $I2C_{slave}$ address and data to be sent to TX RAM of $I2C_{master}$ in either FIFO or non-FIFO mode. The first byte of address comprises ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a $R/\overline{W}$ bit, which is 1 and indicates a WRITE operation. The second byte of address is I2C_SLAVE_ADDR[7:0]. The third byte is ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a $R/\overline{W}$ bit, which is 1 and indicates a READ operation.
7. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
8. Write 1 to I2C_TRANS_START (master) to start $I2C_{master}$'s transfer.
9. Start $I2C_{slave}$'s transfer according to Section 28.4.14.
10. $I2C_{slave}$ compares the slave address sent by $I2C_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $I2C_{master}$'s WRITE command is 1, $I2C_{master}$ checks ACK value each time it

sends a byte. When ack_check_en (master) is 0, $I2C_{master}$ does not check ACK value and take $I2C_{slave}$ as matching slave by default.

- Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $I2C_{master}$ continues data transfer.
- Not match: If the received ACK value does not match ack_exp, $I2C_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.

11. $I2C_{master}$ sends a RSTART and the third byte in TX RAM, which is ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a $R/\overline{W}$ bit that indicates READ.
12. $I2C_{slave}$ repeats step 10. If its address matches the address sent by $I2C_{master}$, $I2C_{slave}$ proceed on to the next steps.
13. After I2C_SLAVE_STRETCH_INT (slave) is generated, the I2C_STRETCH_CAUSE bit is 0. The $I2C_{slave}$ address matches the address sent over SDA, and $I2C_{slave}$ needs to send data.
14. Write data to be sent to TX RAM of $I2C_{slave}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
15. Set I2C_SLAVE_SCL_STRETCH_CLR (slave) to 1 to release SCL.
16. $I2C_{slave}$ sends data, and $I2C_{master}$ checks ACK value or not according to ack_check_en (master) in the READ command.
17. If data to be read by $I2C_{master}$ is larger than 32 bytes, an I2C_SLAVE_STRETCH_INT (slave) interrupt will be generated when TX RAM of $I2C_{slave}$ becomes empty. In this way, $I2C_{slave}$ can hold SCL low, so that software has more time to pad data in TX RAM of $I2C_{slave}$ and read data in RX RAM of $I2C_{master}$. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
18. After $I2C_{master}$ has received the last byte of data, set ack_value (master) to 1. $I2C_{slave}$ will stop transfer once receiving the I2C_NACK_INT interrupt.
19. After data transfer completes, $I2C_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

### 28.5.7 $I2C_{master}$ Reads $I2C_{slave}$ with Two 7-bit Addresses in One Command Sequence

### 28.5.7.1 Introduction

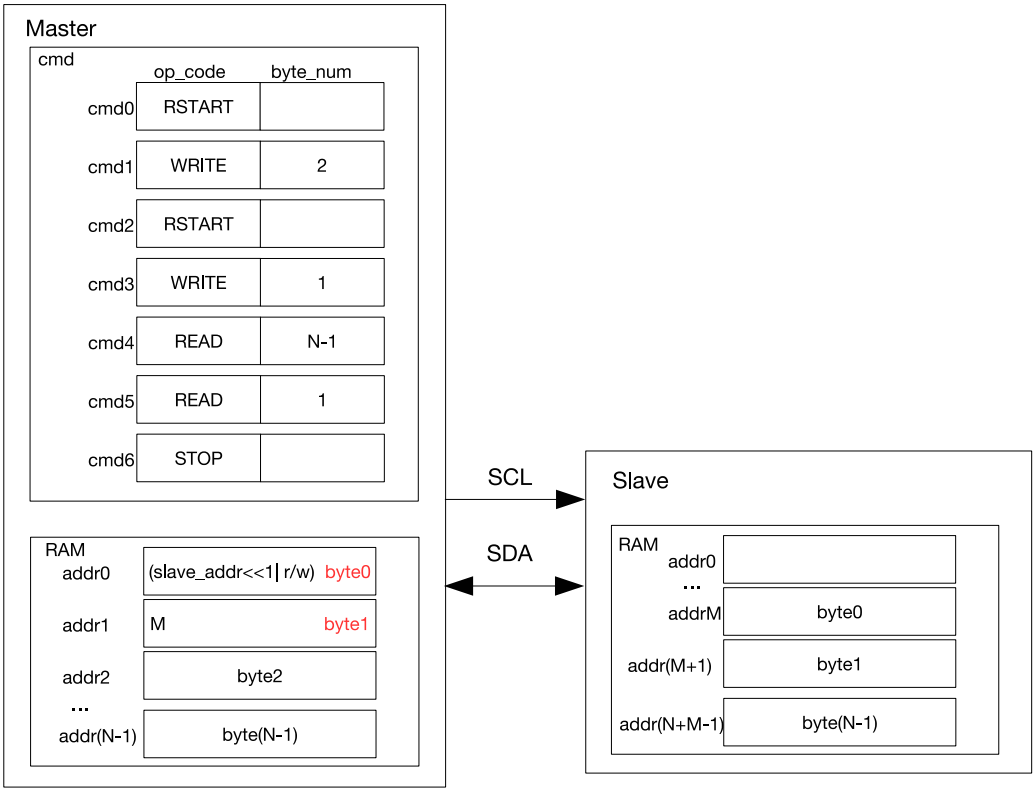



**Figure 28.5-7. I2C$_{master}$ Reading N Bytes of Data from addrM of I2C$_{slave}$ with a 7-bit Address**

Figure 28.5-7 shows how I2C$_{master}$ reads data from specified addresses in an I2C slave. I2C$_{master}$ sends two bytes of addresses: the first byte is a 7-bit I2C$_{slave}$ address followed by a $R/\overline{W}$ bit, which is 0 and indicates a WRITE; the second byte is I2C$_{slave}$'s memory address. After a RSTART condition, I2C$_{master}$ sends the first byte of address again, but the $R/\overline{W}$ bit is 1 which indicates a READ. Then, I2C$_{master}$ reads data starting from addrM.

### 28.5.7.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. We recommend setting I2C_SLAVE_SCL_STRETCH_EN (slave) to 1, so that SCL can be held low for more processing time when I2C$_{slave}$ needs to send data. If this bit is not set, software should write data to be sent to I2C$_{slave}$'s TX RAM before I2C$_{master}$ initiates transfer. Configuration below is applicable to scenario where I2C_SLAVE_SCL_STRETCH_EN (slave) is 1.
3. Set I2C_FIFO_ADDR_CFG_EN (slave) to 1 to enable double addressing mode.
4. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
5. Configure command registers of I2C$_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | 0 | 0 | 1 | 2 |

| | | | | | |
|---|---|---|---|---|---|
| I2C_COMMAND2 (master) | RSTART | — | — | — | — |
| I2C_COMMAND3 (master) | WRITE | 0 | 0 | 1 | 1 |
| I2C_COMMAND4 (master) | READ | 0 | 0 | 1 | N-1 |
| I2C_COMMAND5 (master) | READ | 1 | 0 | 1 | 1 |
| I2C_COMMAND6 (master) | STOP | — | — | — | — |

6. Configure I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register as $I2C_{slave}$'s 7-bit address, and set I2C_ADDR_10BIT_EN (slave) to 0 to enable 7-bit addressing.
7. Write $I2C_{slave}$ address and data to be sent to TX RAM of $I2C_{master}$ in either FIFO or non-FIFO mode according to Section 28.4.10. The first byte of address comprises ( I2C_SLAVE_ADDR[6:0])«1) and a $R/\overline{W}$ bit, which is 0 and indicates a WRITE. The second byte of address is memory address M of $I2C_{slave}$. The third byte is ( I2C_SLAVE_ADDR[6:0])«1) and a $R/\overline{W}$ bit, which is 1 and indicates a READ.
8. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
9. Write 1 to I2C_TRANS_START (master) and I2C_TRANS_START (slave) to start $I2C_{master}$'s transfer.
10. Start $I2C_{slave}$'s transfer according to Section 28.4.14.
11. $I2C_{slave}$ compares the slave address sent by $I2C_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $I2C_{master}$'s WRITE command is 1, $I2C_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, $I2C_{master}$ does not check ACK value and take $I2C_{slave}$ as matching slave by default.
    - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $I2C_{master}$ continues data transfer.
    - Not match: If the received ACK value does not match ack_exp, $I2C_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
12. $I2C_{slave}$ receives memory address sent by $I2C_{master}$ and adds the offset.
13. $I2C_{master}$ sends a RSTART and the third byte in TX RAM, which is ((0x78 | I2C_SLAVE_ADDR[9:8])«1) and a R bit.
14. $I2C_{slave}$ repeats step 11. If its address matches the address sent by $I2C_{master}$, $I2C_{slave}$ proceed on to the next steps.
15. After I2C_SLAVE_STRETCH_INT (slave) is generated, the I2C_STRETCH_CAUSE bit is 0. The $I2C_{slave}$ address matches the address sent over SDA, and $I2C_{slave}$ needs to send data.
16. Write data to be sent to TX RAM of $I2C_{slave}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.
17. Set I2C_SLAVE_SCL_STRETCH_CLR (slave) to 1 to release SCL.
18. $I2C_{slave}$ sends data, and $I2C_{master}$ checks ACK value or not according to ack_check_en (master) in the READ command.

19. If data to be read by $I2C_{master}$ is larger than 32 bytes, an I2C_SLAVE_STRETCH_INT (slave) interrupt will be generated when TX RAM of $I2C_{slave}$ becomes empty. In this way, $I2C_{slave}$ can hold SCL low, so that software has more time to pad data in TX RAM of $I2C_{slave}$ and read data in RX RAM of $I2C_{master}$. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
20. After $I2C_{master}$ has received the last byte of data, set ack_value (master) to 1. $I2C_{slave}$ will stop transfer once receiving the I2C_NACK_INT interrupt.
21. After data transfer completes, $I2C_{master}$ executes the STOP command, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.5.8 $I2C_{master}$ Reads $I2C_{slave}$ with a 7-bit Address in Multiple Command Sequences

### 28.5.8.1 Introduction

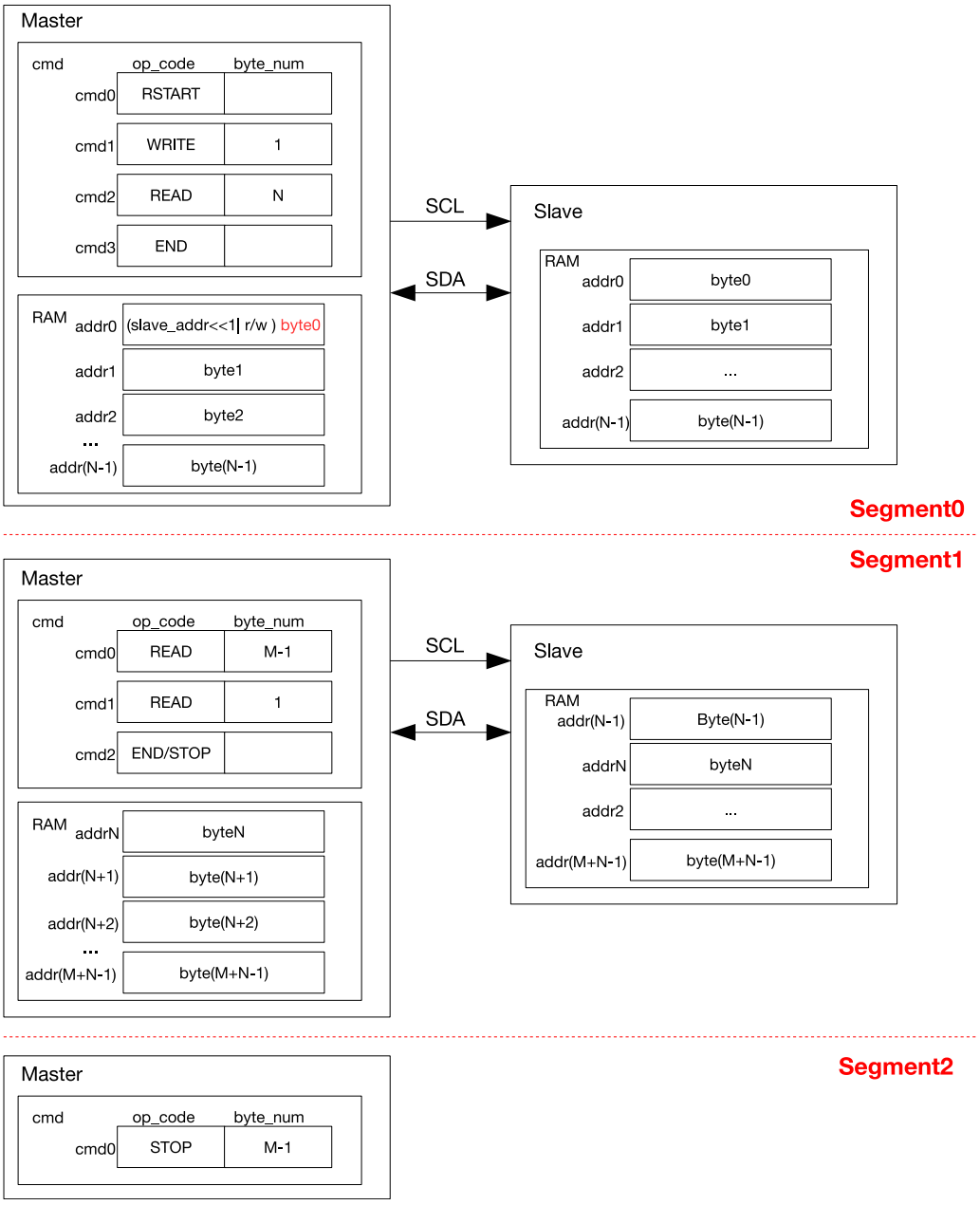



Figure 28.5-8. $I2C_{master}$ Reading $I2C_{slave}$ with a 7-bit Address in Segments

Figure 28.5-8 shows how $I2C_{master}$ reads (N+M) bytes of data from an I2C slave in two/three segments separated by END commands. Configuration procedures are described as follows:

1. The procedures for Segment0 is similar to Figure 28.5-5, except that the last command is an END.
2. Prepare data in the TX RAM of $I2C_{slave}$, and set I2C_TRANS_START to start data transfer. After executing the END command, $I2C_{master}$ refreshes command registers and the RAM as shown in Segment1, and clears the corresponding I2C_END_DETECT_INT interrupt. If cmd2 in Segment1 is a STOP, then data is

read from $I2C_{slave}$ in two segments. $I2C_{master}$ 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 $I2C_{slave}$ 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, $I2C_{master}$ terminates the transfer by sending a STOP bit.

#### 28.5.8.2 Configuration Example

1. Set I2C_MS_MODE (master) to 1, and I2C_MS_MODE (slave) to 0.
2. We recommend setting I2C_SLAVE_SCL_STRETCH_EN (slave) to 1, so that SCL can be held low for more processing time when $I2C_{slave}$ needs to send data. If this bit is not set, software should write data to be sent to $I2C_{slave}$'s TX RAM before $I2C_{master}$ initiates transfer. Configuration below is applicable to scenario where I2C_SLAVE_SCL_STRETCH_EN (slave) is 1.
3. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
4. Configure command registers of $I2C_{master}$.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | RSTART | — | — | — | — |
| I2C_COMMAND1 (master) | WRITE | 0 | 0 | 1 | 1 |
| I2C_COMMAND2 (master) | READ | 0 | 0 | 1 | N |
| I2C_COMMAND3 (master) | END | — | — | — | — |

5. Write $I2C_{slave}$ address to TX RAM of $I2C_{master}$ in FIFO or non-FIFO mode.
6. Write address of $I2C_{slave}$ to I2C_SLAVE_ADDR (slave) in I2C_SLAVE_ADDR_REG (slave) register.
7. Write 1 to I2C_CONF_UPGATE (master) and I2C_CONF_UPGATE (slave) to synchronize registers.
8. Write 1 to I2C_TRANS_START (master) to start $I2C_{master}$'s transfer.
9. Start $I2C_{slave}$'s transfer according to Section 28.4.14.
10. $I2C_{slave}$ compares the slave address sent by $I2C_{master}$ with its own address in I2C_SLAVE_ADDR (slave). When ack_check_en (master) in $I2C_{master}$'s WRITE command is 1, $I2C_{master}$ checks ACK value each time it sends a byte. When ack_check_en (master) is 0, $I2C_{master}$ does not check ACK value and take $I2C_{slave}$ as matching slave by default.
    - Match: If the received ACK value matches ack_exp (master) (the expected ACK value), $I2C_{master}$ continues data transfer.
    - Not match: If the received ACK value does not match ack_exp, $I2C_{master}$ generates an I2C_NACK_INT (master) interrupt and stops data transfer.
11. After I2C_SLAVE_STRETCH_INT (slave) is generated, the I2C_STRETCH_CAUSE bit is 0. The $I2C_{slave}$ address matches the address sent over SDA, and $I2C_{slave}$ needs to send data.
12. Write data to be sent to TX RAM of $I2C_{slave}$ in either FIFO mode or non-FIFO mode according to Section 28.4.10.

13. Set I2C_SLAVE_SCL_STRETCH_CLR (slave) to 1 to release SCL.
14. $I2C_{slave}$ sends data, and $I2C_{master}$ checks ACK value or not according to ack_check_en (master) in the READ command.
15. If data to be read by $I2C_{master}$ in one READ command (N or M) is larger than 32 bytes, an I2C_SLAVE_STRETCH_INT (slave) interrupt will be generated when TX RAM of $I2C_{slave}$ becomes empty. In this way, $I2C_{slave}$ can hold SCL low, so that software has more time to pad data in TX RAM of $I2C_{slave}$ and read data in RX RAM of $I2C_{master}$. After software has finished reading, you can set I2C_SLAVE_STRETCH_INT_CLR (slave) to 1 to clear interrupt, and set I2C_SLAVE_SCL_STRETCH_CLR (slave) to release the SCL line.
16. Once finishing reading data in the first READ command, $I2C_{master}$ executes the END command and triggers an I2C_END_DETECT_INT (master) interrupt, which is cleared by setting I2C_END_DETECT_INT_CLR (master) to 1.
17. Update $I2C_{master}$'s command registers using one of the following two methods:

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | READ | ack_value | ack_exp | 1 | M |
| I2C_COMMAND1 (master) | END | — | — | — | — |

Or

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND0 (master) | READ | 0 | 0 | 1 | M-1 |
| I2C_COMMAND0 (master) | READ | 1 | 0 | 1 | 1 |
| I2C_COMMAND1 (master) | STOP | — | — | — | — |

18. Write M bytes of data to be sent to TX RAM of $I2C_{slave}$. If M is larger than 32, then repeat step 14 in FIFO or non-FIFO mode.
19. Write 1 to I2C_TRANS_START (master) bit to start transfer and repeat step 14.
20. If the last command is a STOP, then set ack_value (master) to 1 after $I2C_{master}$ has received the last byte of data. $I2C_{slave}$ stops transfer upon the I2C_NACK_INT interrupt. $I2C_{master}$ executes the STOP command to stop transfer and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.
21. If the last command is an END, then repeat step 16 and proceed on to the next steps.
22. Update $I2C_{master}$'s command registers.

| Command registers | op_code | ack_value | ack_exp | ack_check_er | byte_num |
|---|---|---|---|---|---|
| I2C_COMMAND1 (master) | STOP | — | — | — | — |

23. Write 1 to I2C_TRANS_START (master) bit to start transfer.
24. $I2C_{master}$ executes the STOP command to stop transfer, and generates an I2C_TRANS_COMPLETE_INT (master) interrupt.

## 28.6 Interrupts

- I2C_SLAVE_STRETCH_INT: Generated when one of the four stretching events occurs in slave mode.
- I2C_DET_START_INT: Triggered when the master or the slave detects a START bit.
- I2C_SCL_MAIN_ST_TO_INT: Triggered when the main state machine SCL_MAIN_FSM remains unchanged for over I2C_SCL_MAIN_ST_TO_I2C[23:0] clock cycles.
- I2C_SCL_ST_TO_INT: Triggered when the state machine SCL_FSM remains unchanged for over I2C_SCL_ST_TO_I2C[23:0] clock cycles.
- I2C_RXFIFO_UDF_INT: Triggered when the I2C controller reads RX FIFO via the APB bus, but RX FIFO is empty.
- I2C_TXFIFO_OVF_INT: Triggered when the I2C controller writes TX FIFO via the APB bus, but TX FIFO is full.
- I2C_NACK_INT: Triggered when the ACK value received by the master is not as expected, or when the ACK value received by the slave is 1.
- I2C_TRANS_START_INT: Triggered when the I2C controller sends a START bit.
- I2C_TIME_OUT_INT: Triggered when SCL stays high or low for more than $2^{I2C_TIME_OUT_VALUE}$ clock cycles during data transfer.
- I2C_TRANS_COMPLETE_INT: Triggered when the I2C controller detects a STOP bit.
- I2C_MST_TXFIFO_UDF_INT: Triggered when TX FIFO of the master underflows.
- I2C_ARBITRATION_LOST_INT: Triggered when the SDA's output value does not match its input value while the master's SCL is high.
- I2C_BYTE_TRANS_DONE_INT: Triggered when the I2C controller sends or receives a byte.
- I2C_END_DETECT_INT: Triggered when op_code of the master indicates an END command and an END condition is detected.
- I2C_RXFIFO_OVF_INT: Triggered when RX FIFO of the I2C controller overflows.
- I2C_TXFIFO_WM_INT: I2C TX FIFO watermark interrupt. Triggered when I2C_FIFO_PRT_EN is 1 and the pointers of TX FIFO are less than I2C_TXFIFO_WM_THRHD[4:0].
- I2C_RXFIFO_WM_INT: I2C RX FIFO watermark interrupt. Triggered when I2C_FIFO_PRT_EN is 1 and the pointers of RX FIFO are greater than I2C_RXFIFO_WM_THRHD[4:0].

## 28.7 Register Summary

The addresses in this section are relative to I2C Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Timing registers** | | | |
| I2C_SCL_LOW_PERIOD_REG | Configures the low level width of SCL | 0x0000 | R/W |
| I2C_SDA_HOLD_REG | Configures the hold time after a negative SCL edge | 0x0030 | R/W |
| I2C_SDA_SAMPLE_REG | Configures the sample time after a positive SCL edge | 0x0034 | R/W |
| I2C_SCL_HIGH_PERIOD_REG | Configures the high level width of SCL | 0x0038 | R/W |
| I2C_SCL_START_HOLD_REG | Configures the delay between the SDA and SCL negative edge for a START condition | 0x0040 | R/W |
| I2C_SCL_RSTART_SETUP_REG | Configures the delay between the positive edge of SCL and the negative edge of SDA | 0x0044 | R/W |
| I2C_SCL_STOP_HOLD_REG | Configures the delay after the SCL clock edge for a STOP condition | 0x0048 | R/W |
| I2C_SCL_STOP_SETUP_REG | Configures the delay between the SDA and SCL positive edge for a STOP condition | 0x004C | R/W |
| I2C_SCL_ST_TIME_OUT_REG | SCL status timeout register | 0x0078 | R/W |
| I2C_SCL_MAIN_ST_TIME_OUT_REG | SCL main status timeout register | 0x007C | R/W |
| **Configuration registers** | | | |
| I2C_CTR_REG | Transmission configuration register | 0x0004 | varies |
| I2C_TO_REG | Timeout control register | 0x000C | R/W |
| I2C_SLAVE_ADDR_REG | Slave address configuration register | 0x0010 | R/W |
| I2C_FIFO_CONF_REG | FIFO configuration register | 0x0018 | R/W |
| I2C_FILTER_CFG_REG | SCL and SDA filter configuration register | 0x0050 | R/W |
| I2C_CLK_CONF_REG | I2C clock configuration register | 0x0054 | R/W |
| I2C_SCL_SP_CONF_REG | Power configuration register | 0x0080 | varies |
| I2C_SCL_STRETCH_CONF_REG | Configures SCL clock stretching | 0x0084 | varies |
| **Status registers** | | | |
| I2C_SR_REG | Describes I2C work status | 0x0008 | RO |
| I2C_FIFO_ST_REG | FIFO status register | 0x0014 | RO |
| I2C_DATA_REG | Read/write FIFO register | 0x001C | R/W |
| **Interrupt registers** | | | |
| I2C_INT_RAW_REG | Raw interrupt status | 0x0020 | R/SS/WTC |
| I2C_INT_CLR_REG | Interrupt clear bits | 0x0024 | WT |
| I2C_INT_ENA_REG | Interrupt enable bits | 0x0028 | R/W |
| I2C_INT_STATUS_REG | Status of captured I2C communication events | 0x002C | RO |
| **Command registers** | | | |
| I2C_COMD0_REG | I2C command register 0 | 0x0058 | varies |

| Name | Description | Address | Access |
|---|---|---|---|
| I2C_COMD1_REG | I2C command register 1 | 0x005C | varies |
| I2C_COMD2_REG | I2C command register 2 | 0x0060 | varies |
| I2C_COMD3_REG | I2C command register 3 | 0x0064 | varies |
| I2C_COMD4_REG | I2C command register 4 | 0x0068 | varies |
| I2C_COMD5_REG | I2C command register 5 | 0x006C | varies |
| I2C_COMD6_REG | I2C command register 6 | 0x0070 | varies |
| I2C_COMD7_REG | I2C command register 7 | 0x0074 | varies |
| **Version register** | | | |
| I2C_DATE_REG | Version control register | 0x00F8 | R/W |

## 28.8 Registers

The addresses in this section are relative to I2C Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 28.1. I2C_SCL_LOW_PERIOD_REG (0x0000)**

| (reserved) | I2C_SCL_LOW_PERIOD | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 0 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 |

**I2C_SCL_LOW_PERIOD** This field is used to configure how long SCL remains low in master mode, in I2C module clock cycles. (R/W)

**Register 28.2. I2C_SDA_HOLD_REG (0x0030)**

| (reserved) | I2C_SDA_HOLD_TIME | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 0 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 |

**I2C_SDA_HOLD_TIME** This field is used to configure the time to hold the data after the falling edge of SCL, in I2C module clock cycles. (R/W)

**Register 28.3. I2C_SDA_SAMPLE_REG (0x0034)**

| (reserved) | I2C_SDA_SAMPLE_TIME | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 0 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 |

**I2C_SDA_SAMPLE_TIME** This field is used to configure how long SDA is sampled, in I2C module clock cycles. (R/W)

### Register 28.4. I2C_SCL_HIGH_PERIOD_REG (0x0038)

| (reserved) | I2C_SCL_WAIT_HIGH_PERIOD | I2C_SCL_HIGH_PERIOD | |
|---|---|---|---|
| 31 … 16 | 15 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | Reset |

**I2C_SCL_HIGH_PERIOD** This field is used to configure how long SCL remains high in master mode, in I2C module clock cycles. (R/W)

**I2C_SCL_WAIT_HIGH_PERIOD** This field is used to configure the SCL_FSM's waiting period for SCL high level in master mode, in I2C module clock cycles. (R/W)

### Register 28.5. I2C_SCL_START_HOLD_REG (0x0040)

| (reserved) | I2C_SCL_START_HOLD_TIME | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 8 | Reset |

**I2C_SCL_START_HOLD_TIME** This field is used to configure the time between the falling edge of SDA and the falling edge of SCL for a START condition, in I2C module clock cycles. (R/W)

### Register 28.6. I2C_SCL_RSTART_SETUP_REG (0x0044)

| (reserved) | I2C_SCL_RSTART_SETUP_TIME | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 8 | Reset |

**I2C_SCL_RSTART_SETUP_TIME** This field is used to configure the time between the rising edge of SCL and the falling edge of SDA for a RSTART condition, in I2C module clock cycles. (R/W)

**Register 28.7. I2C_SCL_STOP_HOLD_REG (0x0048)**

| (reserved) | I2C_SCL_STOP_HOLD_TIME | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 8 | Reset |

**I2C_SCL_STOP_HOLD_TIME** This field is used to configure the delay after the STOP condition, in I2C module clock cycles. (R/W)

**Register 28.8. I2C_SCL_STOP_SETUP_REG (0x004C)**

| (reserved) | I2C_SCL_STOP_SETUP_TIME | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 8 | Reset |

**I2C_SCL_STOP_SETUP_TIME** This field is used to configure the time between the rising edge of SCL and the rising edge of SDA, in I2C module clock cycles. (R/W)

**Register 28.9. I2C_SCL_ST_TIME_OUT_REG (0x0078)**

| (reserved) | I2C_SCL_ST_TO_I2C | |
|---|---|---|
| 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 | 0x10 | Reset |

**I2C_SCL_ST_TO_I2C** The maximum time that SCL_FSM remains unchanged. It should be no more than 23. (R/W)

**Register 28.10. I2C_SCL_MAIN_ST_TIME_OUT_REG (0x007C)**

| (reserved) | I2C_SCL_MAIN_ST_TO_I2C | |
|---|---|---|
| 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 | 0x10 | Reset |

**I2C_SCL_MAIN_ST_TO_I2C** The maximum time that SCL_MAIN_FSM remains unchanged. It should be no more than 23. (R/W)

**Register 28.11. I2C_CTR_REG (0x0004)**

| 31 | | | | | | | | | | | | | | | | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | I2C_ADDR_BROADCASTING_EN | I2C_ADDR_10BIT_RW_CHECK_EN | I2C_SLV_TX_AUTO_START_EN | I2C_CONF_UPGATE | I2C_FSM_RST | I2C_ARBITRATION_EN | I2C_CLK_EN | I2C_RX_LSB_FIRST | I2C_TX_LSB_FIRST | I2C_TRANS_START | I2C_MS_MODE | I2C_RX_FULL_ACK_LEVEL | I2C_SAMPLE_SCL_LEVEL | I2C_SCL_FORCE_OUT | I2C_SDA_FORCE_OUT | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | Reset |

**I2C_SDA_FORCE_OUT** Configures the SDA output mode.

0: Open drain output

1: Direct output

(R/W)

**I2C_SCL_FORCE_OUT** Configures the SCL output mode.

0: Open drain output

1: Direct output

(R/W)

**I2C_SAMPLE_SCL_LEVEL** This bit is used to select the sampling mode. 0: samples SDA data on the SCL high level; 1: samples SDA data on the SCL low level. (R/W)

**I2C_RX_FULL_ACK_LEVEL** This bit is used to configure the ACK value that need to be sent by master when I2C_RXFIFO_CNT has reached the threshold. (R/W)

**I2C_MS_MODE** Set this bit to configure the I2C controller as an I2C Master. Clear this bit to configure the I2C controller as a slave. (R/W)

**I2C_TRANS_START** Set this bit to start sending the data in TX FIFO. (WT)

**I2C_TX_LSB_FIRST** This bit is used to control the order to send data. 0: sends data from the most significant bit; 1: sends data from the least significant bit. (R/W)

**I2C_RX_LSB_FIRST** This bit is used to control the order to receive data. 0: receives data from the most significant bit; 1: receives data from the least significant bit. (R/W)

**I2C_CLK_EN** This field controls APB_CLK clock gating. 0: APB_CLK is gated to save power; 1: APB_CLK is always on. (R/W)

**I2C_ARBITRATION_EN** This is the enable bit for I2C bus arbitration function. (R/W)

**I2C_FSM_RST** This bit is used to reset the SCL_FSM. (WT)

**I2C_CONF_UPGATE** Synchronization bit. (WT)

**I2C_SLV_TX_AUTO_START_EN** This is the enable bit for slave to send data automatically. (R/W)

**I2C_ADDR_10BIT_RW_CHECK_EN** This is the enable bit to check if the R/W bit of 10-bit addressing is consistent with the I2C protocol. (R/W)

**I2C_ADDR_BROADCASTING_EN** This is the enable bit for 7-bit general call addressing. (R/W)

**Register 28.12. I2C_TO_REG (0x000C)**

| (reserved) | I2C_TIME_OUT_EN | I2C_TIME_OUT_VALUE | |
|---|---|---|---|
| 31 … 6 | 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 | 0x10 | Reset |

**I2C_TIME_OUT_VALUE** This field is used to configure the timeout value for receiving a data bit in I2C_SCLK clock cycles. The configured timeout value equals $2^{I2C_TIME_OUT_VALUE}$ clock cycles. (R/W)

**I2C_TIME_OUT_EN** This is the enable bit for timeout control. (R/W)

**Register 28.13. I2C_SLAVE_ADDR_REG (0x0010)**

| I2C_ADDR_10BIT_EN | (reserved) | I2C_SLAVE_ADDR | |
|---|---|---|---|
| 31 | 30 … 15 | 14 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_SLAVE_ADDR** When the I2C controller is in slave mode, this field is used to configure the slave address. (R/W)

**I2C_ADDR_10BIT_EN** This field is used to enable the 10-bit addressing mode in master mode. (R/W)

**Register 28.14. I2C_FIFO_CONF_REG (0x0018)**

| (reserved) | I2C_FIFO_PRT_EN | I2C_TX_FIFO_RST | I2C_RX_FIFO_RST | I2C_FIFO_ADDR_CFG_EN | I2C_NONFIFO_EN | I2C_TXFIFO_WM_THRHD | I2C_RXFIFO_WM_THRHD | |
|---|---|---|---|---|---|---|---|---|
| 31 – 15 | 14 | 13 | 12 | 11 | 10 | 9 – 5 | 4 – 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 | 0 | 0 | 0 | 0 | 0x4 | 0xb | Reset |

**I2C_RXFIFO_WM_THRHD** The watermark 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 is valid. (R/W)

**I2C_TXFIFO_WM_THRHD** The watermark 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 is 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_FIFO_PRT_EN** The control enable bit of FIFO pointer in non-FIFO mode. This bit controls the valid bits and TX/RX FIFO overflow, underflow, full and empty interrupts. (R/W)

**Register 28.15. I2C_FILTER_CFG_REG (0x0050)**

| (reserved) | I2C_SDA_FILTER_EN | I2C_SCL_FILTER_EN | I2C_SDA_FILTER_THRES | I2C_SCL_FILTER_THRES | |
|---|---|---|---|---|---|
| 31 – 10 | 9 | 8 | 7 – 4 | 3 – 0 | |
| 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 | Reset |

**I2C_SCL_FILTER_THRES** When a pulse on the SCL input has smaller width than the value of this field in I2C module clock cycles, the I2C controller ignores that pulse. (R/W)

**I2C_SDA_FILTER_THRES** When a pulse on the SDA input has smaller width than the value of this field in I2C module clock cycles, the I2C controller ignores that pulse. (R/W)

**I2C_SCL_FILTER_EN** This is the filter enable bit for SCL. (R/W)

**I2C_SDA_FILTER_EN** This is the filter enable bit for SDA. (R/W)

**Register 28.16. I2C_CLK_CONF_REG (0x0054)**

| (reserved) | I2C_SCLK_ACTIVE | I2C_SCLK_SEL | I2C_SCLK_DIV_B | I2C_SCLK_DIV_A | I2C_SCLK_DIV_NUM | |
|---|---|---|---|---|---|---|
| 31 … 22 | 21 | 20 | 19 … 14 | 13 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 | 1 | 0 | 0 | 0 | 0 | Reset |

**I2C_SCLK_DIV_NUM** The integral part of the divisor. (R/W)

**I2C_SCLK_DIV_A** The numerator of the divisor's fractional part. (R/W)

**I2C_SCLK_DIV_B** The denominator of the divisor's fractional part. (R/W)

**I2C_SCLK_SEL** The clock selection bit for the I2C controller. 0: XTAL_CLK; 1: RC_FAST_CLK. (R/W)

**I2C_SCLK_ACTIVE** The clock switch bit for the I2C controller. (R/W)

**Register 28.17. I2C_SCL_SP_CONF_REG (0x0080)**

| (reserved) | I2C_SDA_PD_EN | I2C_SCL_PD_EN | I2C_SCL_RST_SLV_NUM | I2C_SCL_RST_SLV_EN | |
|---|---|---|---|---|---|
| 31 … 8 | 7 | 6 | 5 … 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 | Reset |

**I2C_SCL_RST_SLV_EN** When the master is idle, set this bit to send out SCL pulses. The number of pulses equals to I2C_SCL_RST_SLV_NUM[4:0]. (R/W/SC)

**I2C_SCL_RST_SLV_NUM** Configures the pulses of SCL generated in master mode. Valid when I2C_SCL_RST_SLV_EN is 1. (R/W)

**I2C_SCL_PD_EN** The power down enable bit for the I2C output SCL line. 0: Not power down; 1: Power down. Set I2C_SCL_FORCE_OUT and I2C_SCL_PD_EN to 1 to stretch SCL low. (R/W)

**I2C_SDA_PD_EN** The power down enable bit for the I2C output SDA line. 0: Not power down; 1: Power down. Set I2C_SDA_FORCE_OUT and I2C_SDA_PD_EN to 1 to stretch SDA low. (R/W)

**Register 28.18. I2C_SCL_STRETCH_CONF_REG (0x0084)**

| (reserved) | I2C_SLAVE_BYTE_ACK_LVL | I2C_SLAVE_BYTE_ACK_CTL_EN | I2C_SLAVE_SCL_STRETCH_CLR | I2C_SLAVE_SCL_STRETCH_EN | I2C_STRETCH_PROTECT_NUM | |
|---|---|---|---|---|---|---|
| 31 … 14 | 13 | 12 | 11 | 10 | 9 … 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 |

**I2C_STRETCH_PROTECT_NUM** Configures the time period to release the SCL line from stretching to avoid timing violation. Usually it should be larger than the SDA steup time. (R/W)

**I2C_SLAVE_SCL_STRETCH_EN** The enable bit for SCL clock stretching. 0: Disable; 1: Enable. The SCL output line will be stretched low when I2C_SLAVE_SCL_STRETCH_EN is 1 and one of the four stretching events occurs. The cause of stretching can be seen in I2C_STRETCH_CAUSE. (R/W)

**I2C_SLAVE_SCL_STRETCH_CLR** Set this bit to clear SCL clock stretching. (WT)

**I2C_SLAVE_BYTE_ACK_CTL_EN** The enable bit for slave to control the level of the ACK bit. (R/W)

**I2C_SLAVE_BYTE_ACK_LVL** Set the level of the ACK bit when I2C_SLAVE_BYTE_ACK_CTL_EN is set. (R/W)

**Register 28.19. I2C_SR_REG (0x0008)**

| (reserved) | I2C_SCL_STATE_LAST | (reserved) | I2C_SCL_MAIN_STATE_LAST | I2C_TXFIFO_CNT | (reserved) | I2C_STRETCH_CAUSE | I2C_RXFIFO_CNT | (reserved) | I2C_SLAVE_ADDRESSED | I2C_BUS_BUSY | I2C_ARB_LOST | (reserved) | I2C_SLAVE_RW | I2C_RESP_REC | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 28 | 27 | 26 24 | 23 18 | 17 16 | 15 14 | 13 8 | 7 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 0 | 0x3 | 0 | 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**I2C_RESP_REC** The received ACK value in master mode or slave mode. 0: ACK; 1: NACK. (RO)

**I2C_SLAVE_RW** When in slave mode, 0: master writes to slave; 1: master reads from slave. (RO)

**I2C_ARB_LOST** When the I2C controller loses control of the SCL line, this bit changes to 1. (RO)

**I2C_BUS_BUSY** 0: The I2C bus is in idle state; 1: The I2C bus is busy transferring data. (RO)

**I2C_SLAVE_ADDRESSED** When the I2C controller is in slave mode, and the address sent by the master matches the address of the slave, this bit is at high level. (RO)

**I2C_RXFIFO_CNT** This field represents the number of data bytes to be sent. (RO)

**I2C_STRETCH_CAUSE** The cause of SCL clock stretching in slave mode. 0: stretching SCL low when the master starts to read data; 1: stretching SCL low when TX FIFO is empty in slave mode; 2: stretching SCL low when RX FIFO is full in slave mode. (RO)

**I2C_TXFIFO_CNT** This field stores the number of data bytes received in RAM. (RO)

**I2C_SCL_MAIN_STATE_LAST** This field indicates the status of the state machine. 0: idle; 1: address shift; 2: ACK address; 3: receive data; 4: transmit data; 5: send ACK; 6: wait for ACK. (RO)

**I2C_SCL_STATE_LAST** This field indicates the status of the state machine used to produce SCL. 0: idle; 1: start; 2: falling edge; 3: low; 4: rising edge; 5: high; 6: stop. (RO)

**Register 28.20. I2C_FIFO_ST_REG (0x0014)**

| (reserved) | I2C_SLAVE_RW_POINT | (reserved) | I2C_TXFIFO_WADDR | I2C_TXFIFO_RADDR | I2C_RXFIFO_WADDR | I2C_RXFIFO_RADDR | |
|---|---|---|---|---|---|---|---|
| 31 30 | 29 22 | 21 20 | 19 15 | 14 10 | 9 5 | 4 0 | |
| 0 0 | 0 | 0 0 | 0 | 0 | 0 | 0 | Reset |

**I2C_RXFIFO_RADDR** This is the offset address of the APB reading from RX FIFO. (RO)

**I2C_RXFIFO_WADDR** This is the offset address of the I2C controller receiving data and writing to RX FIFO. (RO)

**I2C_TXFIFO_RADDR** This is the offset address of the I2C controller reading from TX FIFO. (RO)

**I2C_TXFIFO_WADDR** This is the offset address of APB bus writing to TX FIFO. (RO)

**I2C_SLAVE_RW_POINT** The received data in I2C slave mode. (RO)

**Register 28.21. I2C_DATA_REG (0x001C)**

| (reserved) | I2C_FIFO_RDATA | |
|---|---|---|
| 31 8 | 7 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_FIFO_RDATA** This field is used to read data from RX FIFO, or write data to TX FIFO. (R/W)

**Register 28.22. I2C_INT_RAW_REG (0x0020)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | I2C_GENERAL_CALL_INT_RAW | I2C_SLAVE_STRETCH_INT_RAW | I2C_DET_START_INT_RAW | I2C_SCL_MAIN_ST_TO_INT_RAW | I2C_SCL_ST_TO_INT_RAW | I2C_RXFIFO_UDF_INT_RAW | I2C_TXFIFO_OVF_INT_RAW | I2C_NACK_INT_RAW | I2C_TRANS_START_INT_RAW | I2C_TIME_OUT_INT_RAW | I2C_TRANS_COMPLETE_INT_RAW | I2C_MST_TXFIFO_UDF_INT_RAW | I2C_ARBITRATION_LOST_INT_RAW | I2C_BYTE_TRANS_DONE_INT_RAW | I2C_END_DETECT_INT_RAW | I2C_RXFIFO_OVF_INT_RAW | I2C_TXFIFO_WM_INT_RAW | I2C_RXFIFO_WM_INT_RAW | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | Reset |

**I2C_RXFIFO_WM_INT_RAW** The raw interrupt bit for the I2C_RXFIFO_WM_INT interrupt. (R/SS/WTC)

**I2C_TXFIFO_WM_INT_RAW** The raw interrupt bit for the I2C_TXFIFO_WM_INT interrupt. (R/SS/WTC)

**I2C_RXFIFO_OVF_INT_RAW** The raw interrupt bit for the I2C_RXFIFO_OVF_INT interrupt. (R/SS/WTC)

**I2C_END_DETECT_INT_RAW** The raw interrupt bit for the I2C_END_DETECT_INT interrupt. (R/SS/WTC)

**I2C_BYTE_TRANS_DONE_INT_RAW** The raw interrupt bit for the I2C_BYTE_TRANS_DONE_INT interrupt. (R/SS/WTC)

**I2C_ARBITRATION_LOST_INT_RAW** The raw interrupt bit for the I2C_ARBITRATION_LOST_INT interrupt. (R/SS/WTC)

**I2C_MST_TXFIFO_UDF_INT_RAW** The raw interrupt bit for the I2C_MST_TXFIFO_UDF_INT interrupt. (R/SS/WTC)

**I2C_TRANS_COMPLETE_INT_RAW** The raw interrupt bit for the I2C_TRANS_COMPLETE_INT interrupt. (R/SS/WTC)

**I2C_TIME_OUT_INT_RAW** The raw interrupt bit for the I2C_TIME_OUT_INT interrupt. (R/SS/WTC)

**I2C_TRANS_START_INT_RAW** The raw interrupt bit for the I2C_TRANS_START_INT interrupt. (R/SS/WTC)

**I2C_NACK_INT_RAW** The raw interrupt bit for the I2C_NACK_INT interrupt. (R/SS/WTC)

**I2C_TXFIFO_OVF_INT_RAW** The raw interrupt bit for the I2C_TXFIFO_OVF_INT interrupt. (R/SS/WTC)

**I2C_RXFIFO_UDF_INT_RAW** The raw interrupt bit for the I2C_RXFIFO_UDF_INT interrupt. (R/SS/WTC)

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

**Register 28.22. I2C_INT_RAW_REG (0x0020)**

**Continued from the previous page...**

**I2C_SCL_ST_TO_INT_RAW** The raw interrupt bit for the I2C_SCL_ST_TO_INT interrupt. (R/SS/WTC)

**I2C_SCL_MAIN_ST_TO_INT_RAW** The raw interrupt bit for the I2C_SCL_MAIN_ST_TO_INT interrupt. (R/SS/WTC)

**I2C_DET_START_INT_RAW** The raw interrupt bit for the I2C_DET_START_INT interrupt. (R/SS/WTC)

**I2C_SLAVE_STRETCH_INT_RAW** The raw interrupt bit for the I2C_SLAVE_STRETCH_INT interrupt. (R/SS/WTC)

**I2C_GENERAL_CALL_INT_RAW** The raw interrupt bit for the I2C_GENARAL_CALL_INT interrupt. (R/SS/WTC)

**Register 28.23. I2C_INT_CLR_REG (0x0024)**

| (reserved) | I2C_GENERAL_CALL_INT_CLR | I2C_SLAVE_STRETCH_INT_CLR | I2C_DET_START_INT_CLR | I2C_SCL_MAIN_ST_TO_INT_CLR | I2C_SCL_ST_TO_INT_CLR | I2C_RXFIFO_UDF_INT_CLR | I2C_TXFIFO_OVF_INT_CLR | I2C_NACK_INT_CLR | I2C_TRANS_START_INT_CLR | I2C_TIME_OUT_INT_CLR | I2C_TRANS_COMPLETE_INT_CLR | I2C_MST_TXFIFO_UDF_INT_CLR | I2C_ARBITRATION_LOST_INT_CLR | I2C_BYTE_TRANS_DONE_INT_CLR | I2C_END_DETECT_INT_CLR | I2C_RXFIFO_OVF_INT_CLR | I2C_TXFIFO_WM_INT_CLR | I2C_RXFIFO_WM_INT_CLR | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 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 | Reset |

**I2C_RXFIFO_WM_INT_CLR** Set this bit to clear the I2C_RXFIFO_WM_INT interrupt. (WT)

**I2C_TXFIFO_WM_INT_CLR** Set this bit to clear the I2C_TXFIFO_WM_INT interrupt. (WT)

**I2C_RXFIFO_OVF_INT_CLR** Set this bit to clear the I2C_RXFIFO_OVF_INT interrupt. (WT)

**I2C_END_DETECT_INT_CLR** Set this bit to clear the I2C_END_DETECT_INT interrupt. (WT)

**I2C_BYTE_TRANS_DONE_INT_CLR** Set this bit to clear the I2C_BYTE_TRANS_DONE_INT interrupt. (WT)

**I2C_ARBITRATION_LOST_INT_CLR** Set this bit to clear the I2C_ARBITRATION_LOST_INT interrupt. (WT)

**I2C_MST_TXFIFO_UDF_INT_CLR** Set this bit to clear the I2C_MST_TXFIFO_UDF_INT interrupt. (WT)

**I2C_TRANS_COMPLETE_INT_CLR** Set this bit to clear the I2C_TRANS_COMPLETE_INT interrupt. (WT)

**I2C_TIME_OUT_INT_CLR** Set this bit to clear the I2C_TIME_OUT_INT interrupt. (WT)

**I2C_TRANS_START_INT_CLR** Set this bit to clear the I2C_TRANS_START_INT interrupt. (WT)

**I2C_NACK_INT_CLR** Set this bit to clear the I2C_NACK_INT interrupt. (WT)

**I2C_TXFIFO_OVF_INT_CLR** Set this bit to clear the I2C_TXFIFO_OVF_INT interrupt. (WT)

**I2C_RXFIFO_UDF_INT_CLR** Set this bit to clear the I2C_RXFIFO_UDF_INT interrupt. (WT)

**I2C_SCL_ST_TO_INT_CLR** Set this bit to clear the I2C_SCL_ST_TO_INT interrupt. (WT)

**I2C_SCL_MAIN_ST_TO_INT_CLR** Set this bit to clear the I2C_SCL_MAIN_ST_TO_INT interrupt. (WT)

**I2C_DET_START_INT_CLR** Set this bit to clear the I2C_DET_START_INT interrupt. (WT)

**I2C_SLAVE_STRETCH_INT_CLR** Set this bit to clear the I2C_SLAVE_STRETCH_INT interrupt. (WT)

**I2C_GENERAL_CALL_INT_CLR** Set this bit for the I2C_GENARAL_CALL_INT interrupt. (WT)

**Register 28.24. I2C_INT_ENA_REG (0x0028)**

| 31–18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | I2C_GENERAL_CALL_INT_ENA | I2C_SLAVE_STRETCH_INT_ENA | I2C_DET_START_INT_ENA | I2C_SCL_MAIN_ST_TO_INT_ENA | I2C_SCL_ST_TO_INT_ENA | I2C_RXFIFO_UDF_INT_ENA | I2C_TXFIFO_OVF_INT_ENA | I2C_NACK_INT_ENA | I2C_TRANS_START_INT_ENA | I2C_TIME_OUT_INT_ENA | I2C_TRANS_COMPLETE_INT_ENA | I2C_MST_TXFIFO_UDF_INT_ENA | I2C_ARBITRATION_LOST_INT_ENA | I2C_BYTE_TRANS_DONE_INT_ENA | I2C_END_DETECT_INT_ENA | I2C_RXFIFO_OVF_INT_ENA | I2C_TXFIFO_WM_INT_ENA | I2C_RXFIFO_WM_INT_ENA | |
| 0 0 0 0 0 0 0 0 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 |

**I2C_RXFIFO_WM_INT_ENA** The interrupt enable bit for the I2C_RXFIFO_WM_INT interrupt. (R/W)

**I2C_TXFIFO_WM_INT_ENA** The interrupt enable bit for the I2C_TXFIFO_WM_INT interrupt. (R/W)

**I2C_RXFIFO_OVF_INT_ENA** The interrupt enable bit for the I2C_RXFIFO_OVF_INT interrupt. (R/W)

**I2C_END_DETECT_INT_ENA** The interrupt enable bit for the I2C_END_DETECT_INT interrupt. (R/W)

**I2C_BYTE_TRANS_DONE_INT_ENA** The interrupt enable bit for the I2C_BYTE_TRANS_DONE_INT interrupt. (R/W)

**I2C_ARBITRATION_LOST_INT_ENA** The interrupt enable bit for the I2C_ARBITRATION_LOST_INT interrupt. (R/W)

**I2C_MST_TXFIFO_UDF_INT_ENA** The interrupt enable bit for the I2C_MST_TXFIFO_UDF_INT interrupt. (R/W)

**I2C_TRANS_COMPLETE_INT_ENA** The interrupt enable bit for the I2C_TRANS_COMPLETE_INT interrupt. (R/W)

**I2C_TIME_OUT_INT_ENA** The interrupt enable bit for the I2C_TIME_OUT_INT interrupt. (R/W)

**I2C_TRANS_START_INT_ENA** The interrupt enable bit for the I2C_TRANS_START_INT interrupt. (R/W)

**I2C_NACK_INT_ENA** The interrupt enable bit for the I2C_NACK_INT interrupt. (R/W)

**I2C_TXFIFO_OVF_INT_ENA** The interrupt enable bit for the I2C_TXFIFO_OVF_INT interrupt. (R/W)

**I2C_RXFIFO_UDF_INT_ENA** The interrupt enable bit for the I2C_RXFIFO_UDF_INT interrupt. (R/W)

**I2C_SCL_ST_TO_INT_ENA** The interrupt enable bit for the I2C_SCL_ST_TO_INT interrupt. (R/W)

**I2C_SCL_MAIN_ST_TO_INT_ENA** The interrupt enable bit for the I2C_SCL_MAIN_ST_TO_INT interrupt. (R/W)

**I2C_DET_START_INT_ENA** The interrupt enable bit for the I2C_DET_START_INT interrupt. (R/W)

**I2C_SLAVE_STRETCH_INT_ENA** The interrupt enable bit for the I2C_SLAVE_STRETCH_INT interrupt. (R/W)

**I2C_GENERAL_CALL_INT_ENA** The interrupt enable bit for the I2C_GENARAL_CALL_INT interrupt. (R/W)

**Register 28.25. I2C_INT_STATUS_REG (0x002C)**

| (reserved) | I2C_GENERAL_CALL_INT_ST | I2C_SLAVE_STRETCH_INT_ST | I2C_DET_START_INT_ST | I2C_SCL_MAIN_ST_TO_INT_ST | I2C_SCL_ST_TO_INT_ST | I2C_RXFIFO_UDF_INT_ST | I2C_TXFIFO_OVF_INT_ST | I2C_NACK_INT_ST | I2C_TRANS_START_INT_ST | I2C_TIME_OUT_INT_ST | I2C_TRANS_COMPLETE_INT_ST | I2C_MST_TXFIFO_UDF_INT_ST | I2C_ARBITRATION_LOST_INT_ST | I2C_BYTE_TRANS_DONE_INT_ST | I2C_END_DETECT_INT_ST | I2C_RXFIFO_OVF_INT_ST | I2C_TXFIFO_WM_INT_ST | I2C_RXFIFO_WM_INT_ST | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 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 | Reset |

**I2C_RXFIFO_WM_INT_ST** The masked interrupt status bit for the I2C_RXFIFO_WM_INT interrupt. (RO)

**I2C_TXFIFO_WM_INT_ST** The masked interrupt status bit for the I2C_TXFIFO_WM_INT interrupt. (RO)

**I2C_RXFIFO_OVF_INT_ST** The masked interrupt status bit for the I2C_RXFIFO_OVF_INT interrupt. (RO)

**I2C_END_DETECT_INT_ST** The masked interrupt status bit for the I2C_END_DETECT_INT interrupt. (RO)

**I2C_BYTE_TRANS_DONE_INT_ST** The masked interrupt status bit for the I2C_BYTE_TRANS_DONE_INT interrupt. (RO)

**I2C_ARBITRATION_LOST_INT_ST** The masked interrupt status bit for the I2C_ARBITRATION_LOST_INT interrupt. (RO)

**I2C_MST_TXFIFO_UDF_INT_ST** The masked interrupt status bit for the I2C_MST_TXFIFO_UDF_INT interrupt. (RO)

**I2C_TRANS_COMPLETE_INT_ST** The masked interrupt status bit for the I2C_TRANS_COMPLETE_INT interrupt. (RO)

**I2C_TIME_OUT_INT_ST** The masked interrupt status bit for the I2C_TIME_OUT_INT interrupt. (RO)

**I2C_TRANS_START_INT_ST** The masked interrupt status bit for the I2C_TRANS_START_INT interrupt. (RO)

**I2C_NACK_INT_ST** The masked interrupt status bit for the I2C_NACK_INT interrupt. (RO)

**I2C_TXFIFO_OVF_INT_ST** The masked interrupt status bit for the I2C_TXFIFO_OVF_INT interrupt. (RO)

**I2C_RXFIFO_UDF_INT_ST** The masked interrupt status bit for the I2C_RXFIFO_UDF_INT interrupt. (RO)

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

**Register 28.25. I2C_INT_STATUS_REG (0x002C)**

**Continued from the previous page...**

**I2C_SCL_ST_TO_INT_ST** The masked interrupt status bit for the I2C_SCL_ST_TO_INT interrupt. (RO)

**I2C_SCL_MAIN_ST_TO_INT_ST** The masked interrupt status bit for the I2C_SCL_MAIN_ST_TO_INT interrupt. (RO)

**I2C_DET_START_INT_ST** The masked interrupt status bit for the I2C_DET_START_INT interrupt. (RO)

**I2C_SLAVE_STRETCH_INT_ST** The masked interrupt status bit for the I2C_SLAVE_STRETCH_INT interrupt. (RO)

**I2C_GENERAL_CALL_INT_ST** The masked interrupt status bit for the I2C_GENARAL_CALL_INT interrupt. (RO)

**Register 28.26. I2C_COMD0_REG (0x0058)**

| I2C_COMMAND0_DONE | (reserved) | I2C_COMMAND0 | |
|---|---|---|---|
| 31 | 30 … 14 | 13 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND0** This is the content of command register 0. It consists of three parts:

- op_code is the command. 1: WRITE; 2: STOP; 3: READ; 4: END; 6: RSTART.
- 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. For more information, see Section 28.4.9.

(R/W)

**I2C_COMMAND0_DONE** When command 0 has been executed in master mode, this bit changes to high level. (R/W/SS)

Register 28.27. I2C_COMD1_REG (0x005C)

| I2C_COMMAND1_DONE | (reserved) | I2C_COMMAND1 | |
|---|---|---|---|
| 31 | 30 … 14 | 13 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND1** This is the content of command register 1. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND1_DONE** When command 1 has been executed in master mode, this bit changes to high level. (R/W/SS)

Register 28.28. I2C_COMD2_REG (0x0060)

| I2C_COMMAND2_DONE | (reserved) | I2C_COMMAND2 | |
|---|---|---|---|
| 31 | 30 … 14 | 13 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND2** This is the content of command register 2. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND2_DONE** When command 2 has been executed in master mode, this bit changes to high Level. (R/W/SS)

Register 28.29. I2C_COMD3_REG (0x0064)

| I2C_COMMAND3_DONE | (reserved) | I2C_COMMAND3 | |
|---|---|---|---|
| 31 | 30 … 14 | 13 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND3** This is the content of command register 3. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND3_DONE** When command 3 has been executed in master mode, this bit changes to high level. (R/W/SS)

Register 28.30. I2C_COMD4_REG (0x0068)

| I2C_COMMAND4_DONE | (reserved) | I2C_COMMAND4 | |
|---|---|---|---|
| 31 | 30 … 14 | 13 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND4** This is the content of command register 4. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND4_DONE** When command 4 has been executed in master mode, this bit changes to high level. (R/W/SS)

Register 28.31. I2C_COMD5_REG (0x006C)

| I2C_COMMAND5_DONE | (reserved) | I2C_COMMAND5 | |
|---|---|---|---|
| 31 | 30 … 14 | 13 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND5** This is the content of command register 5. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND5_DONE** When command 5 has been executed in master mode, this bit changes to high level. (R/W/SS)

Register 28.32. I2C_COMD6_REG (0x0070)

| I2C_COMMAND6_DONE | (reserved) | I2C_COMMAND6 | |
|---|---|---|---|
| 31 | 30 … 14 | 13 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND6** This is the content of command register 6. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND6_DONE** When command 6 has been executed in master mode, this bit changes to high level. (R/W/SS)

**Register 28.33. I2C_COMD7_REG (0x0074)**

| I2C_COMMAND7_DONE | (reserved) | I2C_COMMAND7 | |
|---|---|---|---|
| 31 | 30 … 14 | 13 … 0 | |
| 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**I2C_COMMAND7** This is the content of command register 7. It is the same as that of I2C_COMMAND0. (R/W)

**I2C_COMMAND7_DONE** When command 7 has been executed in master mode, this bit changes to high level. (R/W/SS)

**Register 28.34. I2C_DATE_REG (0x00F8)**

| I2C_DATE | |
|---|---|
| 31 … 0 | |
| 0x20070201 | Reset |

**I2C_DATE** This is the version control register. (R/W)

# Chapter 29

# I2S Controller (I2S)

## 29.1 Overview

ESP32-C3 has a built-in I2S interface, which provides a flexible communication interface for streaming digital data in multimedia applications, especially digital audio applications.

The I2S standard bus defines three signals: a bit clock signal (BCK), a channel/word select signal (WS), and a serial data signal (SD). A basic I2S data bus has one master and one slave. The roles remain unchanged throughout the communication. The I2S module on ESP32-C3 provides separate transmit (TX) and receive (RX) units for high performance.

## 29.2 Terminology

To better illustrate the functionality of I2S, the following terms are used in this chapter.

**Master mode** As a master, I2S drives BCK/WS signals, and sends data to or receives data from a slave.

**Slave mode** As a slave, I2S is driven by BCK/WS signals, and receives data from or sends data to a master.

**Full-duplex** There are two separate data lines. Transmitted and received data are carried simultaneously.

**Half-duplex** Only one side, the master or the slave, sends data first, and the other side receives data. Sending data and receiving data can not happen at the same time.

**A-law and $\mu$-law** A-law and $\mu$-law are compression/decompression algorithms in digital pulse code modulated (PCM) non-uniform quantization, which can effectively improve the signal-to-quantization noise ratio.

**TDM RX mode** In this mode, pulse code modulated (PCM) data is received and stored into memory via direct memory access (DMA), utilizing time division multiplexing (TDM). The signal lines include: BCK, WS, and SD. Data from 16 channels at most can be received. TDM Philips standard, TDM MSB alignment standard, and TDM PCM standard are supported in this mode, depending on user configuration.

**Normal PDM RX mode** In this mode, pulse density modulation (PDM) data is received and stored into memory via DMA. Used signals: WS and DATA. PDM standard is supported in this mode by user configuration.

**TDM TX mode** In this mode, pulse code modulated (PCM) data is sent from memory via DMA, in a way of time division multiplexing (TDM). The signal lines include: BCK, WS, and DATA. Data up to 16 channels can be sent. TDM Philips standard, TDM MSB alignment standard, and TDM PCM standard are supported in this mode, depending on user configuration.

**Normal PDM TX mode** In this mode, pulse density modulation (PDM) data is sent from memory via DMA. The signal lines include: WS and DATA. PDM standard is supported in this mode by user configuration.

**PCM-to-PDM TX mode** In this mode, I2S as a **master**, converts the pulse code modulated (PCM) data from memory via DMA into pulse density modulation (PDM) data, and then sends the data out. Used signals: WS and DATA. PDM standard is supported in this mode by user configuration.

## 29.3 Features

- Supports master mode and slave mode
- Supports full-duplex and half-duplex communications
- Provides separate TX unit and RX unit, independent of each other
- Supports TX unit and RX unit to work independently and simultaneously
- Supports a variety of audio standards:
  - TDM Philips standard
  - TDM MSB alignment standard
  - TDM PCM standard
  - PDM standard
- Configurable high-precision sample clock
- Supports the following frequencies: 8 kHz, 16 kHz, 32 kHz, 44.1 kHz, 48 kHz, 88.2 kHz, 96 kHz, 128 kHz, and 192 kHz (192 kHz is not supported in 32-bit slave mode).
- Supports 8-/16-/24-/32-bit data communication
- Supports DMA access
- Supports standard I2S interface interrupts

## 29.4 System Architecture

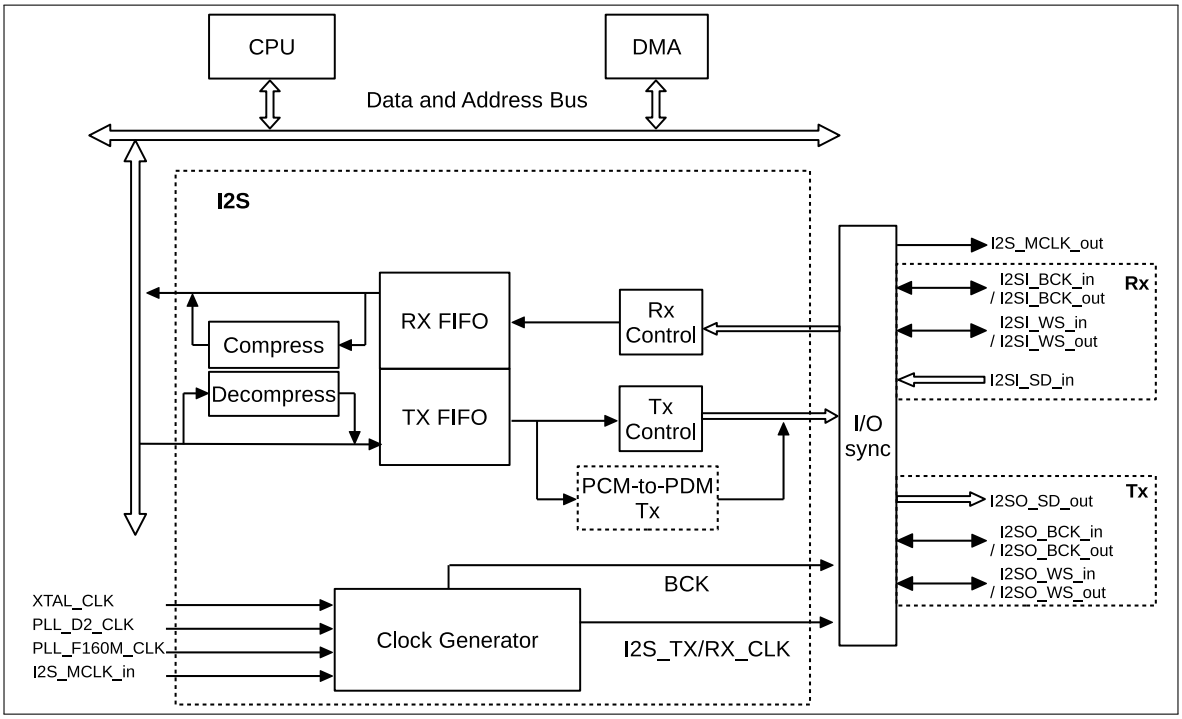



Figure 29.4-1. ESP32-C3 I2S System Diagram

Figure 29.4-1 shows the structure of ESP32-C3 I2S module, consisting of:

- TX unit (TX control)
- RX unit (RX control)
- input and output timing unit (I/O sync)
- clock divider (Clock Generator)
- 64 x 32-bit TX FIFO
- 64 x 32-bit RX FIFO
- Compress/Decompress units

I2S module supports direct access (DMA) to internal memory, see Chapter *2 GDMA Controller (GDMA)*.

Both the TX unit and the RX unit have a three-line interface that includes a bit clock line (BCK), a word select line (WS), and a serial data line (SD). The SD line of the TX unit is fixed as output, and the SD line of the RX unit as input. BCK and WS signal lines for TX unit and RX unit can be configured as master output mode or slave input mode.

The signal bus of I2S module is shown at the right part of Figure 29.4-1. The naming of these signals in RX and TX units follows the pattern: I2SA_B_C, such as I2SI_BCK_in.

- "A": direction of data bus
  - "I": input, receiving
  - "O": output, transmitting

- "B": signal function
  - BCK
  - WS
  - SD
- "C": signal direction
  - "in": input signal into I2S module
  - "out": output signal from I2S module

Table 29.4-1 provides a detailed description of I2S signals.

**Table 29.4-1. I2S Signal Description**

| Signal | Direction | Function |
|---|---|---|
| I2SI_BCK_in | Input | In I2S slave mode, inputs BCK signal for RX unit. |
| I2SI_BCK_out | Output | In I2S master mode, outputs BCK signal for RX unit. |
| I2SI_WS_in | Input | In I2S slave mode, inputs WS signal for RX unit. |
| I2SI_WS_out | Output | In I2S master mode, outputs WS signal for RX unit. |
| I2SI_Data_in | Input | Works as the serial input data bus for I2S RX unit. |
| I2SO_Data_out | Output | Works as the serial output data bus for I2S TX unit. |
| I2SO_BCK_in | Input | In I2S slave mode, inputs BCK signal for TX unit. |
| I2SO_BCK_out | Output | In I2S master mode, outputs BCK signal for TX unit. |
| I2SO_WS_in | Input | In I2S slave mode, inputs WS signal for TX unit. |
| I2SO_WS_out | Output | In I2S master mode, outputs WS signal for TX unit. |
| I2S_MCLK_in | Input | In I2S slave mode, works as a clock source from the external master. |
| I2S_MCLK_out | Output | In I2S master mode, works as a clock source for the external slave. |

**Note:**
Any required signals of I2S must be mapped to the chip's pins via GPIO matrix, see Chapter 5 *IO MUX and GPIO Matrix (GPIO, IO MUX)*.

## 29.5 Supported Audio Standards

ESP32-C3 I2S supports multiple audio standards, including TDM Philips standard, TDM MSB alignment standard, TDM PCM standard, and PDM standard.

Select the needed standard by configuring the following bits:

- I2S_TX/RX_TDM_EN
  - 0: disable TDM mode.
  - 1: enable TDM mode.
- I2S_TX/RX_PDM_EN

  - 0: disable PDM mode.
  - 1: enable PDM mode.
- I2S_TX/RX_MSB_SHIFT
  - 0: WS and SD signals change simultaneously, i.e. enable MSB alignment standard.
  - 1: WS signal changes one BCK clock cycle earlier than SD signal, i.e. enable Philips standard or select PCM standard.
- I2S_TX/RX_PCM_BYPASS
  - 0: enable PCM standard.
  - 1: disable PCM standard.

### 29.5.1 TDM Philips Standard

Philips specifications require that WS signal changes one BCK clock cycle earlier than SD signal on BCK falling edge, which means that WS signal is valid from one clock cycle before transmitting the first bit of channel data and changes one clock before the end of channel data transfer. SD signal line transmits the most significant bit of audio data first.

Compared with Philips standard, TDM Philips standard supports multiple channels, see Figure 29.5-1.

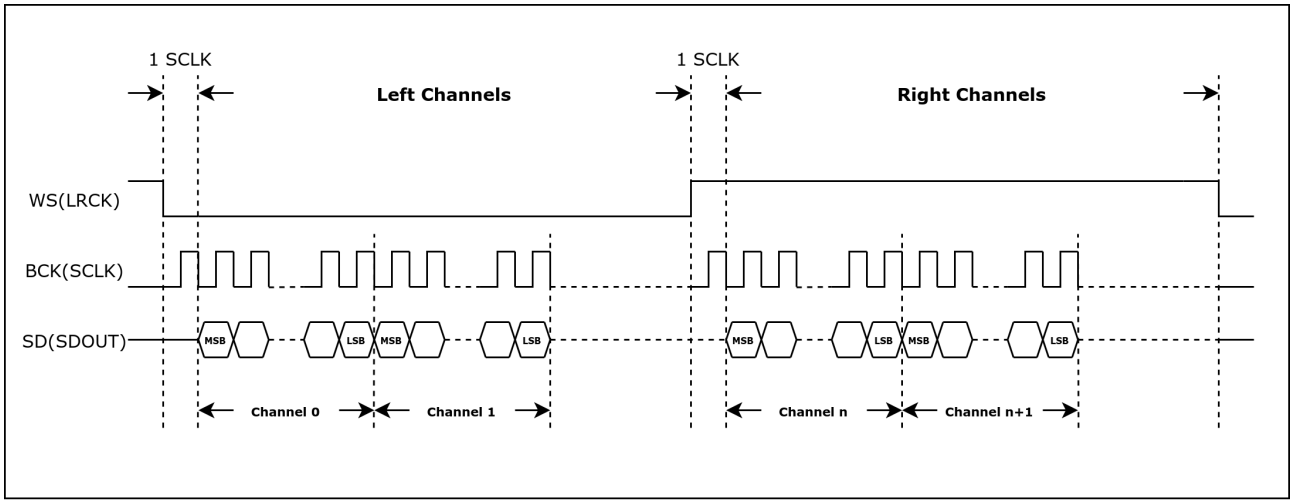



Figure 29.5-1. TDM Philips Standard Timing Diagram

### 29.5.2 TDM MSB Alignment Standard

MSB alignment specifications require WS and SD signals change simultaneously on the falling edge of BCK. The WS signal is valid until the end of channel data transfer. The SD signal line transmits the most significant bit of audio data first.

Compared with MSB alignment standard, TDM MSB alignment standard supports multiple channels, see Figure 29.5-2.

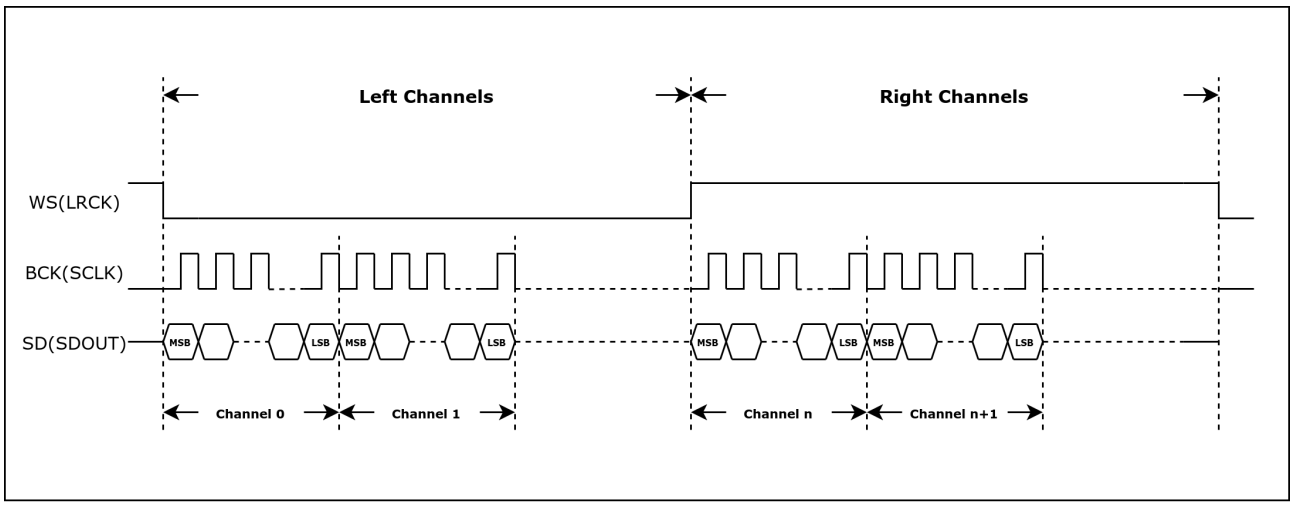



Figure 29.5-2. TDM MSB Alignment Standard Timing Diagram

## 29.5.3 TDM PCM Standard

Short frame synchronization under PCM standard requires WS signal changes one BCK clock cycle earlier than SD signal on the falling edge of BCK, which means that the WS signal becomes valid one clock cycle before transferring the first bit of channel data and remains unchanged in this BCK clock cycle. SD signal line transmits the most significant bit of audio data first.

Compared with PCM standard, TDM PCM standard supports multiple channels, see Figure 29.5-3.

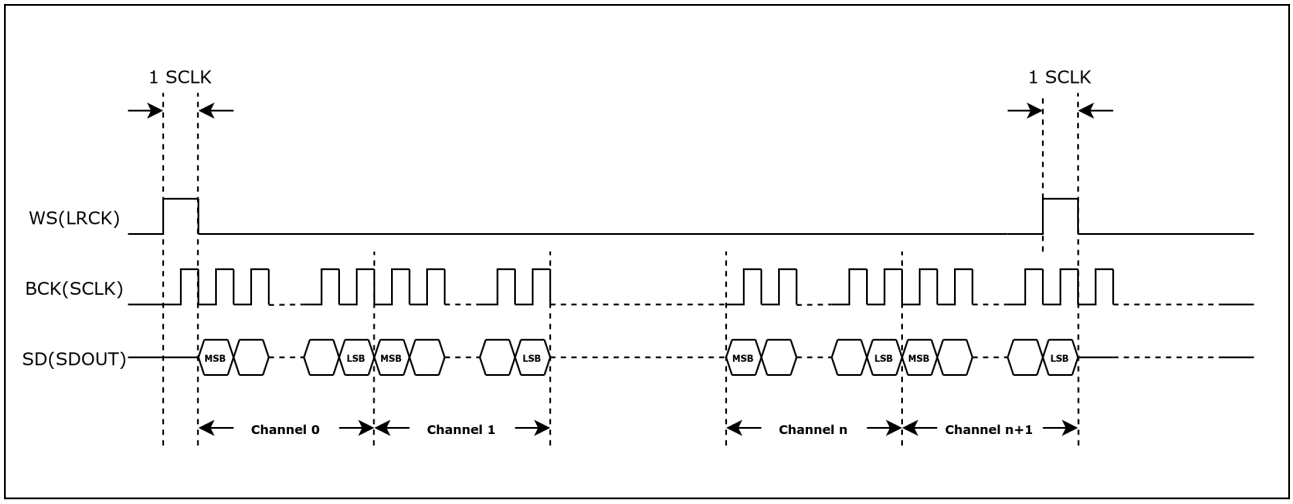



Figure 29.5-3. TDM PCM Standard Timing Diagram

## 29.5.4 PDM Standard

Under PDM standard, WS signal changes continuously during data transmission. The low-level and high-level of this signal indicates the left channel and right channel, respectively. WS and SD signals change simultaneously on the falling edge of BCK, see Figure 29.5-4.

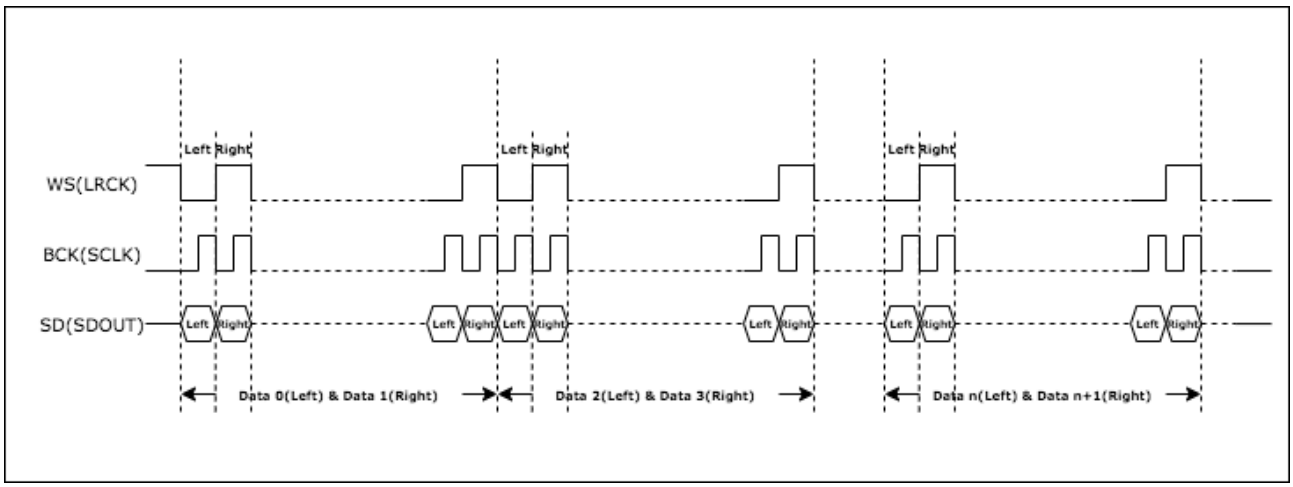



Figure 29.5-4. PDM Standard Timing Diagram

## 29.6 I2S TX/RX Clock

I2S_TX/RX_CLK is the master clock of I2S TX/RX unit, divided from:

- 40 MHz XTAL_CLK
- 160 MHz PLL_F160M_CLK
- 240 MHz PLL_D2_CLK
- or external input clock: I2S_MCLK_in

The serial clock (BCK) of the I2S TX/RX unit is divided from I2S_TX/RX_CLK, as shown in Figure 29.6-1. I2S_TX/RX_CLK_SEL is used to select clock source for TX/RX unit, and I2S_TX/RX_CLK_ACTIVE to enable or disable the clock source.

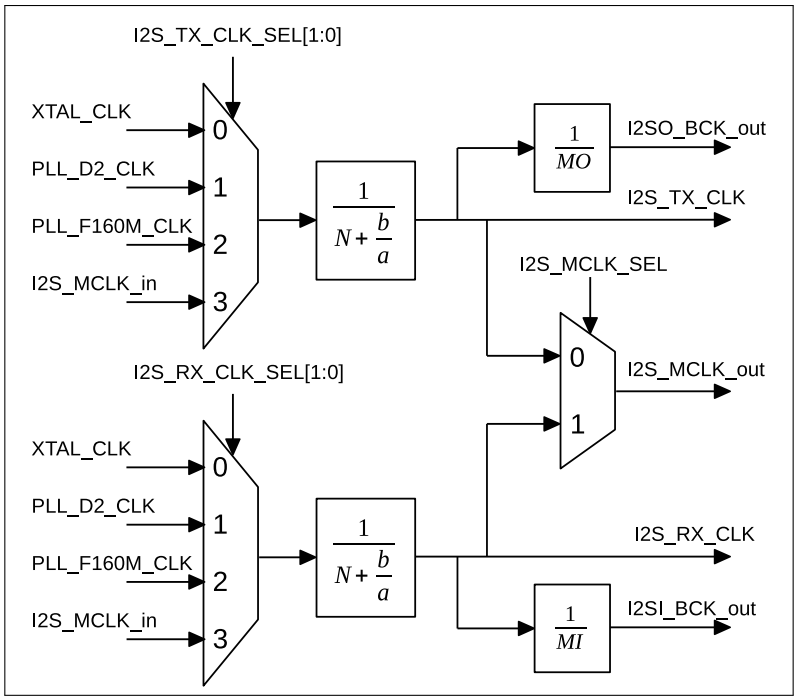



Figure 29.6-1. I2S Clock

The following formula shows the relation between I2S_TX/RX_CLK frequency $f_{\text{I2S_TX/RX_CLK}}$ and the divider clock source frequency $f_{\text{I2S_CLK_S}}$:

$$f_{\text{I2S_TX/RX_CLK}} = \frac{f_{\text{I2S_CLK_S}}}{\text{N} + \frac{\text{b}}{\text{a}}}$$

N is an integer value between 2 and 256. The value of N corresponds to the value of I2S_TX/RX_CLKM_DIV_NUM in register I2S_TX/RX_CLKM_CONF_REG as follows:

- When I2S_TX/RX_CLKM_DIV_NUM = 0, N = 256.
- When I2S_TX/RX_CLKM_DIV_NUM = 1, N = 2.
- When I2S_TX/RX_CLKM_DIV_NUM has any other value, N = I2S_TX/RX_CLKM_DIV_NUM.

The values of "a" and "b" in fractional divider depend only on x, y, z, and yn1. The corresponding formulas are as follows:

- When b $<= \frac{a}{2}$ , yn1 $= 0$, x $= floor([\frac{a}{b}]) - 1$, y $=$ a%b, z $=$ b;
- When b $> \frac{a}{2}$ , yn1 $= 1$, x $= floor([\frac{a}{a - b}]) - 1$, y $=$ a%(a - b), z $=$ a - b.

The values of x, y, z, and yn1 are configured in I2S_TX/RX_CLKM_DIV_X, I2S_TX/RX_CLKM_DIV_Y, I2S_TX/RX_CLKM_DIV_Z, and I2S_TX/RXCLKM_DIV_YN1. To configure the integer divider, clear I2S_TX/RX_CLKM_DIV_X and I2S_TX/RX_CLKM_DIV_Z, then set I2S_TX/RX_CLKM_DIV_Y to 1.

> **Note:**
> Using fractional divider may introduce some clock jitter.

In master TX mode, the serial clock BCK for I2S TX unit is I2SO_BCK_out, divided from I2S_TX_CLK. That is:

$$f_{\text{I2SO_BCK_out}} = \frac{f_{\text{I2S_TX_CLK}}}{\text{MO}}$$

"MO" is an integer value:

$$\text{MO} = \text{I2S_TX_BCK_DIV_NUM} + 1$$

> **Note:**
> I2S_TX_BCK_DIV_NUM must not be configured as 1.

In master RX mode, the serial clock BCK for I2S RX unit is I2SI_BCK_out, divided from I2S_RX_CLK. That is:

$$f_{\text{I2SI_BCK_out}} = \frac{f_{\text{I2S_RX_CLK}}}{\text{MI}}$$

"MI" is an integer value:

$$\text{MI} = \text{I2S_RX_BCK_DIV_NUM} + 1$$

**Note:**

- I2S_RX_BCK_DIV_NUM must not be configured as 1.
- In I2S slave mode, make sure $f_{I2S_TX/RX_CLK}$ >= 8 * $f_{BCK}$. I2S module can output I2S_MCLK_out as the master clock for peripherals.

## 29.7 I2S Reset

The units and FIFOs in I2S module are reset by the following bits.

- I2S TX/RX units: reset by the bits I2S_TX_RESET and I2S_RX_RESET.
- I2S TX/RX FIFO: reset by the bits I2S_TX_FIFO_RESET and I2S_RX_FIFO_RESET.

**Note:**
I2S module clock must be configured first before the module and FIFO are reset.

## 29.8 I2S Master/Slave Mode

The ESP32-C3 I2S module can operate as a master or a slave in half-duplex and full-duplex communication modes, depending on the configuration of I2S_RX_SLAVE_MOD and I2S_TX_SLAVE_MOD.

- I2S_TX_SLAVE_MOD
  - 0: master TX mode
  - 1: slave TX mode
- I2S_RX_SLAVE_MOD
  - 0: master RX mode
  - 1: slave RX mode

### 29.8.1 Master/Slave TX Mode

- I2S works as a master transmitter:
  - Set the bit I2S_TX_START to start transmitting data.
  - TX unit keeps driving the clock signal and serial data.
  - If I2S_TX_STOP_EN is set and all the data in FIFO is transmitted, the master stops transmitting data.
  - If I2S_TX_STOP_EN is cleared and all the data in FIFO is transmitted, meanwhile no new data is filled into FIFO, then the TX unit keeps sending the last data frame.
  - Master stops sending data when the bit I2S_TX_START is cleared.
- I2S works as a slave transmitter:
  - Set the bit I2S_TX_START.

- Wait for the master BCK clock to enable a transmit operation.
- If I2S_TX_STOP_EN is set and all the data in FIFO is transmitted, then the slave keeps sending zeros, till the master stops providing BCK signal.
- If I2S_TX_STOP_EN is cleared and all the data in FIFO is transmitted, meanwhile no new data is filled into FIFO, then the TX unit keeps sending the last data frame.
- If I2S_TX_START is cleared, slave keeps sending zeros till the master stops providing BCK clock signal.

### 29.8.2 Master/Slave RX Mode

- I2S works as a master receiver:
  - Set the bit I2S_RX_START to start receiving data.
  - RX unit keeps outputting clock signal and sampling input data.
  - RX unit stops receiving data when the bit I2S_RX_START is cleared.
- I2S works as a slave receiver:
  - Set the bit I2S_RX_START.
  - Wait for master BCK signal to start receiving data.
  - RX unit stops receiving data when the bit I2S_RX_START is cleared.

## 29.9 Transmitting Data

**Note:**
Updating the configuration described in this and subsequent sections requires to set I2S_TX_UPDATE accordingly, to synchronize registers from APB clock domain to TX clock domain. For more detailed configuration, see Section 29.11.1.

In TX mode, I2S first reads data from DMA and sends these data out via output signals according to the configured data mode and channel mode.

### 29.9.1 Data Format Control

Data format is controlled in the following phases:

- Phase I: read data from memory and write it to TX FIFO.
- Phase II: read the data to send (TX data) from TX FIFO and convert the data according to output data mode.
- Phase III: clock out the TX data serially.

#### 29.9.1.1 Bit Width Control of Channel Valid Data

The bit width of valid data in each channel is determined by I2S_TX_BITS_MOD and I2S_TX_24_FILL_EN, see the table below.

Table 29.9-1. Bit Width of Channel Valid Data

| Channel Valid Data Width | I2S_TX_BITS_MOD | I2S_TX_24_FILL_EN |
|---|---|---|
| 32 | 31 | x[1] |
| | 23 | 1 |
| 24 | 23 | 0 |
| 16 | 15 | x |
| 8 | 7 | x |

[1] This value is ignored.

#### 29.9.1.2 Endian Control of Channel Valid Data

When I2S reads data from DMA, the data endian under various data width is controlled by I2S_TX_BIG_ENDIAN, see the table below.

Table 29.9-2. Endian of Channel Valid Data

| Channel Valid Data Width | Origin Data | Endian of Processed Data | I2S_TX_BIG_ENDIAN |
|---|---|---|---|
| 32 | {B3, B2, B1, B0} | {B3, B2, B1, B0} | 0 |
| | | {B0, B1, B2, B3} | 1 |
| 24 | {B2, B1, B0} | {B2, B1, B0} | 0 |
| | | {B0, B1, B2} | 1 |
| 16 | {B1, B0} | {B1, B0} | 0 |
| | | {B0, B1} | 1 |
| 8 | {B0} | {B0} | x |

#### 29.9.1.3 A-law/$\mu$-law Compression and Decompression

ESP32-C3 I2S compresses/decompresses the valid data into 32-bit by A-law or by $\mu$-law. If the bit width of valid data is smaller than 32, zeros are filled to the extra high bits of the data to be compressed/decompressed by default.

> **Note:**
> Extra high bits here mean the bits[31: channel valid data width] of the data to be compressed/decompressed.

Configure I2S_TX_PCM_BYPASS to:

- 0: compress or decompress the data.
- 1: do not compress or decompress the data.

Configure I2S_TX_PCM_CONF to:

- 0: decompress the data using A-law.
- 1: compress the data using A-law.
- 2: decompress the data using $\mu$-law.
- 3: compress the data using $\mu$-law.

At this point, the first phase of data format control is complete.

#### 29.9.1.4 Bit Width Control of Channel TX Data

The TX data width in each channel is determined by I2S_TX_TDM_CHAN_BITS.

- If TX data width in each channel is larger than the valid data width, zeros will be filled to these extra bits. Configure I2S_TX_LEFT_ALIGN to:
  - 0: the valid data is at the lower bits of TX data. Zeros are filled into higher bits of TX data.
  - 1: the valid data is at the higher bits of TX data. Zeros are filled into lower bits of TX data.
- If the TX data width in each channel is smaller than the valid data width, only the lower bits of valid data are sent out, and the higher bits are discarded.

At this point, the second phase of data format control is complete.

#### 29.9.1.5 Bit Order Control of Channel Data

The data bit order in each channel is controlled by I2S_TX_BIT_ORDER:

- 0: Not reverse the valid data bit order;
- 1: Reverse the valid data bit order.

At this point, the data format control is complete. Figure 29.9-1 shows a complete process of TX data format control.

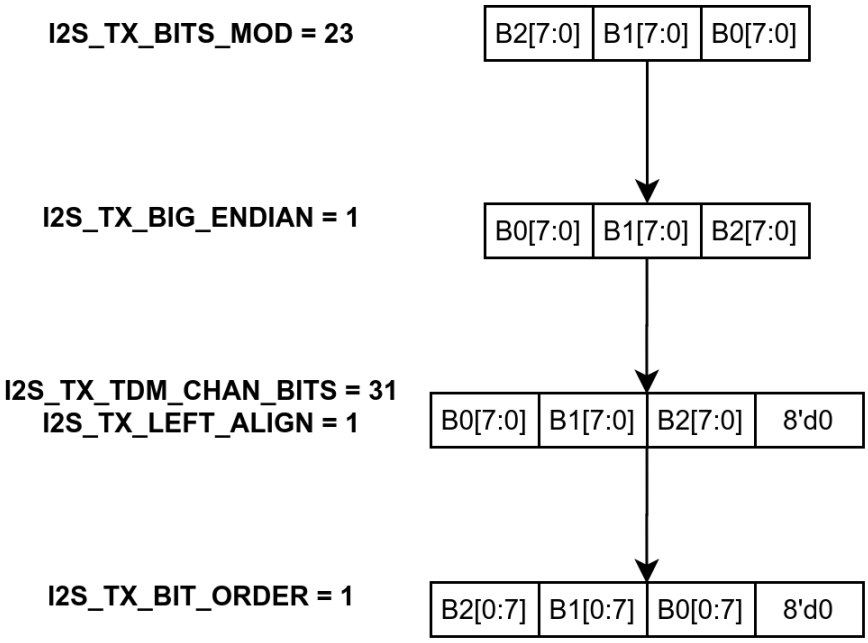



Figure 29.9-1. TX Data Format Control

## 29.9.2 Channel Mode Control

ESP32-C3 I2S supports both TDM TX mode and PDM TX mode. Set I2S_TX_TDM_EN to enable TDM TX mode, or set I2S_TX_PDM_EN to enable PDM TX mode.

**Note:**

- I2S_TX_TDM_EN and I2S_TX_PDM_EN must not be cleared or set simultaneously.
- Most stereo I2S codecs can be controlled by setting the I2S module into 2-channel mode under TDM standard.

### 29.9.2.1 I2S Channel Control in TDM TX Mode

In TDM TX mode, I2S supports up to 16 channels to output data. The total number of TX channels in use is controlled by I2S_TX_TDM_TOT_CHAN_NUM. For example, if I2S_TX_TDM_TOT_CHAN_NUM is set to 5, six channels in total (channel 0 ~ 5) will be used to transmit data, see Figure 29.9-2.

In these TX channels, if I2S_TX_TDM_CHANn_EN is set to:

- 1: this channel sends the channel data out.
- 0: the TX data to be sent by this channel is controlled by I2S_TX_CHAN_EQUAL:
  - 1: the data of previous channel is sent out.
  - 0: the data stored in I2S_SINGLE_DATA is sent out.

In TDM TX master mode, WS signal is controlled by I2S_TX_WS_IDLE_POL and I2S_TX_TDM_WS_WIDTH:

- I2S_TX_WS_IDLE_POL: the default level of WS signal
- I2S_TX_TDM_WS_WIDTH: the cycles the WS default level lasts for when transmitting all channel data

I2S_TX_HALF_SAMPLE_BITS x 2 is equal to the BCK cycles in one WS period.

**TDM Channel Configuration Example**

In this example, register configuration is as follows:

- I2S_TX_TDM_CHAN_NUM = 5, i.e. channel 0 ~ 5 are used to transmit data.
- I2S_TX_CHAN_EQUAL = 1, i.e. that data of previous channel will be transmitted if the bit I2S_TX_TDM_CHAN*n* _EN is cleared. *n* = 0 ~ 5.
- I2S_TX_TDM_CHAN0/2/5_EN = 1, i.e. these channels send their channel data out.
- I2S_TX_TDM_CHAN1/3/4_EN = 0, i.e. these channels send the previous channel data out.

Once the configuration is done, data is transmitted as follows.

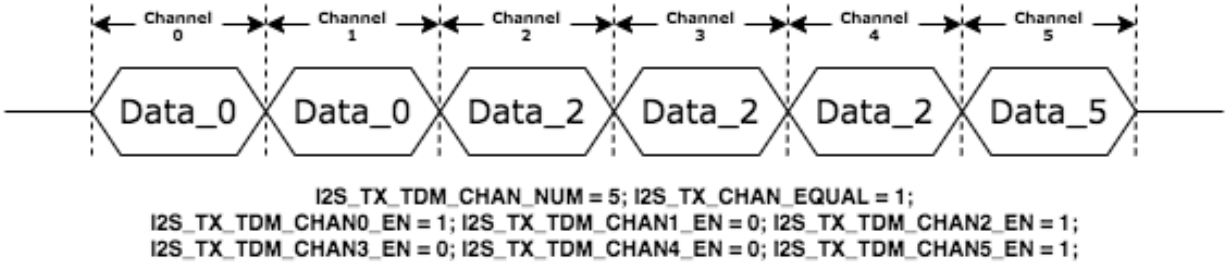



Figure 29.9-2. TDM Channel Control

#### 29.9.2.2 I2S Channel Control in PDM TX Mode

ESP32-C3 I2S supports two PDM TX modes, namely, normal PDM TX mode and PCM-to-PDM TX mode.

In PDM TX mode, fetching data from DMA is controlled by I2S_TX_MONO and I2S_TX_MONO_FST_VLD, see Table 29.9-3. Please configure the two bits according to the data stored in memory, be it the single-channel or dual-channel data.

Table 29.9-3. Data-Fetching Control in PDM TX Mode

| Data-Fetching Control Option | Mode | I2S_TX_MONO | I2S_TX_MONO_FST_VLD |
|---|---|---|---|
| Post data-fetching request to DMA at any edge of WS signal | Stereo mode | 0 | x |
| Post data-fetching request to DMA only at the second half period of WS signal | Mono mode | 1 | 0 |
| Post data-fetching request to DMA only at the first half period of WS signal | Mono mode | 1 | 1 |

In **normal PDM TX mode**, I2S channel mode is controlled by I2S_TX_CHAN_MOD and I2S_TX_WS_IDLE_POL, see the table below.

Table 29.9-4. I2S Channel Control in Normal PDM TX Mode

| Channel Control Option | Left Channel | Right Channel | Mode Control Field[1] | Channel Select Bit[2] |
|---|---|---|---|---|
| Stereo mode | Transmit the left channel data | Transmit the right channel data | 0 | x |
| Mono mode | Transmit the left channel data | Transmit the left channel data | 1 | 0 |
| | Transmit the right channel data | Transmit the right channel data | 1 | 1 |
| | Transmit the right channel data | Transmit the right channel data | 2 | 0 |
| | Transmit the left channel data | Transmit the left channel data | 2 | 1 |
| | Transmit the value of "single"[3] | Transmit the right channel data | 3 | 0 |
| | Transmit the left channel data | Transmit the value of "single" | 3 | 1 |
| | Transmit the left channel data | Transmit the value of "single" | 4 | 0 |
| | Transmit the value of "single" | Transmit the right channel data | 4 | 1 |

[1] I2S_TX_CHAN_MOD
[2] I2S_TX_WS_IDLE_POL
[3] The "single" value is equal to the value of I2S_SINGLE_DATA.

In PDM TX aster mode, the WS level of I2S module is controlled by I2S_TX_WS_IDLE_POL. The frequency of WS signal is half of BCK frequency. The configuration of WS signal is similar to that of BCK signal, see Section 29.6 and Figure 29.9-3.

In **PCM-to-PDM TX mode**, the PCM data from DMA is converted to PDM data and then output in PDM signal format. Configure I2S_PCM2PDM_CONV_EN to enable this mode.

The register configuration for PCM-to-PDM TX mode is as follows:

- Configure 1-line PDM output format or 1-/2-line DAC output mode as the table below:

Table 29.9-5. PCM-to-PDM TX Mode

| Channel Output Format | I2S_TX_PDM_DAC_MODE_EN | I2S_TX_PDM_DAC_2OUT_EN |
|---|---|---|
| 1-line PDM output format[1] | 0 | x |
| 1-line DAC output format[2] | 1 | 0 |
| 2-line DAC output format | 1 | 1 |

**Note:**

1. In PDM output format, SD data of two channels is sent out in one WS period.
2. In DAC output format, SD data of one channel is sent out in one WS period.

- Configure sampling frequency and upsampling rate
  In PCM-to-PDM TX mode, PDM clock frequency is equal to BCK frequency. The relation of sampling frequency ($f_{Sampling}$) and BCK frequency is as follows:

$$f_{Sampling} = \frac{f_{BCK}}{OSR}$$

Upsampling rate (OSR) is related to I2S_TX_PDM_SINC_OSR2 as follows:

$$\text{OSR} = \text{I2S_TX_PDM_SINC_OSR2} \times 64$$

Sampling frequency $f_{\text{Sampling}}$ is related to I2S_TX_PDM_FS as follows:

$$f_{\text{Sampling}} = \text{I2S_TX_PDM_FS} \times 100$$

Configure the registers according to needed sampling frequency, upsampling rate, and PDM clock frequency.

**PDM Channel Configuration Example**

In this example, the register configuration is as follows.

- I2S_TX_CHAN_MOD = 2, i.e. mono mode is selected.
- I2S_TX_WS_IDLE_POL = 1, i.e. both the left channel and right channel transmit the left channel data.

Once the configuration is done, the channel data is transmitted as follows.

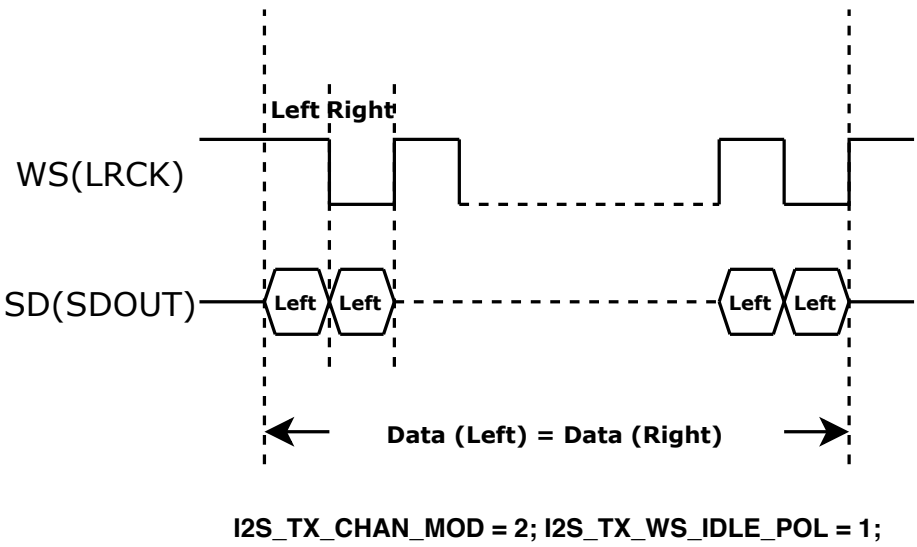



Figure 29.9-3. PDM Channel Control Example

## 29.10 Receiving Data

> **Note:**
> Updating the configuration described in this and subsequent sections requires setting I2S_RX_UPDATE, to synchronize registers from APB clock domain to RX clock domain. For more detailed configuration, see Section 29.11.2.

In RX mode, I2S first reads data from peripheral interface, and then stores the data into memory via DMA, according to the configured channel mode and data mode.

### 29.10.1 Channel Mode Control

ESP32-C3 I2S supports both TDM RX mode and PDM RX mode. Set I2S_RX_TDM_EN to enable TDM RX mode, or set I2S_RX_PDM_EN to enable PDM RX mode.

**Note:**
I2S_RX_TDM_EN and I2S_RX_PDM_EN must not be cleared or set simultaneously.

#### 29.10.1.1 I2S Channel Control in TDM RX Mode

In TDM RX mode, I2S supports up to 16 channels to input data. The total number of RX channels in use is controlled by I2S_RX_TDM_TOT_CHAN_NUM. For example, if I2S_RX_TDM_TOT_CHAN_NUM is set to 5, channel 0 ~ 5 will be used to receive data.

In these RX channels, if I2S_RX_TDM_CHANn_EN is set to:

- 1: this channel data is valid and will be stored into RX FIFO.
- 0: this channel data is invalid and will not be stored into RX FIFO.

In TDM RX master mode, WS signal is controlled by I2S_RX_WS_IDLE_POL and I2S_RX_TDM_WS_WIDTH.

- I2S_RX_WS_IDLE_POL: the default level of WS signal
- I2S_RX_TDM_WS_WIDTH: the cycles the WS default level lasts for when receiving all channel data

I2S_RX_HALF_SAMPLE_BITS x 2 is equal to the BCK cycles in one WS period.

#### 29.10.1.2 I2S Channel Control in PDM RX Mode

In PDM RX mode, I2S converts the serial data from channels to the data to be entered into memory.

In PDM RX master mode, the default level of WS signal is controlled by I2S_RX_WS_IDLE_POL. WS frequency is half of BCK frequency. The configuration of BCK signal is similar to that of WS signal as described in Section 29.6. Note, in PDM RX mode, the value of I2S_RX_HALF_SAMPLE_BITS must be same as that of I2S_RX_BITS_MOD.

### 29.10.2 Data Format Control

Data format is controlled in the following phases:

- Phase I: serial input data is converted into the data to be saved to RX FIFO.
- Phase II: the data is read from RX FIFO and converted according to input data mode.

#### 29.10.2.1 Bit Order Control of Channel Data

The channel data will be stored as the data to be input in order from high to low. The data bit order in each channel is controlled by I2S_RX_BIT_ORDER:

- 0: The bit order of the data to be input is not reversed;
- 1: The bit order of the data to be input is reversed.

At this point, the first phase of data format control is complete.

#### 29.10.2.2 Bit Width Control of Channel Storage (Valid) Data

The storage data width in each channel is controlled by I2S_RX_BITS_MOD and I2S_RX_24_FILL_EN, see the table below.

Table 29.10-1. Channel Storage Data Width

| Channel Storage Data Width | I2S_RX_BITS_MOD | I2S_RX_24_FILL_EN |
|---|---|---|
| 32 | 31 | x |
| | 23 | 1 |
| 24 | 23 | 0 |
| 16 | 15 | x |
| 8 | 7 | x |

#### 29.10.2.3 Bit Width Control of Channel RX Data

The RX data width in each channel is determined by I2S_RX_TDM_CHAN_BITS.

- If the storage data width in each channel is smaller than the received (RX) data width, then only the bits within the storage data width is saved into memory. Configure I2S_RX_LEFT_ALIGN to:
  - 0: only the lower bits of the received data within the storage data width is stored to memory.
  - 1: only the higher bits of the received data within the storage data width is stored to memory.
- If the received data width is smaller than the storage data width in each channel, the higher bits of the received data will be filled with zeros and then the data is saved to memory.

#### 29.10.2.4 Endian Control of Channel Storage Data

The received data is then converted into storage data (to be stored to memory) after some processing, such as discarding extra bits or filling zeros in missing bits. The endian of the storage data is controlled by I2S_RX_BIG_ENDIAN under various data width, see the table below.

Table 29.10-2. Channel Storage Data Endian

| Channel Storage Data Width | Origin Data | Endian of Processed Data | I2S_RX_BIG_ENDIAN |
|---|---|---|---|
| 32 | {B3, B2, B1, B0} | {B3, B2, B1, B0} | 0 |
| | | {B0, B1, B2, B3} | 1 |
| 24 | {B2, B1, B0} | {B2, B1, B0} | 0 |
| | | {B0, B1, B2} | 1 |
| 16 | {B1, B0} | {B1, B0} | 0 |
| | | {B0, B1} | 1 |
| 8 | {B0} | {B0} | x |

#### 29.10.2.5 A-law/$\mu$-law Compression and Decompression

ESP32-C3 I2S compresses/decompresses the storage data in 32-bit by A-law or by $\mu$-law. By default, zeros are filled into high bits.

Configure I2S_RX_PCM_BYPASS to:

- 0: compress or decompress the data.
- 1: do not compress or decompress the data.

Configure I2S_RX_PCM_CONF to:

- 0: decompress the data using A-law.
- 1: compress the data using A-law.
- 2: decompress the data using $\mu$-law.
- 3: compress the data using $\mu$-law.

At this point, the data format control is complete. Data then is stored into memory via DMA.

## 29.11 Software Configuration Process

### 29.11.1 Configure I2S as TX Mode

Follow the steps below to configure I2S as TX mode via software:

1. Configure the clock as described in Section 29.6.
2. Configure signal pins according to Table 29.4-1.
3. Select the mode needed by configuring the bit I2S_TX_SLAVE_MOD.
   - 0: master TX mode
   - 1: slave TX mode
4. Set needed TX data mode and TX channel mode as described in Section 29.9, and then set the bit I2S_TX_UPDATE.
5. Reset TX unit and TX FIFO as described in Section 29.7.
6. Enable corresponding interrupts, see Section 29.12.
7. Configure DMA outlink.
8. Set I2S_TX_STOP_EN if needed. For more information, please refer to Section 29.8.1.
9. Start transmitting data:
   - In master mode, wait till I2S slave gets ready, then set I2S_TX_START to start transmitting data.
   - In slave mode, set the bit I2S_TX_START. When the I2S master supplies BCK and WS signals, I2S slave starts transmitting data.
10. Wait for the interrupt signals set in Step 6, or check whether the transfer is completed by querying I2S_TX_IDLE:
    - 0: transmitter is working.

- 1: transmitter is in idle.

11. Clear I2S_TX_START to stop data transfer.

### 29.11.2 Configure I2S as RX Mode

Follow the steps below to configure I2S as RX mode via software:

1. Configure the clock as described in Section 29.6.
2. Configure signal pins according to Table 29.4-1.
3. Select the mode needed by configuring the bit I2S_RX_SLAVE_MOD.
   - 0: master RX mode
   - 1: slave RX mode
4. Set needed RX data mode and RX channel mode as described in Section 29.10, and then set the bit I2S_RX_UPDATE.
5. Reset RX unit and its FIFO according to Section 29.7.
6. Enable corresponding interrupts, see Section 29.12.
7. Configure DMA inlink, and set the length of RX data in I2S_RXEOF_NUM_REG.
8. Start receiving data:
   - In master mode, when the slave is ready, set I2S_RX_START to start receiving data.
   - In slave mode, set I2S_RX_START to start receiving data when get BCK and WS signals from the master.
9. The received data is then stored to the specified address of ESP32-C3 memory according the configuration of DMA. Then the corresponding interrupt set in Step 6 is generated.

## 29.12 I2S Interrupts

- I2S_TX_HUNG_INT: triggered when transmitting data is timed out. For example, if module is configured as TX slave mode, but the master does not provide BCK or WS signal for a long time (specified in I2S_LC_HUNG_CO NF_REG), then this interrupt will be triggered.
- I2S_RX_HUNG_INT: triggered when receiving data is timed out. For example, if I2S module is configured as RX slave mode, but the master does not send data for a long time (specified in I2S_LC_HUNG_CONF_REG), then this interrupt will be triggered.
- I2S_TX_DONE_INT: triggered when transmitting data is completed.
- I2S_RX_DONE_INT: triggered when receiving data is completed.

## 29.13 Register Summary

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| Interrupt registers | | | |
| I2S_INT_RAW_REG | I2S interrupt raw register | 0x000C | RO/WTC/SS |
| I2S_INT_ST_REG | I2S interrupt status register | 0x0010 | RO |
| I2S_INT_ENA_REG | I2S interrupt enable register | 0x0014 | R/W |
| I2S_INT_CLR_REG | I2S interrupt clear register | 0x0018 | WT |
| RX control and configuration registers | | | |
| I2S_RX_CONF_REG | I2S RX configuration register | 0x0020 | varies |
| I2S_RX_CONF1_REG | I2S RX configuration register 1 | 0x0028 | R/W |
| I2S_RX_CLKM_CONF_REG | I2S RX clock configuration register | 0x0030 | R/W |
| I2S_TX_PCM2PDM_CONF_REG | I2S TX PCM-to-PDM configuration register | 0x0040 | R/W |
| I2S_TX_PCM2PDM_CONF1_REG | I2S TX PCM-to-PDM configuration register 1 | 0x0044 | R/W |
| I2S_RX_TDM_CTRL_REG | I2S TX TDM mode control register | 0x0050 | R/W |
| I2S_RXEOF_NUM_REG | I2S RX data number control register | 0x0064 | R/W |
| TX control and configuration registers | | | |
| I2S_TX_CONF_REG | I2S TX configuration register | 0x0024 | varies |
| I2S_TX_CONF1_REG | I2S TX configuration register 1 | 0x002C | R/W |
| I2S_TX_CLKM_CONF_REG | I2S TX clock configuration register | 0x0034 | R/W |
| I2S_TX_TDM_CTRL_REG | I2S TX TDM mode control register | 0x0054 | R/W |
| RX clock and timing registers | | | |
| I2S_RX_CLKM_DIV_CONF_REG | I2S RX unit clock divider configuration register | 0x0038 | R/W |
| I2S_RX_TIMING_REG | I2S RX timing control register | 0x0058 | R/W |
| TX clock and timing registers | | | |
| I2S_TX_CLKM_DIV_CONF_REG | I2S TX unit clock divider configuration register | 0x003C | R/W |
| I2S_TX_TIMING_REG | I2S TX timing control register | 0x005C | R/W |
| Control and configuration registers | | | |
| I2S_LC_HUNG_CONF_REG | I2S timeout configuration register | 0x0060 | R/W |
| I2S_CONF_SIGLE_DATA_REG | I2S single data register | 0x0068 | R/W |
| TX status register | | | |
| I2S_STATE_REG | I2S TX status register | 0x006C | RO |
| Version register | | | |
| I2S_DATE_REG | Version control register | 0x0080 | R/W |

## 29.14 Registers

**Register 29.1. I2S_INT_RAW_REG (0x000C)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | I2S_TX_HUNG_INT_RAW | I2S_RX_HUNG_INT_RAW | I2S_TX_DONE_INT_RAW | I2S_RX_DONE_INT_RAW | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**I2S_RX_DONE_INT_RAW** The raw interrupt status bit for I2S_RX_DONE_INT interrupt. (RO/WTC/SS)

**I2S_TX_DONE_INT_RAW** The raw interrupt status bit for I2S_TX_DONE_INT interrupt. (RO/WTC/SS)

**I2S_RX_HUNG_INT_RAW** The raw interrupt status bit for I2S_RX_HUNG_INT interrupt. (RO/WTC/SS)

**I2S_TX_HUNG_INT_RAW** The raw interrupt status bit for I2S_TX_HUNG_INT interrupt. (RO/WTC/SS)

**Register 29.2. I2S_INT_ST_REG (0x0010)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | I2S_TX_HUNG_INT_ST | I2S_RX_HUNG_INT_ST | I2S_TX_DONE_INT_ST | I2S_RX_DONE_INT_ST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**I2S_RX_DONE_INT_ST** The masked interrupt status bit for I2S_RX_DONE_INT interrupt. (RO)

**I2S_TX_DONE_INT_ST** The masked interrupt status bit for I2S_TX_DONE_INT interrupt. (RO)

**I2S_RX_HUNG_INT_ST** The masked interrupt status bit for I2S_RX_HUNG_INT interrupt. (RO)

**I2S_TX_HUNG_INT_ST** The masked interrupt status bit for I2S_TX_HUNG_INT interrupt. (RO)

### Register 29.3. I2S_INT_ENA_REG (0x0014)

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | I2S_TX_HUNG_INT_ENA | I2S_RX_HUNG_INT_ENA | I2S_TX_DONE_INT_ENA | I2S_RX_DONE_INT_ENA | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**I2S_RX_DONE_INT_ENA** The interrupt enable bit for I2S_RX_DONE_INT interrupt. (R/W)

**I2S_TX_DONE_INT_ENA** The interrupt enable bit for I2S_TX_DONE_INT interrupt. (R/W)

**I2S_RX_HUNG_INT_ENA** The interrupt enable bit for I2S_RX_HUNG_INT interrupt. (R/W)

**I2S_TX_HUNG_INT_ENA** The interrupt enable bit for I2S_TX_HUNG_INT interrupt. (R/W)

### Register 29.4. I2S_INT_CLR_REG (0x0018)

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | I2S_TX_HUNG_INT_CLR | I2S_RX_HUNG_INT_CLR | I2S_TX_DONE_INT_CLR | I2S_RX_DONE_INT_CLR | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**I2S_RX_DONE_INT_CLR** Set this bit to clear I2S_RX_DONE_INT interrupt. (WT)

**I2S_TX_DONE_INT_CLR** Set this bit to clear I2S_TX_DONE_INT interrupt. (WT)

**I2S_RX_HUNG_INT_CLR** Set this bit to clear I2S_RX_HUNG_INT interrupt. (WT)

**I2S_TX_HUNG_INT_CLR** Set this bit to clear I2S_TX_HUNG_INT interrupt. (WT)

**Register 29.5. I2S_RX_CONF_REG (0x0020)**

| (reserved) | I2S_RX_PDM_EN | I2S_RX_TDM_EN | I2S_RX_BIT_ORDER | I2S_RX_WS_IDLE_POL | I2S_RX_24_FILL_EN | I2S_RX_LEFT_ALIGN | I2S_RX_STOP_MODE | I2S_RX_PCM_BYPASS | I2S_RX_PCM_CONF | I2S_RX_MONO_FST_VLD | I2S_RX_UPDATE | I2S_RX_BIG_ENDIAN | (reserved) | I2S_RX_MONO | (reserved) | I2S_RX_SLAVE_MOD | I2S_RX_START | I2S_RX_FIFO_RESET | I2S_RX_RESET | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 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 | 1 | 0 | 1 | 0x1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**I2S_RX_RESET** Set this bit to reset RX unit. (WT)

**I2S_RX_FIFO_RESET** Set this bit to reset RX FIFO. (WT)

**I2S_RX_START** Set this bit to start receiving data. (R/W)

**I2S_RX_SLAVE_MOD** Set this bit to enable slave RX mode. (R/W)

**I2S_RX_MONO** Set this bit to enable RX unit in mono mode. (R/W)

**I2S_RX_BIG_ENDIAN** I2S RX byte endian. 1: low address data is saved to high address. 0: low address data is saved to low address. (R/W)

**I2S_RX_UPDATE** Set 1 to update I2S RX registers from APB clock domain to I2S RX clock domain. This bit will be cleared by hardware after register update is done. (R/W/SC)

**I2S_RX_MONO_FST_VLD** 1: The first channel data is valid in I2S RX mono mode. 0: The second channel data is valid in I2S RX mono mode. (R/W)

**I2S_RX_PCM_CONF** I2S RX compress/decompress configuration bit. 0 (atol): A-law decompress, 1 (ltoa): A-law compress, 2 (utol): $\mu$-law decompress, 3 (ltou): $\mu$-law compress. (R/W)

**I2S_RX_PCM_BYPASS** Set this bit to bypass Compress/Decompress module for received data. (R/W)

**I2S_RX_STOP_MODE** 0: I2S RX stops only when I2S_RX_START is cleared. 1: I2S RX stops when I2S_RX_START is 0 or in_suc_eof is 1. 2: I2S RX stops when I2S_RX_START is 0 or RX FIFO is full. (R/W)

**I2S_RX_LEFT_ALIGN** 1: I2S RX left alignment mode. 0: I2S RX right alignment mode. (R/W)

**I2S_RX_24_FILL_EN** 1: store 24-bit channel data to 32 bits (Extra bits are filled with zeros). 0: store 24-bit channel data to 24 bits. (R/W)

**I2S_RX_WS_IDLE_POL** 0: WS remains low when receiving left channel data, and remains high when receiving right channel data. 1: WS remains high when receiving left channel data, and remains low when receiving right channel data. (R/W)

**I2S_RX_BIT_ORDER** Configures whether to reverse the bit order of the I2S RX data to be received. 0: Not reverse. 1: Reverse. (R/W)

**I2S_RX_TDM_EN** 1: Enable I2S TDM RX mode. 0: Disable I2S TDM RX mode. (R/W)

**I2S_RX_PDM_EN** 1: Enable I2S PDM RX mode. 0: Disable I2S PDM RX mode. (R/W)

**Register 29.6. I2S_RX_CONF1_REG (0x0028)**

| (reserved) | I2S_RX_MSB_SHIFT | I2S_RX_TDM_CHAN_BITS | I2S_RX_HALF_SAMPLE_BITS | I2S_RX_BITS_MOD | I2S_RX_BCK_DIV_NUM | I2S_RX_TDM_WS_WIDTH | |
|---|---|---|---|---|---|---|---|
| 31 30 | 29 | 28 24 | 23 18 | 17 13 | 12 7 | 6 0 | |
| 0 0 | 1 | 0xf | 0xf | 0xf | 6 | 0x0 | Reset |

**I2S_RX_TDM_WS_WIDTH** The width of rx_ws_out (WS default level) in TDM mode is (I2S_RX_TDM_WS_WIDTH + 1) * T_BCK. (R/W)

**I2S_RX_BCK_DIV_NUM** Configure the divider of BCK in RX mode. Note this divider must not be configured to 1. (R/W)

**I2S_RX_BITS_MOD** Configure the valid data bit length of I2S RX channel. 7: all the valid channel data is in 8-bit mode. 15: all the valid channel data is in 16-bit mode. 23: all the valid channel data is in 24-bit mode. 31: all the valid channel data is in 32-bit mode. (R/W)

**I2S_RX_HALF_SAMPLE_BITS** I2S RX half sample bits. This value x 2 is equal to the BCK cycles in one WS period. (R/W)

**I2S_RX_TDM_CHAN_BITS** Configure RX bit number for each channel in TDM mode. Bit number expected = this value + 1. (R/W)

**I2S_RX_MSB_SHIFT** Control the timing between WS signal and the MSB of data. 1: WS signal changes one BCK clock earlier. 0: Align at rising edge. (R/W)

**Register 29.7. I2S_RX_CLKM_CONF_REG (0x0030)**

| (reserved) | I2S_MCLK_SEL | I2S_RX_CLK_SEL | I2S_RX_CLK_ACTIVE | (reserved) | I2S_RX_CLKM_DIV_NUM | |
|---|---|---|---|---|---|---|
| 31 30 | 29 | 28 27 | 26 | 25 8 | 7 0 | |
| 0 0 | 0 | 0 | 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 2 | Reset |

**I2S_RX_CLKM_DIV_NUM** Integral I2S clock divider value. (R/W)

**I2S_RX_CLK_ACTIVE** Clock enable signal of I2S RX unit. (R/W)

**I2S_RX_CLK_SEL** Select clock source for I2S RX unit. 0: XTAL_CLK. 1: PLL_D2_CLK. 2: PLL_F160M_CLK. 3: I2S_MCLK_in. (R/W)

**I2S_MCLK_SEL** 0: Use I2S TX unit clock as I2S_MCLK_OUT. 1: Use I2S RX unit clock as I2S_MCLK_OUT. (R/W)

### Register 29.8. I2S_TX_PCM2PDM_CONF_REG (0x0040)

| (reserved) | I2S_PCM2PDM_CONV_EN | I2S_TX_PDM_DAC_MODE_EN | I2S_TX_PDM_DAC_2OUT_EN | (reserved) | I2S_TX_PDM_SINC_OSR2 | (reserved) | |
|---|---|---|---|---|---|---|---|
| 31 26 | 25 | 24 | 23 | 22 5 | 4 1 | 0 | |
| 0 0 0 0 0 0 | 0 | 0 | 0 | 0x0 | 0x2 | 0 | Reset |

**I2S_TX_PDM_SINC_OSR2** I2S TX PDM OSR value. (R/W)

**I2S_TX_PDM_DAC_2OUT_EN** 0: 1-line DAC output mode. 1: 2-line DAC output mode. Only valid when I2S_TX_PDM_DAC_MODE_EN is set. (R/W)

**I2S_TX_PDM_DAC_MODE_EN** 0: 1-line PDM output mode. 1: DAC output mode. (R/W)

**I2S_PCM2PDM_CONV_EN** Enable bit for I2S TX PCM-to-PDM conversion. (R/W)

### Register 29.9. I2S_TX_PCM2PDM_CONF1_REG (0x0044)

| (reserved) | I2S_TX_PDM_FS | (reserved) | |
|---|---|---|---|
| 31 20 | 19 10 | 9 0 | |
| 0 0 0 0 0 0 | 480 | 960 | Reset |

**I2S_TX_PDM_FS** I2S PDM TX upsampling parameter. (R/W)

### Register 29.10. I2S_RX_TDM_CTRL_REG (0x0050)

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | I2S_RX_TDM_TOT_CHAN_NUM | | | | I2S_RX_TDM_CHAN15_EN | I2S_RX_TDM_CHAN14_EN | I2S_RX_TDM_CHAN13_EN | I2S_RX_TDM_CHAN12_EN | I2S_RX_TDM_CHAN11_EN | I2S_RX_TDM_CHAN10_EN | I2S_RX_TDM_CHAN9_EN | I2S_RX_TDM_CHAN8_EN | I2S_RX_TDM_PDM_CHAN7_EN | I2S_RX_TDM_PDM_CHAN6_EN | I2S_RX_TDM_PDM_CHAN5_EN | I2S_RX_TDM_PDM_CHAN4_EN | I2S_RX_TDM_PDM_CHAN3_EN | I2S_RX_TDM_PDM_CHAN2_EN | I2S_RX_TDM_PDM_CHAN1_EN | I2S_RX_TDM_PDM_CHAN0_EN | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x0 | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Reset |

**I2S_RX_TDM_PDM_CHAN*n*_EN (*n* = 0 - 7)** 1: Enable the valid data input of I2S RX TDM or PDM channel *n*. 0: Disable. Channel *n* only inputs 0. (R/W)

**I2S_RX_TDM_CHAN*n*_EN (*n* = 8 - 15)** 1: Enable the valid data input of I2S RX TDM channel *n*. 0: Disable. Channel *n* only inputs 0. (R/W)

**I2S_RX_TDM_TOT_CHAN_NUM** The total number of channels in use in I2S RX TDM mode. Total channel number in use = this value + 1. (R/W)

### Register 29.11. I2S_RXEOF_NUM_REG (0x0064)

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | I2S_RX_EOF_NUM | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40 | | | | | | | | | | | | Reset |

**I2S_RX_EOF_NUM** The bit length of RX data is (I2S_RX_BITS_MOD + 1) * (I2S_RX_EOF_NUM + 1). Once the length of received data reaches such bit length, an in_suc_eof interrupt is triggered in the configured DMA RX channel. (R/W)

**Register 29.12. I2S_TX_CONF_REG (0x0024)**

| 31–28 | 27 | 26–24 | 23–21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11–10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | I2S_SIG_LOOPBACK | I2S_TX_CHAN_MOD | (reserved) | I2S_TX_PDM_EN | I2S_TX_TDM_EN | I2S_TX_BIT_ORDER | I2S_TX_WS_IDLE_POL | I2S_TX_24_FILL_EN | I2S_TX_LEFT_ALIGN | (reserved) | I2S_TX_STOP_EN | I2S_TX_PCM_BYPASS | I2S_TX_PCM_CONF | I2S_TX_MONO_FST_VLD | I2S_TX_UPDATE | I2S_TX_BIG_ENDIAN | I2S_TX_CHAN_EQUAL | I2S_TX_MONO | (reserved) | I2S_TX_SLAVE_MOD | I2S_TX_START | I2S_TX_FIFO_RESET | I2S_TX_RESET | |
| 0 0 0 0 | 0 | 0 | 0 0 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0x0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**I2S_TX_RESET** Set this bit to reset TX unit. (WT)

**I2S_TX_FIFO_RESET** Set this bit to reset TX FIFO. (WT)

**I2S_TX_START** Set this bit to start transmitting data. (R/W)

**I2S_TX_SLAVE_MOD** Set this bit to enable slave TX mode. (R/W)

**I2S_TX_MONO** Set this bit to enable TX unit in mono mode. (R/W)

**I2S_TX_CHAN_EQUAL** 1: The left channel data is equal to right channel data in I2S TX mono mode or TDM mode. 0: The invalid channel data is I2S_SINGLE_DATA in I2S TX mono mode or TDM mode. (R/W)

**I2S_TX_BIG_ENDIAN** I2S TX byte endian. 1: low address data is saved to high address. 0: low address data is saved to low address. (R/W)

**I2S_TX_UPDATE** Set 1 to update I2S TX registers from APB clock domain to I2S TX clock domain. This bit will be cleared by hardware after register update is done. (R/W/SC)

**I2S_TX_MONO_FST_VLD** 1: The first channel data is valid in I2S TX mono mode. 0: The second channel data is valid in I2S TX mono mode. (R/W)

**I2S_TX_PCM_CONF** I2S TX compress/decompress configuration bits. 0 (atol): A-law decompress, 1 (ltoa): A-law compress, 2 (utol): $\mu$-law decompress, 3 (ltou): $\mu$-law compress. (R/W)

**I2S_TX_PCM_BYPASS** Set this bit to bypass Compress/Decompress module for transmitted data. (R/W)

**I2S_TX_STOP_EN** Set this bit to stop outputting BCK signal and WS signal when TX FIFO is empty. (R/W)

**I2S_TX_LEFT_ALIGN** 1: I2S TX left alignment mode. 0: I2S TX right alignment mode. (R/W)

**I2S_TX_24_FILL_EN** 1: Sent 32 bits in 24-bit channel data mode. (Extra bits are filled with zeros). 0: Sent 24 bits in 24-bit channel data mode. (R/W)

**I2S_TX_WS_IDLE_POL** 0: WS remains low when sending left channel data, and remains high when sending right channel data. 1: WS remains high when sending left channel data, and remains low when sending right channel data. (R/W)

**I2S_TX_BIT_ORDER** Configures whether to reverse the bit order of valid data to be sent by the I2S TX. 0: Not reverse. 1: Reverse. (R/W)

**I2S_TX_TDM_EN** 1: Enable I2S TDM TX mode. 0: Disable I2S TDM TX mode. (R/W)

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

### Register 29.12. I2S_TX_CONF_REG (0x0024)

Continued from the previous page...

**I2S_TX_PDM_EN** 1: Enable I2S PDM TX mode. 0: Disable I2S PDM TX mode. (R/W)

**I2S_TX_CHAN_MOD** I2S TX channel configuration bits. For more information, see Table 29.9-4. (R/W)

**I2S_SIG_LOOPBACK** Enable signal loop back mode with TX unit and RX unit sharing the same WS and BCK signals. (R/W)

### Register 29.13. I2S_TX_CONF1_REG (0x002C)

| (reserved) | I2S_TX_BCK_NO_DLY | I2S_TX_MSB_SHIFT | I2S_TX_TDM_CHAN_BITS | I2S_TX_HALF_SAMPLE_BITS | I2S_TX_BITS_MOD | I2S_TX_BCK_DIV_NUM | I2S_TX_TDM_WS_WIDTH | |
|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 24 | 23 18 | 17 13 | 12 7 | 6 0 | |
| 0 | 1 | 1 | 0xf | 0xf | 0xf | 6 | 0x0 | Reset |

**I2S_TX_TDM_WS_WIDTH** The width of tx_ws_out (WS default level) in TDM mode is (I2S_TX_TDM_WS_WIDTH + 1) * T_BCK. (R/W)

**I2S_TX_BCK_DIV_NUM** Configure the divider of BCK in TX mode. Note this divider must not be configured to 1. (R/W)

**I2S_TX_BITS_MOD** Set the bits to configure the valid data bit length of I2S TX channel. 7: all the valid channel data is in 8-bit mode. 15: all the valid channel data is in 16-bit mode. 23: all the valid channel data is in 24-bit mode. 31: all the valid channel data is in 32-bit mode. (R/W)

**I2S_TX_HALF_SAMPLE_BITS** I2S TX half sample bits. This value x 2 is equal to the BCK cycles in one WS period. (R/W)

**I2S_TX_TDM_CHAN_BITS** Configure TX bit number for each channel in TDM mode. Bit number expected = this value + 1. (R/W)

**I2S_TX_MSB_SHIFT** Control the timing between WS signal and the MSB of data. 1: WS signal changes one BCK clock earlier. 0: Align at rising edge. (R/W)

**I2S_TX_BCK_NO_DLY** 1: BCK is not delayed to generate rising/falling edge in master mode. 0: BCK is delayed to generate rising/falling edge in master mode. (R/W)

### Register 29.14. I2S_TX_CLKM_CONF_REG (0x0034)

| (reserved) | I2S_CLK_EN | I2S_TX_CLK_SEL | I2S_TX_CLK_ACTIVE | (reserved) | I2S_TX_CLKM_DIV_NUM | |
|---|---|---|---|---|---|---|
| 31 30 | 29 | 28 27 | 26 | 25 8 | 7 0 | |
| 0 0 | 0 | 0 | 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 2 | Reset |

**I2S_TX_CLKM_DIV_NUM** Integral I2S TX clock divider value. (R/W)

**I2S_TX_CLK_ACTIVE** I2S TX unit clock enable signal. (R/W)

**I2S_TX_CLK_SEL** Select clock clock for I2S TX unit. 0: XTAL_CLK. 1: PLL_D2_CLK. 2: PLL_F160M_CLK. 3: I2S_MCLK_in. (R/W)

**I2S_CLK_EN** Set this bit to enable clock gate. (R/W)

### Register 29.15. I2S_TX_TDM_CTRL_REG (0x0054)

| (reserved) | I2S_TX_TDM_SKIP_MSK_EN | I2S_TX_TDM_TOT_CHAN_NUM | I2S_TX_TDM_CHAN15_EN | I2S_TX_TDM_CHAN14_EN | I2S_TX_TDM_CHAN13_EN | I2S_TX_TDM_CHAN12_EN | I2S_TX_TDM_CHAN11_EN | I2S_TX_TDM_CHAN10_EN | I2S_TX_TDM_CHAN9_EN | I2S_TX_TDM_CHAN8_EN | I2S_TX_TDM_CHAN7_EN | I2S_TX_TDM_CHAN6_EN | I2S_TX_TDM_CHAN5_EN | I2S_TX_TDM_CHAN4_EN | I2S_TX_TDM_CHAN3_EN | I2S_TX_TDM_CHAN2_EN | I2S_TX_TDM_CHAN1_EN | I2S_TX_TDM_CHAN0_EN | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 21 | 20 | 19 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 | 0x0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | Reset |

**I2S_TX_TDM_CHAN*n*_EN (*n* = 0 - 15)** 1: Enable the valid data output of I2S TX TDM channel *n*. 0: Channel TX data is controlled by I2S_TX_CHAN_EQUAL and I2S_SINGLE_DATA. See Section 29.9.2.1. (R/W)

**I2S_TX_TDM_TOT_CHAN_NUM** Set the total number of channels in use in I2S TX TDM mode. Total channel number in use = this value + 1. (R/W)

**I2S_TX_TDM_SKIP_MSK_EN** When DMA TX buffer stores the data of (I2S_TX_TDM_TOT_CHAN_NUM + 1) channels, and only the data of the enabled channels is sent, then this bit should be set. Clear it when all the data stored in DMA TX buffer is for enabled channels. (R/W)

**Register 29.16. I2S_RX_CLKM_DIV_CONF_REG (0x0038)**

| (reserved) | I2S_RX_CLKM_DIV_YN1 | I2S_RX_CLKM_DIV_X | I2S_RX_CLKM_DIV_Y | I2S_RX_CLKM_DIV_Z | |
|---|---|---|---|---|---|
| 31 28 | 27 | 26 18 | 17 9 | 8 0 | |
| 0 0 0 0 | 0 | 0x0 | 0x1 | 0x0 | Reset |

**I2S_RX_CLKM_DIV_Z** For b <= a/2, the value of I2S_RX_CLKM_DIV_Z is b. For b > a/2, the value of I2S_RX_CLKM_DIV_Z is (a - b). (R/W)

**I2S_RX_CLKM_DIV_Y** For b <= a/2, the value of I2S_RX_CLKM_DIV_Y is (a%b). For b > a/2, the value of I2S_RX_CLKM_DIV_Y is (a%(a - b)). (R/W)

**I2S_RX_CLKM_DIV_X** For b <= a/2, the value of I2S_RX_CLKM_DIV_X is floor(a/b) - 1. For b > a/2, the value of I2S_RX_CLKM_DIV_X is floor(a/(a - b)) - 1. (R/W)

**I2S_RX_CLKM_DIV_YN1** For b <= a/2, the value of I2S_RX_CLKM_DIV_YN1 is 0. For b > a/2, the value of I2S_RX_CLKM_DIV_YN1 is 1. (R/W)

> **Note:**
> "a" and "b" represent the denominator and the numerator of fractional divider, respectively. For more information, see Section 29.6.

**Register 29.17. I2S_RX_TIMING_REG (0x0058)**

| (reserved) | I2S_RX_BCK_IN_DM | (reserved) | I2S_RX_WS_IN_DM | (reserved) | I2S_RX_BCK_OUT_DM | (reserved) | I2S_RX_WS_OUT_DM | (reserved) | I2S_RX_SD_IN_DM | |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 30 | 29 28 | 27 26 | 25 24 | 23 22 | 21 20 | 19 18 | 17 16 | 15 2 | 1 0 | |
| 0 0 | 0x0 | 0 0 | 0x0 | 0 0 | 0x0 | 0 0 | 0x0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**I2S_RX_SD_IN_DM** The delay mode of I2S RX SD input signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_RX_WS_OUT_DM** The delay mode of I2S RX WS output signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_RX_BCK_OUT_DM** The delay mode of I2S RX BCK output signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_RX_WS_IN_DM** The delay mode of I2S RX WS input signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_RX_BCK_IN_DM** The delay mode of I2S RX BCK input signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**Register 29.18. I2S_TX_CLKM_DIV_CONF_REG (0x003C)**

| (reserved) | I2S_TX_CLKM_DIV_YN1 | I2S_TX_CLKM_DIV_X | I2S_TX_CLKM_DIV_Y | I2S_TX_CLKM_DIV_Z | |
|---|---|---|---|---|---|
| 31 28 | 27 | 26 18 | 17 9 | 8 0 | |
| 0 0 0 0 | 0 | 0x0 | 0x1 | 0x0 | Reset |

**I2S_TX_CLKM_DIV_Z** For b <= a/2, the value of I2S_TX_CLKM_DIV_Z is b. For b > a/2, the value of I2S_TX_CLKM_DIV_Z is (a - b). (R/W)

**I2S_TX_CLKM_DIV_Y** For b <= a/2, the value of I2S_TX_CLKM_DIV_Y is (a%b). For b > a/2, the value of I2S_TX_CLKM_DIV_Y is (a%(a - b)). (R/W)

**I2S_TX_CLKM_DIV_X** For b <= a/2, the value of I2S_TX_CLKM_DIV_X is floor(a/b) - 1. For b > a/2, the value of I2S_TX_CLKM_DIV_X is floor(a/(a - b)) - 1. (R/W)

**I2S_TX_CLKM_DIV_YN1** For b <= a/2, the value of I2S_TX_CLKM_DIV_YN1 is 0. For b > a/2, the value of I2S_TX_CLKM_DIV_YN1 is 1. (R/W)

**Note:**
"a" and "b" represent the denominator and the numerator of fractional divider, respectively. For more information, see Section 29.6.

Register 29.19. I2S_TX_TIMING_REG (0x005C)

| (reserved) | I2S_TX_BCK_IN_DM | (reserved) | I2S_TX_WS_IN_DM | (reserved) | I2S_TX_BCK_OUT_DM | (reserved) | I2S_TX_WS_OUT_DM | (reserved) | I2S_TX_SD1_OUT_DM | (reserved) | I2S_TX_SD_OUT_DM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 30 | 29 28 | 27 26 | 25 24 | 23 22 | 21 20 | 19 18 | 17 16 | 15 6 | 5 4 | 3 2 | 1 0 | |
| 0 0 | 0x0 | 0 0 | 0x0 | 0 0 | 0x0 | 0 0 | 0x0 | 0 0 0 0 0 0 0 0 0 0 | 0x0 | 0 0 | 0x0 | Reset |

**I2S_TX_SD_OUT_DM** The delay mode of I2S TX SD output signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_TX_SD1_OUT_DM** The delay mode of I2S TX SD1 output signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_TX_WS_OUT_DM** The delay mode of I2S TX WS output signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_TX_BCK_OUT_DM** The delay mode of I2S TX BCK output signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_TX_WS_IN_DM** The delay mode of I2S TX WS input signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

**I2S_TX_BCK_IN_DM** The delay mode of I2S TX BCK input signal. 0: bypass. 1: delay by rising edge. 2: delay by falling edge. 3: not used. (R/W)

Register 29.20. I2S_LC_HUNG_CONF_REG (0x0060)

| (reserved) | I2S_LC_FIFO_TIMEOUT_ENA | I2S_LC_FIFO_TIMEOUT_SHIFT | I2S_LC_FIFO_TIMEOUT | |
|---|---|---|---|---|
| 31 12 | 11 | 10 8 | 7 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 | 0 | 0x10 | Reset |

**I2S_LC_FIFO_TIMEOUT** I2S_TX_HUNG_INT or I2S_RX_HUNG_INT interrupt will be triggered when FIFO hung counter is equal to this value. (R/W)

**I2S_LC_FIFO_TIMEOUT_SHIFT** The bits are used to scale tick counter threshold. The tick counter is reset when counter value >= $88000/2^{I2S_LC_FIFO_TIMEOUT_SHIFT}$. (R/W)

**I2S_LC_FIFO_TIMEOUT_ENA** The enable bit for FIFO timeout. (R/W)

### Register 29.21. I2S_CONF_SIGLE_DATA_REG (0x0068)

| I2S_SINGLE_DATA (31:0) | |
|---|---|
| 0 | Reset |

**I2S_SINGLE_DATA** The configured constant channel data to be sent out. (R/W)

### Register 29.22. I2S_STATE_REG (0x006C)

| (reserved) (31:1) | I2S_TX_IDLE (0) | |
|---|---|---|
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 1 | Reset |

**I2S_TX_IDLE** 1: I2S TX unit is in idle state. 0: I2S TX unit is working. (RO)

### Register 29.23. I2S_DATE_REG (0x0080)

| (reserved) (31:28) | I2S_DATE (27:0) | |
|---|---|---|
| 0 0 0 0 | 0x2007220 | Reset |

**I2S_DATE** Version control register. (R/W)

# Chapter 30

# USB Serial/JTAG Controller (USB_SERIAL_JTAG)

The ESP32-C3 contains an USB Serial/JTAG Controller. This unit can be used to program the SoC's flash, read program output, as well as attach a debugger to the running program. All of these are possible for any computer with a USB host ('host' in the rest of this text) without any active external components.

## 30.1 Overview

While programming and debugging an ESP32-C3 project using the UART and JTAG functionality is certainly possible, it has a few downsides. First of all, both UART and JTAG take up IO pins and as such, fewer pins are left usable for controlling external signals in software. Additionally, an external chip or adapter is needed for both UART and JTAG to interface with a host computer, which means it will be necessary to integrate these two functionalities in the form of external chips or debugging adapters.

In order to alleviate these issues,, as well as to negate the need for external devices, the ESP32-C3 contains an USB Serial/JTAG Controller, which integrates the functionality of both an USB-to-serial converter as well as those of an USB-to-JTAG adapter. As this device directly interfaces to an external USB host using only the two data lines required by USB2.0, debugging the ESP32-C3 only requires two pins to be dedicated to this functionality.

## 30.2 Features

- USB Full-speed device.
- Fixed function device, hardwired for CDC-ACM (Communication Device Class - Abstract Control Model) and JTAG adapter functionality.
- 2 OUT Endpoints, 3 IN Endpoints in addition to Control Endpoint 0; Up to 64-byte data payload size.
- Internal PHY, so no or very few external components needed to connect to a host computer.
- CDC-ACM adherent serial port emulation is plug-and-play on most modern OSes.
- JTAG interface allows fast communication with CPU debug core using a compact representation of JTAG instructions.
- CDC-ACM supports host controllable chip reset and entry into download mode.

As shown in Figure 30.2-1, the USB Serial/JTAG Controller consists of an USB PHY, a USB device interface, a JTAG command processor and a response capture unit, as well as the CDC-ACM registers. The PHY and part of the device interface are clocked from a 48 MHz clock derived from the main PLL, the rest of the logic is clocked from APB_CLK. The JTAG command processor is connected to the JTAG debug unit of the main processor; the CDC-ACM registers are connected to the APB bus and as such can be read from and written to by software running on the main CPU.

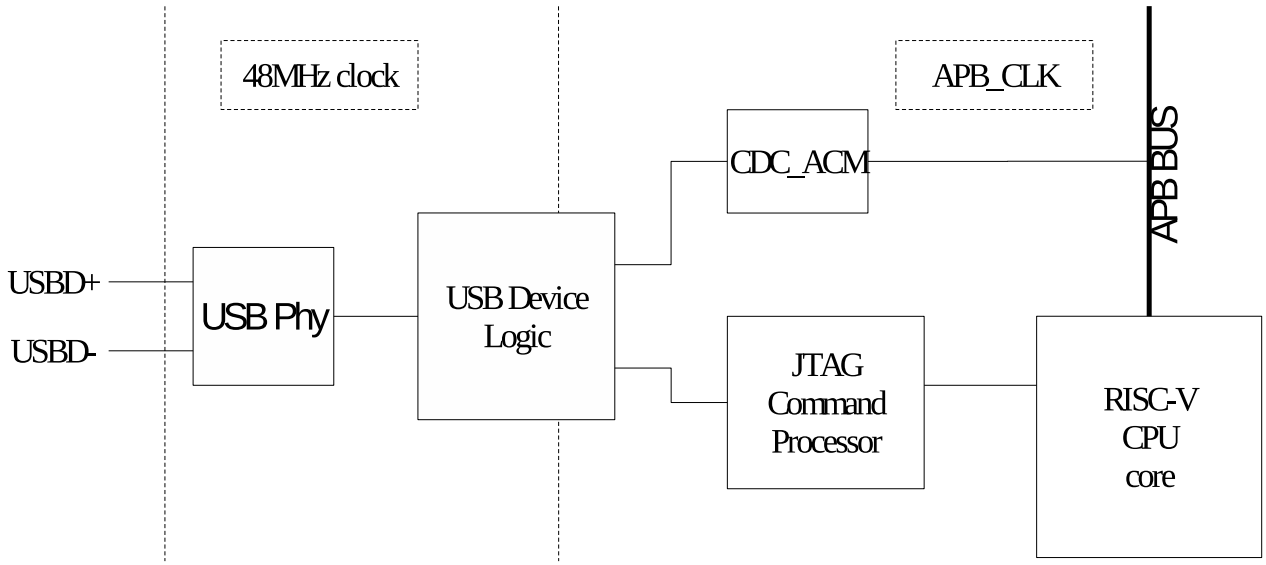



Figure 30.2-1. USB Serial/JTAG High Level Diagram

Note that while the USB Serial/JTAG device is a USB 2.0 device, it only supports Full-speed (12 Mbps) and not the High-speed (480 Mbps) mode the USB2.0 standard introduced.

Figure 30.2-2 shows the internal details of the USB Serial/JTAG controller on the USB side. The USB Serial/JTAG Controller consists of an USB 2.0 Full Speed device. It contains a control endpoint, a dummy interrupt endpoint, two bulk input endpoints as well as two bulk output endpoints. Together, these form an USB Composite device, which consists of an CDC-ACM USB class device as well as a vendor-specific device implementing the JTAG interface. On the SoC side, the JTAG interface is directly connected to the RISC-V CPU's debugging interface, allowing debugging of programs running on that core. Meanwhile, the CDC-ACM device is exposed as a set of registers, allowing a program on the CPU to read and write from this. Additionally, the ROM startup code of the SoC contains code allowing the user to reprogram attached flash memory using this interface.

## 30.3 Functional Description

The USB Serial/JTAG Controller interfaces with an USB host processor on one side, and the CPU debug hardware as well as the software running on the USB port on the other side.

### 30.3.1 CDC-ACM USB Interface Functional Description

The CDC-ACM interface adheres to the standard USB CDC-ACM class for serial port emulation. It contains a dummy interrupt endpoint (which will never send any events, as they are not implemented nor needed) and a Bulk IN as well as a Bulk OUT endpoint for the host's received and sent serial data respectively. These endpoints can handle 64-byte packets at a time, allowing for high throughput. As CDC-ACM is a standard USB device class, a host generally does not need any special installation procedures for it to function: when the USB debugging device is properly connected to a host, the operating system should show a new serial port moments later.

The CDC-ACM interface accepts the following standard CDC-ACM control requests:

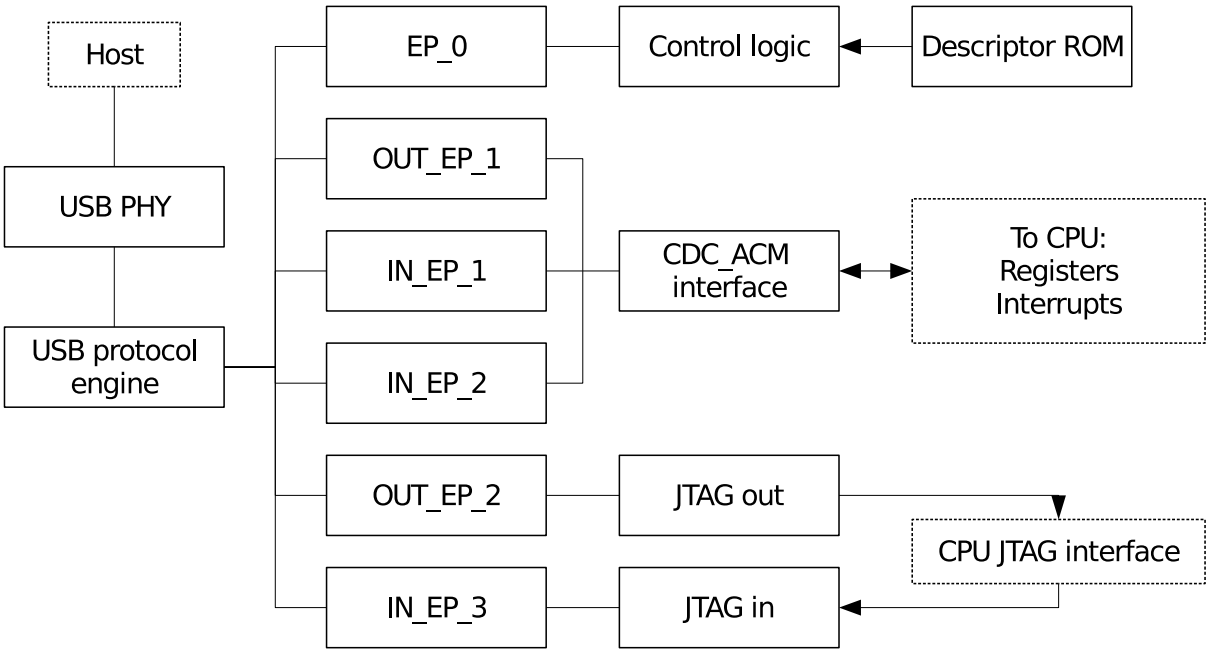



Figure 30.2-2. USB Serial/JTAG Block Diagram

Table 30.3-1. Standard CDC-ACM Control Requests

| Command | Action |
|---|---|
| SEND_BREAK | Accepted but ignored (dummy) |
| SET_LINE_CODING | Accepted but ignored (dummy) |
| GET_LINE_CODING | Always returns 9600 baud, no parity, 8 databits, 1 stopbit |
| SET_CONTROL_LINE_STATE | Set the state of the RTS/DTR lines, see Table 30.3-2 |

Aside from general-purpose communication, the CDC-ACM interface also can be used to reset the ESP32-C3 and optionally make it go into download mode in order to flash new firmware. This is done by setting the RTS and DTR lines on the virtual serial port.

Table 30.3-2. CDC-ACM Settings with RTS and DTR

| RTS | DTR | Action |
|---|---|---|
| 0 | 0 | Clear download mode flag |
| 0 | 1 | Set download mode flag |
| 1 | 0 | Reset ESP32-C3 |
| 1 | 1 | No action |

Note that if the download mode flag is set when the ESP32-C3 is reset, the ESP32-C3 will reboot into download mode. When this flag is cleared and the chip is reset, the ESP32-C3 will boot from flash. For specific sequences, please refer to Section 30.4. All these functions can also be disabled by programming various eFuses, please refer to Chapter 4 *eFuse Controller (EFUSE)* for more details.

### 30.3.2 CDC-ACM Firmware Interface Functional Description

As the USB Serial/JTAG Controller is connected to the internal APB bus of the ESP32-C3, the CPU can interact with it. This is mainly used to read and write data from and to the virtual serial port on the attached host.

USB CDC-ACM serial data is sent to and received from the host in packets of 0 to 64 bytes in size. When enough CDC-ACM data has accumulated in the host, the host will send a packet to the CDC-ACM receive endpoint, and when the USB Serial/JTAG Controller has a free buffer, it will accept this packet. Conversely, the host will check periodically if the USB Serial/JTAG Controller has a packet ready to be sent to the host, and if so, receive this packet.

Firmware can get notified of new data from the host in one of two ways. First of all, the USB_SERIAL_JTAG_SERIAL_OUT_EP_DATA_AVAIL bit will remain set to one as long as there still is unread host data in the buffer. Secondly, the availability of data will trigger the USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT interrupt as well.

When data is available, it can be read by firmware by repeatedly reading bytes from USB_SERIAL_JTAG_EP1_REG. The amount of bytes to read can be determined by checking the USB_SERIAL_JTAG_SERIAL_OUT_EP_DATA_AVAIL bit after reading each byte to see if there is more data to read. After all data is read, the USB debug device is automatically readied to receive a new data packet from the host.

When the firmware has data to send, it can do so by putting it in the send buffer and triggering a flush, allowing the host to receive the data in a USB packet. In order to do so, there needs to be space available in the send buffer. Firmware can check this by reading USB_REG_SERIAL_IN_EP_DATA_FREE; a one in this register field indicates there is still free room in the buffer. While this is the case, firmware can fill the buffer by writing bytes to the USB_SERIAL_JTAG_EP1_REG register.

Writing the buffer doesn't immediately trigger sending data to the host. This does not happen until the buffer is flushed; a flush causes the entire buffer to be readied for reception by the USB host at once. A flush can be triggered in two ways: after the 64th byte is written to the buffer, the USB hardware will automatically flush the buffer to the host. Alternatively, firmware can trigger a flush by writing a one to USB_REG_SERIAL_WR_DONE.

Regardless of how a flush is triggered, the send buffer will be unavailable for firmware to write into until it has been fully read by the host. As soon as this happens, the USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT interrupt will be triggered, indicating the send buffer can receive another 64 bytes.

### 30.3.3 USB-to-JTAG Interface

The USB-to-JTAG interface uses a vendor-specific class for its implementation. It consists of two endpoints, one to receive commands and one to send responses. Additionally, some less time-sensitive commands can be given as control requests.

### 30.3.4 JTAG Command Processor

Commands from the host to the JTAG interface are interpreted by the JTAG command processor. Internally, the JTAG command processor implements a full four-wire JTAG bus, consisting of the TCK, TMS and TDI output lines to the RISC-V CPU, as well as the TDO line signalling back from the CPU to the JTAG response

capture unit. These signals adhere to the IEEE 1149.1 JTAG standards. Additionally, there is a SRST line to reset the SoC.

The JTAG command processor parses each received nibble (4-bit value) as a command. As USB data is received in 8-bit bytes, this means each byte contains two commands. The USB command processor will execute high-nibble first and low-nibble second. The commands are used to control the TCK, TMS, TDI, and SRST lines of the internal JTAG bus, as well as signal the JTAG response capture unit that the state of the TDO line (which is driven by the CPU debug logic) needs to be captured.

Of this internal JTAG bus, TCK, TMS, TDI and TDO are connected directly to the JTAG debugging logic of the RISC-V CPU. SRST is connected to the reset logic of the digital circuitry in the SoC and a high level on this line will cause a digital system reset. Note that the USB Serial/JTAG Controller itself is not affected by SRST.

A nibble can contain the following commands:

**Table 30.3-3. Commands of a Nibble**

| bit | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| CMD_CLK | 0 | cap | tms | tdi |
| CMD_RST | 1 | 0 | 0 | srst |
| CMD_FLUSH | 1 | 0 | 1 | 0 |
| CMD_RSV | 1 | 0 | 1 | 1 |
| CMD_REP | 1 | 1 | R1 | R0 |

- CMD_CLK will set the TDI and TMS to the indicated values and emit one clock pulse on TCK. If the CAP bit is 1, it will also instruct the JTAG response capture unit to capture the state of the TDO line. This instruction forms the basis of JTAG communication.
- CMD_RST will set the state of the SRST line to the indicated value. This can be used to reset the ESP32-C3.
- CMD_FLUSH will instruct the JTAG response capture unit to flush the buffer of all bits it collected so the host is able to read them. Note that in some cases, a JTAG transaction will end in an odd number of commands and as such an odd number of nibbles. In this case, it is allowable to repeat the CMD_FLUSH to get an even number of nibbles fitting an integer number of bytes.
- CMD_RSV is reserved in the current implementation. The ESP32-C3 will ignore this command when it receives it.
- CMD_REP repeats the last (non-CMD_REP) command a certain number of times. It's intended goal is to compress command streams which repeat the same CMD_CLK instruction multiple times. A command like CMD_CLK can be followed by multiple CMD_REP commands. The number of repetitions done by one CMD_REP can be expressed as $no_repetitions = (R1 \times 2 + R0) \times (4^{cmd_rep_count})$, where cmd_rep_count is how many CMD_REP instructions went directly before it. Note that the CMD_REP is only intended to repeat a CMD_CLK command. Specifically, using it on a CMD_FLUSH command may lead to an unresponsive USB device, needing an USB reset to recover.

### 30.3.5 USB-to-JTAG Interface: CMD_REP usage example

Here is a list of commands as an illustration of the use of CMD_REP. Note each command is a nibble; in this example the bytewise command stream would be 0x0D 0x5E 0xCF.

1. 0x0 (CMD_CLK: cap=0, tdi=0, tms=0)
2. 0xD (CMD_REP: R1=0, R0=1)
3. 0x5 (CMD_CLK: cap=1, tdi=0, tms=1)
4. 0xE (CMD_REP: R1=1, R0=0)
5. 0xC (CMD_REP: R1=0, R0=0)
6. 0xF (CMD_REP: R1=1, R0=1)

This is what happens at every step:

1. TCK is clocked with the TDI and TMS lines set to 0. No data is captured.
2. TCK is clocked another $(0 \times 2 + 1) \times (4^0) = 1$ time with the same settings as step 1.
3. TCK is clocked with the TDI line set to 0 and TMS set to 1. Data on the TDO line is captured.
4. TCK is clocked another $(1 \times 2 + 0) \times (4^0) = 2$ times with the same settings as step 3.
5. Nothing happens: $(0 \times 2 + 0) \times (4^1) = 0$. Note that this does increase cmd_rep_count for the next step.
6. TCK is clocked another $(1 \times 2 + 1) \times (4^2) = 48$ times with the same settings as step 3.

In other words: This example stream has the same net effect as command 1 twice, then repeating command 3 for 51 times.

### 30.3.6 USB-to-JTAG Interface: Response Capture Unit

The response capture unit reads the TDO line of the internal JTAG bus and captures its value when the command parser executes a CMD_CLK with cap=1. It puts this bit into an internal shift register, and writes a byte into the USB buffer when 8 bits have been collected. Of these 8 bits, the least significant one is the one that is read from TDO the earliest.

As soon as either 64 bytes (512 bits) have been collected or a CMD_FLUSH command is executed, the response capture unit will make the buffer available for the host to receive. Note that the interface to the USB logic is double-buffered. This way, as long as USB throughput is sufficient, the response capture unit can always receive more data: while one of the buffers is waiting to be sent to the host, the other one can receive more data. When the host has received data from its buffer and the response capture unit flushes its buffer, the two buffers change position.

This also means that a command stream can cause at most 128 bytes of capture data to be generated (less if there are flush commands in the stream) without the host acting to receive the generated data. If more data is generated anyway, the command stream is paused and the device will not accept more commands before the generated capture data is read out.

Note that in general, the logic of the response capture unit tries not to send zero-byte responses: for instance, sending a series of CMD_FLUSH commands will not cause a series of zero-byte USB responses to be sent. However, in the current implementation, some zero-byte responses may be generated in extraordinary circumstances. It's recommended to ignore these responses.

### 30.3.7 USB-to-JTAG Interface: Control Transfer Requests

Aside from the command processor and the response capture unit, the USB-to-JTAG interface also understands some control requests, as documented in the table below:

Table 30.3-4. USB-to-JTAG Control Requests

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 01000000b | 0 (VEND_JTAG_SETDIV) | [divider] | interface | 0 | None |
| 01000000b | 1 (VEND_JTAG_SETIO) | [iobits] | interface | 0 | None |
| 11000000b | 2 (VEND_JTAG_GETTDO) | 0 | interface | 1 | [iostate] |
| 10000000b | 6 (GET_DESCRIPTOR) | 0x2000 | 0 | 256 | [jtag cap desc] |

- VEND_JTAG_SETDIV sets the divider used. This directly affects the duration of a TCK clock pulse. The TCK clock pulses are derived from APB_CLK, which is divided down using an internal divider. This control request allows the host to set this divider. Note that on startup, the divider is set to 2, meaning the TCK clock rate will generally be 40 MHz.
- VEND_JTAG_SETIO can bypass the JTAG command processor to set the internal TDI, TDO, TMS and SRST lines to given values. These values are encoded in the wValue field in the format of 11'b0, srst, trst, tck, tms, tdi.
- VEND_JTAG_GETTDO can bypass the JTAG response capture unit to read the internal TDO signal directly. This request returns one byte of data, of which the least significant bit represents the status of the TDO line.
- GET_DESCRIPTOR is a standard USB request, however it can also be used with a vendor-specific wValue of 0x2000 to get the JTAG capabilities descriptor. This returns a certain amount of bytes representing the following fixed structure, which describes the capabilities of the USB-to-JTAG adapter. This structure allows host software to automatically support future revisions of the hardware without needing an update.

The JTAG capabilities descriptor of the ESP32-C3 is as follows. Note that all 16-bit values are little-endian.

Table 30.3-5. JTAG Capabilities Descriptor

| Byte | Value | Description |
|---|---|---|
| 0 | 1 | JTAG protocol capabilities structure version |
| 1 | 10 | Total length of JTAG protocol capabilities |
| 2 | 1 | Type of this struct: 1 for speed capabilities struct |
| 3 | 8 | Length of this speed capabilities struct |
| 4 ~ 5 | 8000 | APB_CLK speed in 10 kHz increments. Note that the maximal TCK speed is half of this |
| 6 ~ 7 | 1 | Minimum divisor settable by the VEND_JTAG_SETDIV request |
| 8 ~ 9 | 255 | Maximum divisor settable by the VEND_JTAG_SETDIV request |

## 30.4 Recommended Operation

**Note:**
When burning EFUSE_DIS_PAD_JTAG and EFUSE_DIS_USB_JTAG, e.g., when preparing for secure boot, the USB Serial/JTAG controller may lose connection until ESP32-C3 reboots.

There is very little setup needed in order to use the USB Serial/JTAG Device. The USB-to-JTAG hardware itself does not need any setup aside from the standard USB initialization the host operating system already does. The CDC-ACM emulation, on the host side, also is plug-and-play.

On the firmware side, very little initialization should be needed either: the USB hardware is self-initializing and after boot-up, if a host is connected and listening on the CDC-ACM interface, data can be exchanged as described above without any specific setup aside from the firmware optionally setting up an interrupt service handler.

One thing to note is that there may be situations where the host is either not attached or the CDC-ACM virtual port is not opened. In this case, the packets that are flushed to the host will never be picked up and the transmit buffer will never be empty. It is important to detect this and time out, as this is the only way to reliably detect that the port on the host side is closed.

Another thing to note is that the USB device is dependent on both the PLL for the 48 MHz USB PHY clock, as well as APB_CLK. Specifically, an APB_CLK of 40 MHz or more is required for proper USB compliant operation, although the USB device will still function with most hosts with an APB_CLK as low as 10 MHz. Behaviour shown when this happens is dependent on the host USB hardware and drivers, and can include the device being unresponsive and it disappearing when first accessed.

More specifically, the APB_CLK will be affected by clock gating the USB Serial/JTAG Controller, which may happen in Light Sleep. Additionally, the USB serial/JTAG Controller (as well as the attached RISC-V CPU) will be entirely powered down in Deep Sleep mode. If a device needs to be debugged in either of these two modes, it may be preferable to use an external JTAG debugger and serial interface instead.

The CDC-ACM interface can also be used to reset the SoC and take it into or out of download mode. Generating the correct sequence of handshake signals can be a bit complicated: Most operating systems only allow setting or resetting DTR and RTS separately, and not in tandem. Additionally, some drivers (e.g. the standard CDC-ACM driver on Windows) do not set DTR until RTS is set and the user needs to explicitly set RTS in order to 'propagate' the DTR value. These are the recommended procedures:

To reset the SoC into download mode:

**Table 30.4-1. Reset SoC into Download Mode**

| Action | Internal state | Note |
|---|---|---|
| Clear DTR | RTS=?, DTR=0 | Initialize to known values |
| Clear RTS | RTS=0, DTR=0 | - |
| Set DTR | RTS=0, DTR=1 | Set download mode flag |
| Clear RTS | RTS=0, DTR=1 | Propagate DTR |
| Set RTS | RTS=1, DTR=1 | - |
| Clear DTR | RTS=1, DTR=0 | Reset SoC |
| Set RTS | RTS=1, DTR=0 | Propagate DTR |
| Clear RTS | RTS=0, DTR=0 | Clear download flag |

To reset the SoC into booting from flash:

Table 30.4-2. Reset SoC into Booting

| Action | Internal state | Note |
|---|---|---|
| Clear DTR | RTS=?, DTR=0 | - |
| Clear RTS | RTS=0, DTR=0 | Clear download flag |
| Set RTS | RTS=1, DTR=0 | Reset SoC |
| Clear RTS | RTS=0, DTR=0 | Exit reset |

## 30.5 Register Summary

The addresses in this section are relative to USB Serial/JTAG Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Registers** | | | |
| USB_SERIAL_JTAG_EP1_REG | FIFO access for the CDC-ACM data IN and OUT endpoints | 0x0000 | R/W |
| USB_SERIAL_JTAG_CONF0_REG | PHY hardware configuration | 0x0018 | R/W |
| USB_SERIAL_JTAG_TEST_REG | Registers used for debugging the PHY | 0x001C | R/W |
| USB_SERIAL_JTAG_MISC_CONF_REG | Clock enable control | 0x0044 | R/W |
| USB_SERIAL_JTAG_MEM_CONF_REG | Memory power control | 0x0048 | R/W |
| **Status Registers** | | | |
| USB_SERIAL_JTAG_EP1_CONF_REG | Configuration and control registers for the CDC-ACM FIFOs | 0x0004 | varies |
| USB_SERIAL_JTAG_JFIFO_ST_REG | JTAG FIFO status and control registers | 0x0020 | varies |
| USB_SERIAL_JTAG_FRAM_NUM_REG | Last received SOF frame index register | 0x0024 | RO |
| USB_SERIAL_JTAG_IN_EP0_ST_REG | Control IN endpoint status information | 0x0028 | RO |
| USB_SERIAL_JTAG_IN_EP1_ST_REG | CDC-ACM IN endpoint status information | 0x002C | RO |
| USB_SERIAL_JTAG_IN_EP2_ST_REG | CDC-ACM interrupt IN endpoint status information | 0x0030 | RO |
| USB_SERIAL_JTAG_IN_EP3_ST_REG | JTAG IN endpoint status information | 0x0034 | RO |
| USB_SERIAL_JTAG_OUT_EP0_ST_REG | Control OUT endpoint status information | 0x0038 | RO |
| USB_SERIAL_JTAG_OUT_EP1_ST_REG | CDC-ACM OUT endpoint status information | 0x003C | RO |
| USB_SERIAL_JTAG_OUT_EP2_ST_REG | JTAG OUT endpoint status information | 0x0040 | RO |
| **Interrupt Registers** | | | |
| USB_SERIAL_JTAG_INT_RAW_REG | Interrupt raw status register | 0x0008 | R/WTC/SS |
| USB_SERIAL_JTAG_INT_ST_REG | Interrupt status register | 0x000C | RO |
| USB_SERIAL_JTAG_INT_ENA_REG | Interrupt enable status register | 0x0010 | R/W |
| USB_SERIAL_JTAG_INT_CLR_REG | Interrupt clear status register | 0x0014 | WT |
| **Version Registers** | | | |
| USB_SERIAL_JTAG_DATE_REG | Version register | 0x0080 | R/W |

## 30.6 Registers

The addresses in this section are relative to USB Serial/JTAG Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 30.1. USB_SERIAL_JTAG_EP1_REG (0x0000)**

| (reserved) | USB_SERIAL_JTAG_RDWR_BYTE | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**USB_SERIAL_JTAG_RDWR_BYTE** Write and read byte data to/from UART Tx/Rx FIFO through this field. When USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT is set then user can write data (up to 64 bytes) into UART Tx FIFO. When USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT is set, user can check USB_SERIAL_JTAG_OUT_EP1_WR_ADDR and USB_SERIAL_JTAG_OUT_EP1_RD_ADDR to know how many data is received, then read that amount of data from UART Rx FIFO. (R/W)

**Register 30.2. USB_SERIAL_JTAG_CONF0_REG (0x0018)**

| 31 | | | | | | | | | | | | | | | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | USB_SERIAL_JTAG_USB_PAD_ENABLE | USB_SERIAL_JTAG_PULLUP_VALUE | USB_SERIAL_JTAG_DM_PULLDOWN | USB_SERIAL_JTAG_DM_PULLUP | USB_SERIAL_JTAG_DP_PULLDOWN | USB_SERIAL_JTAG_DP_PULLUP | USB_SERIAL_JTAG_PAD_PULL_OVERRIDE | USB_SERIAL_JTAG_VREF_OVERRIDE | USB_SERIAL_JTAG_VREFH | | USB_SERIAL_JTAG_VREFL | USB_SERIAL_JTAG_EXCHG_PINS | USB_SERIAL_JTAG_EXCHG_PINS_OVERRIDE | USB_SERIAL_JTAG_PHY_SEL | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | | 0 | | 0 | 0 | 0 | Reset |

**USB_SERIAL_JTAG_PHY_SEL** Select internal/external PHY. 1'b0: internal PHY, 1'b1: external PHY. (R/W)

**USB_SERIAL_JTAG_EXCHG_PINS_OVERRIDE** Enable software control USB D+ D- exchange. (R/W)

**USB_SERIAL_JTAG_EXCHG_PINS** USB D+ D- exchange (R/W)

**USB_SERIAL_JTAG_VREFL** Control single-end input high threshold. 1.76 V to 2 V, step 80 mV. (R/W)

**USB_SERIAL_JTAG_VREFH** Control single-end input low threshold. 0.8 V to 1.04 V, step 80 mV. (R/W)

**USB_SERIAL_JTAG_VREF_OVERRIDE** Enable software control input threshold. (R/W)

**USB_SERIAL_JTAG_PAD_PULL_OVERRIDE** Enable software control USB D+ D- pull-up pull-down. (R/W)

**USB_SERIAL_JTAG_DP_PULLUP** Control USB D+ pull-up. (R/W)

**USB_SERIAL_JTAG_DP_PULLDOWN** Control USB D+ pull-down. (R/W)

**USB_SERIAL_JTAG_DM_PULLUP** Control USB D- pull-up. (R/W)

**USB_SERIAL_JTAG_DM_PULLDOWN** Control USB D- pull-down. (R/W)

**USB_SERIAL_JTAG_PULLUP_VALUE** Control pull-up value. 0: 2.2 K; 1: 1.1 K. (R/W)

**USB_SERIAL_JTAG_USB_PAD_ENABLE** Enable USB pad function. (R/W)

### Register 30.3. USB_SERIAL_JTAG_TEST_REG (0x001C)

| (reserved) | USB_SERIAL_JTAG_TEST_TX_DM | USB_SERIAL_JTAG_TEST_TX_DP | USB_SERIAL_JTAG_TEST_USB_OE | USB_SERIAL_JTAG_TEST_ENABLE | |
|---|---|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | Reset |

**USB_SERIAL_JTAG_TEST_ENABLE** Enable test of the USB pad. (R/W)

**USB_SERIAL_JTAG_TEST_USB_OE** USB pad output enable in test. (R/W)

**USB_SERIAL_JTAG_TEST_TX_DP** USB D+ tx value in test. (R/W)

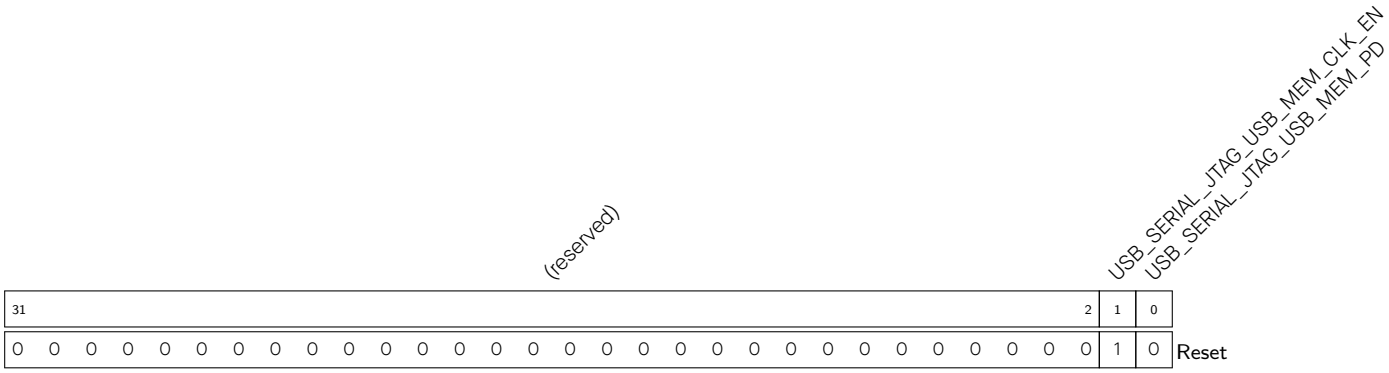
**USB_SERIAL_JTAG_TEST_TX_DM** USB D- tx value in test. (R/W)

### Register 30.4. USB_SERIAL_JTAG_MISC_CONF_REG (0x0044)

| (reserved) | USB_SERIAL_JTAG_CLK_EN | |
|---|---|---|
| 31 … 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 | Reset |

**USB_SERIAL_JTAG_CLK_EN** 1'h1: Force clock on for register. 1'h0: Support clock only when application writes registers. (R/W)

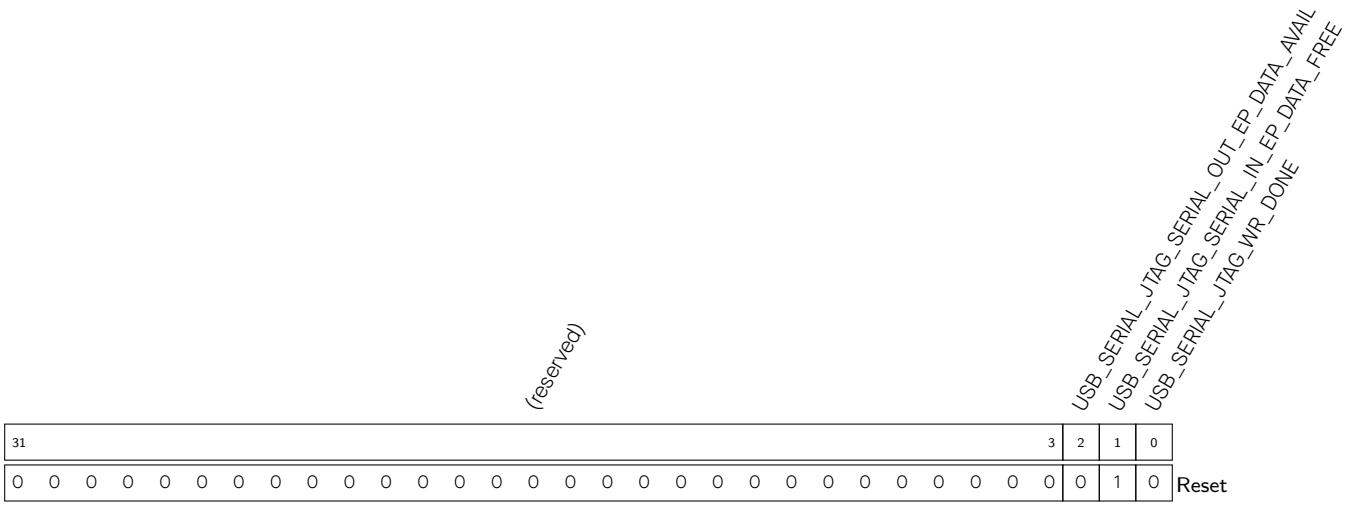
**Register 30.5. USB_SERIAL_JTAG_MEM_CONF_REG (0x0048)**




**USB_SERIAL_JTAG_USB_MEM_PD** Set to power down USB memory. (R/W)

**USB_SERIAL_JTAG_USB_MEM_CLK_EN** Set to force clock-on for USB memory. (R/W)

**Register 30.6. USB_SERIAL_JTAG_EP1_CONF_REG (0x0004)**




**USB_SERIAL_JTAG_WR_DONE** Set this bit to indicate writing byte data to UART Tx FIFO is done. This bit then stays 0 until data in UART Tx FIFO is read by the USB Host. (WT)

**USB_SERIAL_JTAG_SERIAL_IN_EP_DATA_FREE** 1'b1: Indicate UART Tx FIFO is not full and data can be written into in. After writing USB_SERIAL_JTAG_WR_DONE, this will be 1'b0 until the data is sent to the USB Host. (RO)

**USB_SERIAL_JTAG_SERIAL_OUT_EP_DATA_AVAIL** 1'b1: Indicate there is data in UART Rx FIFO. (RO)

**Register 30.7. USB_SERIAL_JTAG_JFIFO_ST_REG (0x0020)**

| (reserved) | USB_SERIAL_JTAG_OUT_FIFO_RESET | USB_SERIAL_JTAG_IN_FIFO_RESET | USB_SERIAL_JTAG_OUT_FIFO_FULL | USB_SERIAL_JTAG_OUT_FIFO_EMPTY | USB_SERIAL_JTAG_OUT_FIFO_CNT | USB_SERIAL_JTAG_IN_FIFO_FULL | USB_SERIAL_JTAG_IN_FIFO_EMPTY | USB_SERIAL_JTAG_IN_FIFO_CNT | |
|---|---|---|---|---|---|---|---|---|---|
| 31 … 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 | 1 | 0 | 0 | 1 | 0 | Reset |

**USB_SERIAL_JTAG_IN_FIFO_CNT** JTAG in FIFO counter. (RO)

**USB_SERIAL_JTAG_IN_FIFO_EMPTY** Set to indicate JTAG in FIFO is empty. (RO)

**USB_SERIAL_JTAG_IN_FIFO_FULL** Set to indicate JTAG in FIFO is full. (RO)

**USB_SERIAL_JTAG_OUT_FIFO_CNT** JTAT out FIFO counter. (RO)

**USB_SERIAL_JTAG_OUT_FIFO_EMPTY** Set to indicate JTAG out FIFO is empty. (RO)

**USB_SERIAL_JTAG_OUT_FIFO_FULL** Set to indicate JTAG out FIFO is full. (RO)

**USB_SERIAL_JTAG_IN_FIFO_RESET** Write 1 to reset JTAG in FIFO. (R/W)

**USB_SERIAL_JTAG_OUT_FIFO_RESET** Write 1 to reset JTAG out FIFO. (R/W)

**Register 30.8. USB_SERIAL_JTAG_FRAM_NUM_REG (0x0024)**

| (reserved) | USB_SERIAL_JTAG_SOF_FRAME_INDEX | |
|---|---|---|
| 31 … 11 | 10 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**USB_SERIAL_JTAG_SOF_FRAME_INDEX** Frame index of received SOF frame. (RO)

### Register 30.9. USB_SERIAL_JTAG_IN_EP0_ST_REG (0x0028)

| (reserved) | USB_SERIAL_JTAG_IN_EP0_RD_ADDR | USB_SERIAL_JTAG_IN_EP0_WR_ADDR | USB_SERIAL_JTAG_IN_EP0_STATE | |
|---|---|---|---|---|
| 31 … 16 | 15 … 9 | 8 … 2 | 1 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 1 | Reset |

**USB_SERIAL_JTAG_IN_EP0_STATE** State of IN Endpoint 0. (RO)

**USB_SERIAL_JTAG_IN_EP0_WR_ADDR** Write data address of IN endpoint 0. (RO)

**USB_SERIAL_JTAG_IN_EP0_RD_ADDR** Read data address of IN endpoint 0. (RO)

### Register 30.10. USB_SERIAL_JTAG_IN_EP1_ST_REG (0x002C)

| (reserved) | USB_SERIAL_JTAG_IN_EP1_RD_ADDR | USB_SERIAL_JTAG_IN_EP1_WR_ADDR | USB_SERIAL_JTAG_IN_EP1_STATE | |
|---|---|---|---|---|
| 31 … 16 | 15 … 9 | 8 … 2 | 1 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 1 | Reset |

**USB_SERIAL_JTAG_IN_EP1_STATE** State of IN Endpoint 1. (RO)

**USB_SERIAL_JTAG_IN_EP1_WR_ADDR** Write data address of IN endpoint 1. (RO)

**USB_SERIAL_JTAG_IN_EP1_RD_ADDR** Read data address of IN endpoint 1. (RO)

**Register 30.11. USB_SERIAL_JTAG_IN_EP2_ST_REG (0x0030)**

| (reserved) | USB_SERIAL_JTAG_IN_EP2_RD_ADDR | USB_SERIAL_JTAG_IN_EP2_WR_ADDR | USB_SERIAL_JTAG_IN_EP2_STATE | |
|---|---|---|---|---|
| 31 … 16 | 15 … 9 | 8 … 2 | 1 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 1 | Reset |

**USB_SERIAL_JTAG_IN_EP2_STATE** State of IN Endpoint 2. (RO)

**USB_SERIAL_JTAG_IN_EP2_WR_ADDR** Write data address of IN endpoint 2. (RO)

**USB_SERIAL_JTAG_IN_EP2_RD_ADDR** Read data address of IN endpoint 2. (RO)

**Register 30.12. USB_SERIAL_JTAG_IN_EP3_ST_REG (0x0034)**

| (reserved) | USB_SERIAL_JTAG_IN_EP3_RD_ADDR | USB_SERIAL_JTAG_IN_EP3_WR_ADDR | USB_SERIAL_JTAG_IN_EP3_STATE | |
|---|---|---|---|---|
| 31 … 16 | 15 … 9 | 8 … 2 | 1 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 1 | Reset |

**USB_SERIAL_JTAG_IN_EP3_STATE** State of IN Endpoint 3. (RO)

**USB_SERIAL_JTAG_IN_EP3_WR_ADDR** Write data address of IN endpoint 3. (RO)

**USB_SERIAL_JTAG_IN_EP3_RD_ADDR** Read data address of IN endpoint 3. (RO)

**Register 30.13. USB_SERIAL_JTAG_OUT_EP0_ST_REG (0x0038)**

| (reserved) | USB_SERIAL_JTAG_OUT_EP0_RD_ADDR | USB_SERIAL_JTAG_OUT_EP0_WR_ADDR | USB_SERIAL_JTAG_OUT_EP0_STATE | |
|---|---|---|---|---|
| 31 … 16 | 15 … 9 | 8 … 2 | 1 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | Reset |

**USB_SERIAL_JTAG_OUT_EP0_STATE** State of OUT Endpoint 0. (RO)

**USB_SERIAL_JTAG_OUT_EP0_WR_ADDR** Write data address of OUT Endpoint 0. When USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT is detected, there are USB_SERIAL_JTAG_OUT_EP0_WR_ADDR - 2 bytes of data in OUT EP0. (RO)

**USB_SERIAL_JTAG_OUT_EP0_RD_ADDR** Read data address of OUT endpoint 0. (RO)

**Register 30.14. USB_SERIAL_JTAG_OUT_EP1_ST_REG (0x003C)**

| (reserved) | USB_SERIAL_JTAG_OUT_EP1_REC_DATA_CNT | USB_SERIAL_JTAG_OUT_EP1_RD_ADDR | USB_SERIAL_JTAG_OUT_EP1_WR_ADDR | USB_SERIAL_JTAG_OUT_EP1_STATE | |
|---|---|---|---|---|---|
| 31 … 23 | 22 … 16 | 15 … 9 | 8 … 2 | 1 … 0 | |
| 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | Reset |

**USB_SERIAL_JTAG_OUT_EP1_STATE** State of OUT Endpoint 1. (RO)

**USB_SERIAL_JTAG_OUT_EP1_WR_ADDR** Write data address of OUT Endpoint 1. When USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT is detected, there are USB_SERIAL_JTAG_OUT_EP1_WR_ADDR - 2 bytes of data in OUT EP1. (RO)

**USB_SERIAL_JTAG_OUT_EP1_RD_ADDR** Read data address of OUT endpoint 1. (RO)

**USB_SERIAL_JTAG_OUT_EP1_REC_DATA_CNT** Data count in OUT Endpoint 1 when one packet is received. (RO)

**Register 30.15. USB_SERIAL_JTAG_OUT_EP2_ST_REG (0x0040)**

| (reserved) | USB_SERIAL_JTAG_OUT_EP2_RD_ADDR | USB_SERIAL_JTAG_OUT_EP2_WR_ADDR | USB_SERIAL_JTAG_OUT_EP2_STATE | |
|---|---|---|---|---|
| 31 16 | 15 9 | 8 2 | 1 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | Reset |

**USB_SERIAL_JTAG_OUT_EP2_STATE** State of OUT Endpoint 2. (RO)

**USB_SERIAL_JTAG_OUT_EP2_WR_ADDR** Write data address of OUT endpoint 2. When USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT is detected, there are USB_SERIAL_JTAG_OUT_EP2_WR_ADDR - 2 bytes of data in OUT EP2. (RO)

**USB_SERIAL_JTAG_OUT_EP2_RD_ADDR** Read data address of OUT endpoint 2. (RO)

**Register 30.16. USB_SERIAL_JTAG_INT_RAW_REG (0x0008)**

| 31 | | | | | | | | | | | | | | | | | | | | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_RAW | USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_RAW | USB_SERIAL_JTAG_USB_BUS_RESET_INT_RAW | USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_RAW | USB_SERIAL_JTAG_STUFF_ERR_INT_RAW | USB_SERIAL_JTAG_CRC16_ERR_INT_RAW | USB_SERIAL_JTAG_CRC5_ERR_INT_RAW | USB_SERIAL_JTAG_PID_ERR_INT_RAW | USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_RAW | USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_RAW | USB_SERIAL_JTAG_SOF_INT_RAW | USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_RAW | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | Reset |

**USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_RAW** The raw interrupt bit turns to high level when a flush command is received for IN endpoint 2 of JTAG. (R/WTC/SS)

**USB_SERIAL_JTAG_SOF_INT_RAW** The raw interrupt bit turns to high level when a SOF frame is received. (R/WTC/SS)

**USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_RAW** The raw interrupt bit turns to high level when the Serial Port OUT Endpoint received one packet. (R/WTC/SS)

**USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_RAW** The raw interrupt bit turns to high level when the Serial Port IN Endpoint is empty. (R/WTC/SS)

**USB_SERIAL_JTAG_PID_ERR_INT_RAW** The raw interrupt bit turns to high level when a PID error is detected. (R/WTC/SS)

**USB_SERIAL_JTAG_CRC5_ERR_INT_RAW** The raw interrupt bit turns to high level when a CRC5 error is detected. (R/WTC/SS)

**USB_SERIAL_JTAG_CRC16_ERR_INT_RAW** The raw interrupt bit turns to high level when a CRC16 error is detected. (R/WTC/SS)

**USB_SERIAL_JTAG_STUFF_ERR_INT_RAW** The raw interrupt bit turns to high level when a bit stuffing error is detected. (R/WTC/SS)

**USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_RAW** The raw interrupt bit turns to high level when an IN token for IN endpoint 1 is received. (R/WTC/SS)

**USB_SERIAL_JTAG_USB_BUS_RESET_INT_RAW** The raw interrupt bit turns to high level when a USB bus reset is detected. (R/WTC/SS)

**USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_RAW** The raw interrupt bit turns to high level when OUT endpoint 1 received packet with zero payload. (R/WTC/SS)

**USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_RAW** The raw interrupt bit turns to high level when OUT endpoint 2 received packet with zero payload. (R/WTC/SS)

**Register 30.17. USB_SERIAL_JTAG_INT_ST_REG (0x000C)**

| 31 | | | | | | | | | | | | | | | | | | | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_ST | USB_SERIAL_JTAG_SOF_INT_ST | USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_ST | USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_ST | USB_SERIAL_JTAG_PID_ERR_INT_ST | USB_SERIAL_JTAG_CRC5_ERR_INT_ST | USB_SERIAL_JTAG_CRC16_ERR_INT_ST | USB_SERIAL_JTAG_STUFF_ERR_INT_ST | USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_ST | USB_SERIAL_JTAG_USB_BUS_RESET_INT_ST | USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_ST | USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_ST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT interrupt. (RO)

**USB_SERIAL_JTAG_SOF_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_SOF_INT interrupt. (RO)

**USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT interrupt. (RO)

**USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT interrupt. (RO)

**USB_SERIAL_JTAG_PID_ERR_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_PID_ERR_INT interrupt. (RO)

**USB_SERIAL_JTAG_CRC5_ERR_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_CRC5_ERR_INT interrupt. (RO)

**USB_SERIAL_JTAG_CRC16_ERR_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_CRC16_ERR_INT interrupt. (RO)

**USB_SERIAL_JTAG_STUFF_ERR_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_STUFF_ERR_INT interrupt. (RO)

**USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT interrupt. (RO)

**USB_SERIAL_JTAG_USB_BUS_RESET_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_USB_BUS_RESET_INT interrupt. (RO)

**USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT interrupt. (RO)

**USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_ST** The raw interrupt status bit for the USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT interrupt. (RO)

**Register 30.18. USB_SERIAL_JTAG_INT_ENA_REG (0x0010)**

| 31 | | | | | | | | | | | | | | | | | | | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_ENA | USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_ENA | USB_SERIAL_JTAG_USB_BUS_RESET_INT_ENA | USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_ENA | USB_SERIAL_JTAG_STUFF_ERR_INT_ENA | USB_SERIAL_JTAG_CRC16_ERR_INT_ENA | USB_SERIAL_JTAG_CRC5_ERR_INT_ENA | USB_SERIAL_JTAG_PID_ERR_INT_ENA | USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_ENA | USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_ENA | USB_SERIAL_JTAG_SOF_INT_ENA | USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_ENA | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT interrupt. (R/W)

**USB_SERIAL_JTAG_SOF_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_SOF_INT interrupt. (R/W)

**USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT interrupt. (R/W)

**USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT interrupt. (R/W)

**USB_SERIAL_JTAG_PID_ERR_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_PID_ERR_INT interrupt. (R/W)

**USB_SERIAL_JTAG_CRC5_ERR_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_CRC5_ERR_INT interrupt. (R/W)

**USB_SERIAL_JTAG_CRC16_ERR_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_CRC16_ERR_INT interrupt. (R/W)

**USB_SERIAL_JTAG_STUFF_ERR_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_STUFF_ERR_INT interrupt. (R/W)

**USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT interrupt. (R/W)

**USB_SERIAL_JTAG_USB_BUS_RESET_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_USB_BUS_RESET_INT interrupt. (R/W)

**USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT interrupt. (R/W)

**USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_ENA** The interrupt enable bit for the USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT interrupt. (R/W)

**Register 30.19. USB_SERIAL_JTAG_INT_CLR_REG (0x0014)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_CLR | USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_CLR | USB_SERIAL_JTAG_USB_BUS_RESET_INT_CLR | USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_CLR | USB_SERIAL_JTAG_STUFF_ERR_INT_CLR | USB_SERIAL_JTAG_CRC16_ERR_INT_CLR | USB_SERIAL_JTAG_CRC5_ERR_INT_CLR | USB_SERIAL_JTAG_PID_ERR_INT_CLR | USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_CLR | USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_CLR | USB_SERIAL_JTAG_SOF_INT_CLR | USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_CLR | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_JTAG_IN_FLUSH_INT interrupt. (WT)

**USB_SERIAL_JTAG_SOF_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_JTAG_SOF_INT interrupt. (WT)

**USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_SERIAL_OUT_RECV_PKT_INT interrupt. (WT)

**USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_SERIAL_IN_EMPTY_INT interrupt. (WT)

**USB_SERIAL_JTAG_PID_ERR_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_PID_ERR_INT interrupt. (WT)

**USB_SERIAL_JTAG_CRC5_ERR_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_CRC5_ERR_INT interrupt. (WT)

**USB_SERIAL_JTAG_CRC16_ERR_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_CRC16_ERR_INT interrupt. (WT)

**USB_SERIAL_JTAG_STUFF_ERR_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_STUFF_ERR_INT interrupt. (WT)

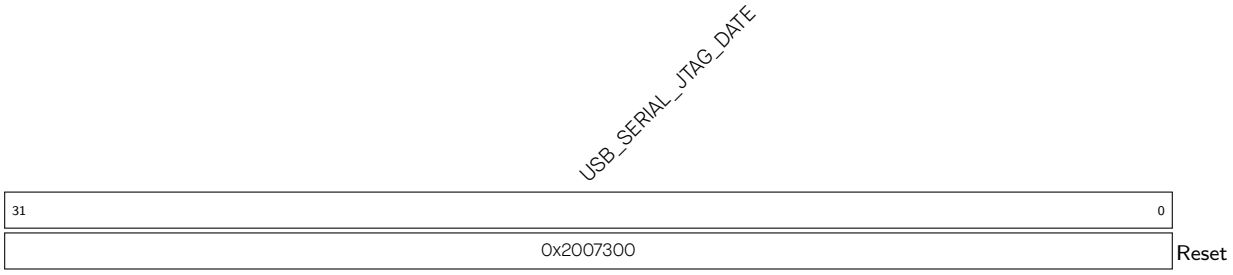
**USB_SERIAL_JTAG_IN_TOKEN_REC_IN_EP1_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_IN_TOKEN_IN_EP1_INT interrupt. (WT)

**USB_SERIAL_JTAG_USB_BUS_RESET_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_USB_BUS_RESET_INT interrupt. (WT)

**USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_OUT_EP1_ZERO_PAYLOAD_INT interrupt. (WT)

**USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT_CLR** Set this bit to clear the USB_SERIAL_JTAG_OUT_EP2_ZERO_PAYLOAD_INT interrupt. (WT)

**Register 30.20. USB_SERIAL_JTAG_DATE_REG (0x0080)**




**USB_SERIAL_JTAG_DATE** Version control register. (R/W)

# Chapter 31

# Two-wire Automotive Interface (TWAI)

The Two-wire Automotive Interface (TWAI®) is a multi-master, multi-cast communication protocol with functions such as error detection and signaling and inbuilt message priorities and arbitration. The TWAI protocol is suited for automotive and industrial applications (see Section 31.2 for more details).

ESP32-C3 contains a TWAI controller that can be connected to the TWAI bus via an external transceiver. The TWAI controller contains numerous advanced features, and can be utilized in a wide range of use cases such as automotive products, industrial automation controls, building automation, etc.

## 31.1 Features

The TWAI controller on ESP32-C3 supports the following features:

- Compatible with ISO 11898-1 protocol (CAN Specification 2.0)
- Supports Standard Frame Format (11-bit ID) and Extended Frame Format (29-bit ID)
- Bit rates from 1 Kbit/s to 1 Mbit/s
- Multiple modes of operation
  - Normal
  - Listen-only (no influence on bus)
  - Self-test (no acknowledgment required during data transmission)
- 64-byte Receive FIFO
- Special transmissions
  - Single-shot transmissions (does not automatically re-transmit upon error)
  - Self Reception (the TWAI controller transmits and receives messages simultaneously)
- Acceptance Filter (supports single and dual filter modes)
- Error detection and handling
  - Error Counters
  - Configurable Error Warning Limit
  - Error Code Capture
  - Arbitration Lost Capture

## 31.2 Functional Protocol

### 31.2.1 TWAI Properties

The TWAI protocol connects two or more nodes in a bus network, and allows nodes to exchange messages in a latency bounded manner. A TWAI bus has the following properties.

**Single Channel and Non-Return-to-Zero:** The bus consists of a single channel to carry bits, and thus communication is half-duplex. Synchronization is also implemented in this channel, so extra channels (e.g., clock or enable) are not required. The bit stream of a TWAI message is encoded using the Non-Return-to-Zero (NRZ) method.

**Bit Values:** The single channel can either be in a dominant or recessive state, representing a logical 0 and a logical 1 respectively. A node transmitting data in a dominant state always overrides the other node transmitting data in a recessive state. The physical implementation on the bus is left to the application level to decide (e.g., differential pair or a single wire).

**Bit Stuffing:** Certain fields of TWAI messages are bit-stuffed. A transmitter that transmits five consecutive bits of the same value (e.g., dominant value or recessive value) should automatically insert a complementary bit. Likewise, a receiver that receives five consecutive bits should treat the next bit as a stuffed bit. Bit stuffing is applied to the following fields: SOF, arbitration field, control field, data field, and CRC sequence (see Section 31.2.2 for more details).

**Multi-cast:** All nodes receive the same bits as they are connected to the same bus. Data is consistent across all nodes unless there is a bus error (see Section 31.2.3 for more details).

**Multi-master:** Any node can initiate a transmission. If a transmission is already ongoing, a node will wait until the current transmission is over before initiating a new transmission.

**Message Priority and Arbitration:** If two or more nodes simultaneously initiate a transmission, the TWAI protocol ensures that one node will win arbitration of the bus. The arbitration field of the message transmitted by each node is used to determine which node will win arbitration.

**Error Detection and Signaling:** Each node actively monitors the bus for errors, and signals the detected errors by transmitting an error frame.

**Fault Confinement:** Each node maintains a set of error counters that are incremented/decremented according to a set of rules. When the error counters surpass a certain threshold, the node will automatically eliminate itself from the network by switching itself off.

**Configurable Bit Rate:** The bit rate for a single TWAI bus is configurable. However, all nodes on the same bus must operate at the same bit rate.

**Transmitters and Receivers:** At any point in time, a TWAI node can either be a transmitter or a receiver.

- A node generating a message is a transmitter. The node remains a transmitter until the bus is idle or until the node loses arbitration. Please note that nodes that have not lost arbitration can all be transmitters.
- All nodes that are not transmitters are receivers.

### 31.2.2 TWAI Messages

TWAI nodes use messages to transmit data, and signal errors to other nodes when detecting errors on the bus. Messages are split into various frame types, and some frame types will have different frame formats.

The TWAI protocol has of the following frame types:

- Data frame
- Remote frame
- Error frame
- Overload frame
- Interframe space

The TWAI protocol has the following frame formats:

- Standard Frame Format (SFF) that uses a 11-bit identifier
- Extended Frame Format (EFF) that uses a 29-bit identifier

#### 31.2.2.1 Data Frames and Remote Frames

Data frames are used by nodes to send data to other nodes, and can have a payload of 0 to 8 data bytes. Remote frames are used for nodes to request a data frame with the same identifier from other nodes, and thus they do not contain any data bytes. However, data frames and remote frames share many fields. Figure 31.2-1 illustrates the fields and sub-fields of different frames and formats.

**Arbitration Field**

When two or more nodes transmits a data or remote frame simultaneously, the arbitration field is used to determine which node will win arbitration of the bus. In the arbitration field, if a node transmits a recessive bit while detects a dominant bit, this indicates that another node has overridden its recessive bit. Therefore, the node transmitting the recessive bit has lost arbitration of the bus and should immediately switch to be a receiver.

The arbitration field primarily consists of a frame identifier that is transmitted from the most significant bit first. Given that a dominant bit represents a logical 0, and a recessive bit represents a logical 1:

- A frame with the smallest ID value always wins arbitration.
- Given the same ID and format, data frames always prevail over remote frames due to their RTR bits being dominant.
- Given the same first 11 bits of ID, a Standard Format Data Frame always prevails over an Extended Format Data Frame due to its SRR bits being recessive.

**Control Field**

The control field primarily consists of the DLC (Data Length Code) which indicates the number of payload data bytes for a data frame, or the number of requested data bytes for a remote frame. The DLC is transmitted from the most significant bit first.

**Data Field**

The data field contains the actual payload data bytes of a data frame. Remote frames do not contain any data field.

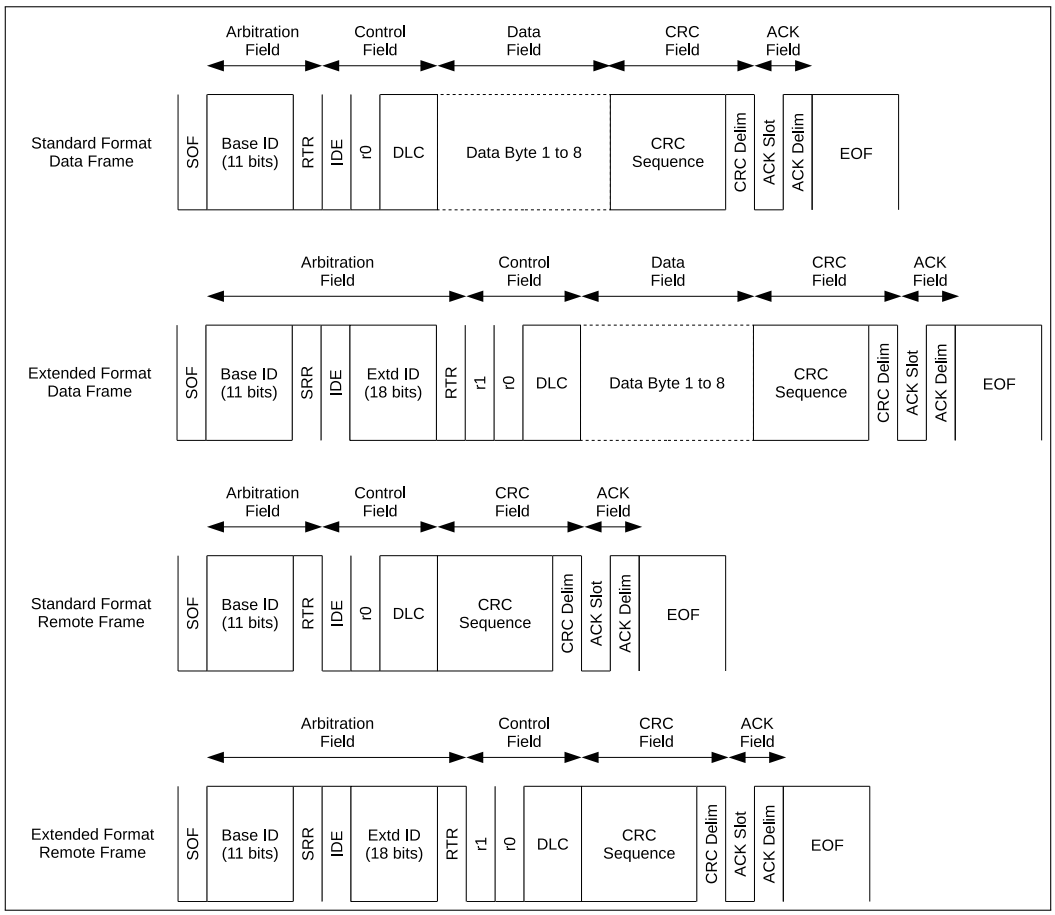



Figure 31.2-1. Bit Fields in Data Frames and Remote Frames

### CRC Field

The CRC field primarily consists of a CRC sequence. The CRC sequence is a 15-bit cyclic redundancy code calculated form the de-stuffed contents (everything from the SOF to the end of the data field) of a data or remote frame.

### ACK Field

The ACK field primarily consists of an ACK Slot and an ACK Delim. The ACK field indicates that the receiver has received an effective message from the transmitter.

Table 31.2-1. Data Frames and Remote Frames in SFF and EFF

| Data/Remote Frames | Description |
|---|---|
| SOF | The SOF (Start of Frame) is a single dominant bit used to synchronize nodes on the bus. |
| Base ID | The Base ID (ID.28 to ID.18) is the 11-bit identifier for SFF, or the first 11 bits of the 29-bit identifier for EFF. |
| RTR | The RTR (Remote Transmission Request) bit indicates whether the message is a data frame (dominant) or a remote frame (recessive). This means that a remote frame will always lose arbitration to a data frame if they have the same ID. |

Cont'd on next page

Table 31.2-1 – cont'd from previous page

| Data/Remote Frames | Description |
|---|---|
| SRR | The SRR (Substitute Remote Request) bit is transmitted in EFF to substitute for the RTR bit at the same position in SFF. |
| IDE | The IDE (Identifier Extension) bit indicates whether the message is SFF (dominant) or EFF (recessive). This means that a SFF frame will always win arbitration over an EFF frame if they have the same Base ID. |
| Extd ID | The Extended ID (ID.17 to ID.0) is the remaining 18 bits of the 29-bit identifier for EFF. |
| r1 | The r1 bit (reserved bit 1) is always dominant. |
| r0 | The r0 bit (reserved bit 0) is always dominant. |
| DLC | The DLC (Data Length Code) is 4-bit long and should contain any value from 0 to 8. Data frames use the DLC to indicate the number of data bytes in the data frame. Remote frames used the DLC to indicate the number of data bytes to request from another node. |
| Data Bytes | The data payload of data frames. The number of bytes should match the value of DLC. Data byte 0 is transmitted first, and each data byte is transmitted from the most significant bit first. |
| CRC Sequence | The CRC sequence is a 15-bit cyclic redundancy code. |
| CRC Delim | The CRC Delim (CRC Delimiter) is a single recessive bit that follows the CRC sequence. |
| ACK Slot | The ACK Slot (Acknowledgment Slot) is intended for receiver nodes to indicate that the data or remote frame was received without any issue. The transmitter node will send a recessive bit in the ACK Slot and receiver nodes should override the ACK Slot with a dominant bit if the frame was received without errors. |
| ACK Delim | The ACK Delim (Acknowledgment Delimiter) is a single recessive bit. |
| EOF | The EOF (End of Frame) marks the end of a data or remote frame, and consists of seven recessive bits. |

#### 31.2.2.2 Error and Overload Frames

**Error Frames**

Error frames are transmitted when a node detects a bus error. Error frames notably consist of an Error Flag which is made up of six consecutive bits of the same value, thus violating the bit-stuffing rule. Therefore, when a particular node detects a bus error and transmits an error frame, all other nodes will then detect a stuff error and transmit their own error frames in response. This has the effect of propagating the detection of a bus error across all nodes on the bus.

When a node detects a bus error, it will transmit an error frame starting from the next bit. However, if the type of bus error was a CRC error, then the error frame will start at the bit following the ACK Delim (see Section 31.2.3 for more details). The following Figure 31.2-2 shows different fields of an error frame:

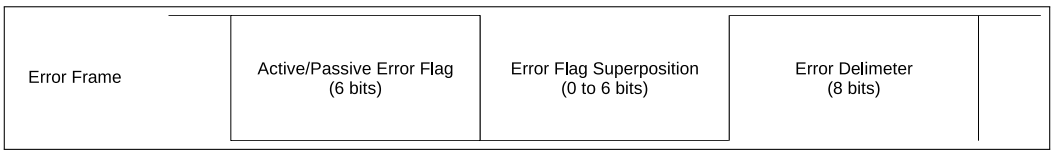



Figure 31.2-2. Fields of an Error Frame

Table 31.2-2. Error Frame

| Error Frame | Description |
|---|---|
| Error Flag | The Error Flag has two forms, the Active Error Flag consisting of 6 dominant bits and the Passive Error Flag consisting of 6 recessive bits (unless overridden by dominant bits of other nodes). Active Error Flags are sent by error active nodes, whilst Passive Error Flags are sent by error passive nodes. |
| Error Flag Superposition | The Error Flag Superposition field meant to allow for other nodes on the bus to transmit their respective Active Error Flags. The superposition field can range from 0 to 6 bits, and ends when the first recessive bit is detected (i.e., the first it of the Delimiter). |
| Error Delimeter | The Delimiter field marks the end of the error/overload frame, and consists of 8 recessive bits. |

### Overload Frames

An overload frame has the same bit fields as an error frame containing an Active Error Flag. The key difference is in the cases that can trigger the transmission of an overload frame. Figure 31.2-3 below shows the bit fields of an overload frame.

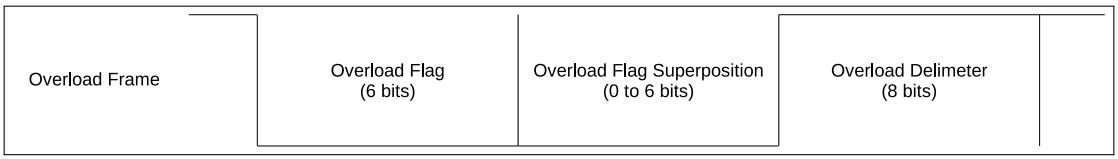



Figure 31.2-3. Fields of an Overload Frame

Table 31.2-3. Overload Frame

| Overload Flag | Description |
|---|---|
| Overload Flag | Consists of 6 dominant bits. Same as an Active Error Flag. |
| Overload Flag Superposition | Allows for the superposition of Overload Flags from other nodes, similar to an Error Flag Superposition. |
| Overload Delimiter | Consists of 8 recessive bits. Same as an Error Delimiter. |

Overload frames will be transmitted under the following cases:

1. A receiver requires a delay of the next data or remote frame.
2. A dominant bit is detected at the first and second bit of intermission.

3. A dominant bit is detected at the eighth (last) bit of an Error Delimiter. Note that in this case, TEC and REC will not be incremented (see Section 31.2.3 for more details).

Transmitting an overload frame due to one of the above cases must also satisfy the following rules:

- The start of an overload frame due to case 1 is only allowed to be started at the first bit time of an expected intermission.
- The start of an overload frame due to case 2 and 3 is only allowed to be started one bit after detecting the dominant bit.
- A maximum of two overload frames may be generated in order to delay the transmission of the next data or remote frame.

#### 31.2.2.3 Interframe Space

The Interframe Space acts as a separator between frames. Data frames and remote frames must be separated from preceding frames by an Interframe Space, regardless of the preceding frame's type (data frame, remote frame, error frame, or overload frame). However, error frames and overload frames do not need to be separated from preceding frames.

Figure 31.2-4 shows the fields within an Interframe Space:

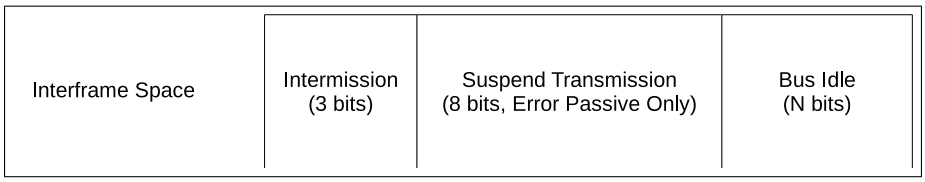



Figure 31.2-4. The Fields within an Interframe Space

Table 31.2-4. Interframe Space

| Interframe Space | Description |
|---|---|
| Intermission | The Intermission consists of 3 recessive bits. |
| Suspend Transmission | An Error Passive node that has just transmitted a message must include a Suspend Transmission field. This field consists of 8 recessive bits. Error Active nodes should not include this field. |
| Bus Idle | The Bus Idle field is of arbitrary length. Bus Idle ends when an SOF is transmitted. If a node has a pending transmission, the SOF should be transmitted at the first bit following Intermission. |

### 31.2.3 TWAI Errors

#### 31.2.3.1 Error Types

Bus Errors in TWAI are categorized into the following types:

**Bit Error**

A Bit Error occurs when a node transmits a bit value (i.e., dominant or recessive) but the opposite bit is detected (e.g., a dominant bit is transmitted but a recessive is detected). However, if the transmitted bit is

recessive and is located in the Arbitration Field or ACK Slot or Passive Error Flag, then detecting a dominant bit will not be considered a Bit Error.

**Stuff Error**

A stuff error is detected when six consecutive bits of the same value are detected (which violats the bit-stuffing encoding rules).

**CRC Error**

A receiver of a data or remote frame will calculate CRC based on the bits it has received. A CRC error occurs when the CRC calculated by the receiver does not match the CRC sequence in the received data or remote Frame.

**Format Error**

A Format Error is detected when a format-fixed bit field of a message contains an illegal bit. For example, the r1 and r0 fields must be dominant.

**ACK Error**

An ACK Error occurs when a transmitter does not detect a dominant bit at the ACK Slot.

#### 31.2.3.2 Error States

TWAI nodes implement fault confinement by each maintaining two error counters, where the counter values determine the error state. The two error counters are known as the Transmit Error Counter (TEC) and Receive Error Counter (REC). TWAI has the following error states.

**Error Active**

An Error Active node is able to participate in bus communication and transmit an Active Error Flag when it detects an error.

**Error Passive**

An Error Passive node is able to participate in bus communication, but can only transmit an Passive Error Flag when it detects an error. Error Passive nodes that have transmitted a data or remote frame must also include the Suspend Transmission field in the subsequent Interframe Space.

**Bus Off**

A Bus Off node is not permitted to influence the bus in any way (i.e., is not allowed to transmit data).

#### 31.2.3.3 Error Counters

The TEC and REC are incremented/decremented according to the following rules. **Note that more than one rule can apply to a given message transfer.**

1. When a receiver detects an error, the REC is increased by 1, except when the detected error was a Bit Error during the transmission of an Active Error Flag or an Overload Flag.
2. When a receiver detects a dominant bit as the first bit after sending an Error Flag, the REC is increased by 8.
3. When a transmitter sends an Error Flag, the TEC is increased by 8. However, the following scenarios are exempt from this rule:
    - A transmitter is Error Passive since the transmitter generates an Acknowledgment Error because of not detecting a dominant bit in the ACK Slot, while detecting a dominant bit when sending a passive error flag. In this case, the TEC should not be increased.

- A transmitter transmits an Error Flag due to a Stuff Error during Arbitration. If the stuffed bit should have been recessive but was monitored as dominant, then the TEC should not be increased.

4. If a transmitter detects a Bit Error whilst sending an Active Error Flag or Overload Flag, the TEC is increased by 8.
5. If a receiver detects a Bit Error while sending an Active Error Flag or Overload Flag, the REC is increased by 8.
6. A node can tolerate up to 7 consecutive dominant bits after sending an Active/Passive Error Flag, or Overload Flag. After detecting the 14th consecutive dominant bit (when sending an Active Error Flag or Overload Flag), or the 8th consecutive dominant bit following a Passive Error Flag, a transmitter will increase its TEC by 8 and a receiver will increase its REC by 8. Every additional 8 consecutive dominant bits will also increase the TEC (for transmitters) or REC (for receivers) by 8 as well.
7. When a transmitter has transmitted a message (getting ACK and no errors until the EOF is complete), the TEC is decremented by 1, unless the TEC is already at 0.
8. When a receiver successfully receives a message (no errors before ACK Slot, and successful sending of ACK), the REC is decremented.
   - If the REC is between 1 and 127, the REC will be decremented by 1.
   - If the REC is greater than 127, the REC will be set to 127.
   - If the REC is 0, the REC will remain 0.
9. A node becomes Error Passive when its TEC and/or REC is greater than or equal to 128. Though the node becomes Error Passive, it still sends an Active Error Flag. Note that once the REC has reached to 128, any further increases to its value are invalid until the REC returns to a value less than 128.
10. A node becomes Bus Off when its TEC is greater than or equal to 256.
11. An Error Passive node becomes Error Active when both the TEC and REC are less than or equal to 127.
12. A Bus Off node can become Error Active (with both its TEC and REC reset to 0) after it monitors 128 occurrences of 11 consecutive recessive bits on the bus.

## 31.2.4 TWAI Bit Timing

### 31.2.4.1 Nominal Bit

The TWAI protocol allows a TWAI bus to operate at a particular bit rate. However, all nodes within a TWAI bus must operate at the same bit rate.

- **The Nominal Bit Rate** is defined as the number of bits transmitted per second.
- **The Nominal Bit Time** is defined as **1/Nominal Bit Rate**.

A single Nominal Bit Time is divided into multiple segments, and each segment is made up of multiple Time Quanta. A **Time Quantum** is a minimum unit of time, and is implemented as some form of prescaled clock signal in each node. Figure 31.2-5 illustrates the segments within a single Nominal Bit Time.

TWAI controllers will operate in time steps of one Time Quanta where the state of the TWAI bus is analyzed. If the bus states in two consecutive Time Quantas are different (i.e., recessive to dominant or vice versa), it

means an edge is generated. The intersection of PBS1 and PBS2 is considered the Sample Point and the sampled bus value is considered the value of that bit.

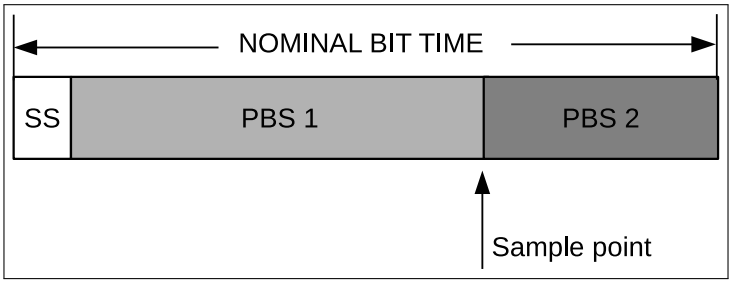



Figure 31.2-5. Layout of a Bit

Table 31.2-5. Segments of a Nominal Bit Time

| Segment | Description |
|---|---|
| SS | The SS (Synchronization Segment) is 1 Time Quantum long. If all nodes are perfectly synchronized, the edge of a bit will lie in the SS. |
| PBS1 | PBS1 (Phase Buffer Segment 1) can be 1 to 16 Time Quanta long. PBS1 is meant to compensate for the physical delay times within the network. PBS1 can also be lengthened for synchronization purposes. |
| PBS2 | PBS2 (Phase Buffer Segment 2) can be 1 to 8 Time Quanta long. PBS2 is meant to compensate for the information processing time of nodes. PBS2 can also be shortened for synchronization purposes. |

#### 31.2.4.2 Hard Synchronization and Resynchronization

Due to clock skew and jitter, the bit timing of nodes on the same bus may become out of phase. Therefore, a bit edge may come before or after the SS. To ensure that the internal bit timing clocks of each node are kept in phase, TWAI has various methods of synchronization. The **Phase Error "e"** is measured in the number of Time Quanta and relative to the SS.

- A positive Phase Error (e > 0) is when the edge lies after the SS and before the Sample Point (i.e., the edge is late).
- A negative Phase Error (e < 0) is when the edge lies after the Sample Point of the previous bit and before SS (i.e., the edge is early).

To correct for Phase Errors, there are two forms of synchronization, known as **Hard Synchronization** and **Resynchronization**. **Hard Synchronization** and **Resynchronization** obey the following rules:

- Only one synchronization may occur in a single bit time.
- Synchronizations only occurs on recessive to dominant edges.

**Hard Synchronization**

Hard Synchronization occurs on the recessive to dominant (i.e., the first SOF bit after Bus Idle) edges when the bus is idle. All nodes will restart their internal bit timings so that the recessive to dominant edge lies within the SS of the restarted bit timing.

**Resynchronization**

Resynchronization occurs on recessive to dominant edges when the bus is not idel. If the edge has a positive Phase Error (e > 0), PBS1 is lengthened by a certain number of Time Quanta. If the edge has a negative Phase Error (e < 0), PBS2 will be shortened by a certain number of Time Quanta.

The number of Time Quanta to lengthen or shorten depends on the magnitude of the Phase Error, and is also limited by the Synchronization Jump Width (SJW) value which is programmable.

- When the magnitude of the Phase Error (**e**) is less than or equal to the SJW, PBS1/PBS2 are lengthened/shortened by the **e** number of Time Quanta. This has a same effect as Hard Synchronization.
- When the magnitude of the Phase Error is greater to the SJW, PBS1/PBS2 are lengthened/shortened by the SJW number of Time Quanta. This means it may take multiple bits of synchronization before the Phase Error is entirely corrected.

## 31.3 Architectural Overview

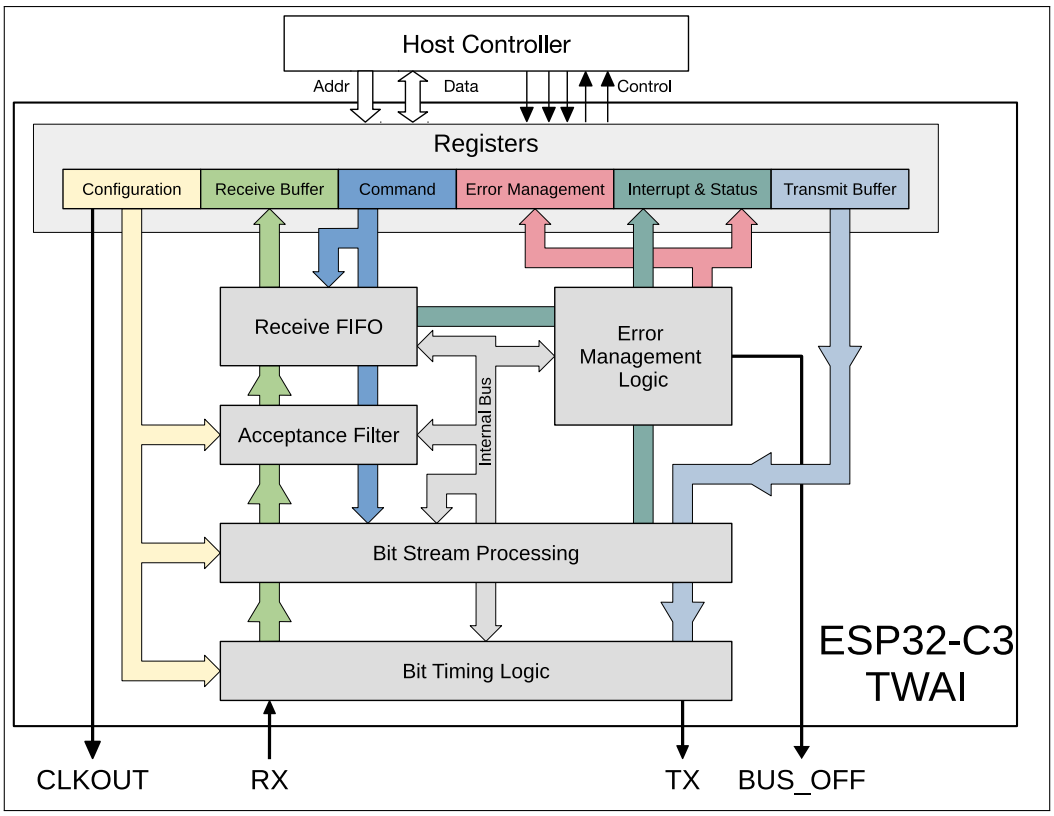



Figure 31.3-1. TWAI Overview Diagram

The major functional blocks of the TWAI controller are shown in Figure 31.3-1.

### 31.3.1 Registers Block

The ESP32-C3 CPU accesses peripherals using 32-bit aligned words. However, the majority of registers in the TWAI controller only contain useful data at the least significant byte (bits [7:0]). Therefore, in these registers, bits [31:8] are ignored on writes, and return 0 on reads.

**Configuration Registers**

The configuration registers store various configuration items for the TWAI controller such as bit rates, operation mode, Acceptance Filter, etc. Configuration registers can only be modified whilst the TWAI controller is in Reset Mode (See Section 31.4.1).

**Command Registers**

The command register is used by the CPU to drive the TWAI controller to initiate certain actions such as transmitting a message or clearing the Receive Buffer. The command register can only be modified when the TWAI controller is in Operation Mode (see section 31.4.1).

**Interrupt & Status Registers**

The interrupt register indicates what events have occurred in the TWAI controller (each event is represented by a separate bit). The status register indicates the current status of the TWAI controller.

**Error Management Registers**

The error management registers include error counters and capture registers. The error counter registers represent TEC and REC values. The capture registers will record information about instances where TWAI controller detects a bus error, or when it loses arbitration.

**Transmit Buffer Registers**

The transmit buffer is a 13-byte buffer used to store a TWAI message to be transmitted.

**Receive Buffer Registers**

The Receive Buffer is a 13-byte buffer which stores a single message. The Receive Buffer acts as a window of Receive FIFO, whose first message will be mapped into the Receive Buffer.

Note that the Transmit Buffer registers, Receive Buffer registers, and the Acceptance Filter registers share the same address range (offset 0x0040 to 0x0070). Their access is governed by the following rules:

- When the TWAI controller is in Reset Mode, all reads and writes to the address range maps to the Acceptance Filter registers.
- When the TWAI controller is in Operation Mode:
  - All reads to the address range maps to the Receive Buffer registers.
  - All writes to the address range maps to the Transmit Buffer registers.

### 31.3.2 Bit Stream Processor

The Bit Stream Processing (BSP) module frames data from the Transmit Buffer (e.g. bit stuffing and additional CRC fields) and generates a bit stream for the Bit Timing Logic (BTL) module. At the same time, the BSP module is also responsible for processing the received bit stream (e.g., de-stuffing and verifying CRC) from the BTL module and placing the message into the Receive FIFO. The BSP will also detect errors on the TWAI bus and report them to the Error Management Logic (EML).

### 31.3.3 Error Management Logic

The Error Management Logic (EML) module updates the TEC and REC, records error information like error types and positions, and updates the error state of the TWAI controller such that the BSP module generates the correct Error Flags. Furthermore, this module also records the bit position when the TWAI controller loses arbitration.

### 31.3.4 Bit Timing Logic

The Bit Timing Logic (BTL) module transmits and receives messages at the configured bit rate. The BTL module also handles bit timing synchronization so that communication remains stable. A single bit time consists of multiple programmable segments that allows users to set the length of each segment to account for factors such as propagation delay and controller processing time, etc.

### 31.3.5 Acceptance Filter

The Acceptance Filter is a programmable message filtering unit that allows the TWAI controller to accept or reject a received message based on the message's ID field. Only accepted messages will be stored in the Receive FIFO. The Acceptance Filter's registers can be programmed to specify a single filter, or two separate filters (dual filter mode).

### 31.3.6 Receive FIFO

The Receive FIFO is a 64-byte buffer (inside the TWAI controller) that stores received messages accepted by the Acceptance Filter. Messages in the Receive FIFO can vary in size (between 3 to 13-bytes). When the Receive FIFO is full (or does not have enough space to store the next received message in its entirety), the Overrun Interrupt will be triggered, and any subsequent received messages will be lost until adequate space is cleared in the Receive FIFO. The first message in the Receive FIFO will be mapped to the 13-byte Receive Buffer until that message is cleared (using the Release Receive Buffer command bit). After being cleared, the Receive Buffer will map to the next message in the Receive FIFO, and the space occupied by the previous message in the Receive FIFO can be used to receive new messages.

## 31.4 Functional Description

### 31.4.1 Modes

The ESP32-C3 TWAI controller has two working modes: Reset Mode and Operation Mode. Reset Mode and Operation Mode are entered by setting or clearing the TWAI_RESET_MODE bit.

#### 31.4.1.1 Reset Mode

Entering Reset Mode is required in order to modify the various configuration registers of the TWAI controller. When entering Reset Mode, the TWAI controller is essentially disconnected from the TWAI bus. When in Reset Mode, the TWAI controller will not be able to transmit any messages (including error signals). Any transmission in progress is immediately terminated. Likewise, the TWAI controller will not be able to receive any messages either.

#### 31.4.1.2 Operation Mode

In operation mode, the TWAI controller connects to the bus and write-protect all configuration registers to ensure consistency during operation. When in Operation Mode, the TWAI controller can transmit and receive messages (including error signaling) depending on which operation sub-mode the TWAI controller was configured with. The TWAI controller supports the following operation sub-modes:

- **Normal Mode:** The TWAI controller can transmit and receive messages including error signals (such as error and overload Frames).
- **Self-test Mode:** Self-test mode is similar to normal Mode, but the TWAI controller will consider the transmission of a data or RTR frame successful and do not generate an ACK error even if it was not acknowledged. This is commonly used when the TWAI controller does self-test.
- **Listen-only Mode:** The TWAI controller will be able to receive messages, but will remain completely passive on the TWAI bus. Thus, the TWAI controller will not be able to transmit any messages, acknowledgments, or error signals. The error counters will remain frozen. This mode is useful for TWAI bus monitoring.

Note that when exiting Reset Mode (i.e., entering Operation Mode), the TWAI controller must wait for 11 consecutive recessive bits to occur before being able to fully connect the TWAI bus (i.e., be able to transmit or receive).

### 31.4.2 Bit Timing

The operating bit rate of the TWAI controller must be configured whilst the TWAI controller is in Reset Mode. The bit rate is configured using TWAI_BUS_TIMING_0_REG and TWAI_BUS_TIMING_1_REG, and the two registers contain the following fields:

The following Table 31.4-1 illustrates the bit fields of TWAI_BUS_TIMING_0_REG.

**Table 31.4-1. Bit Information of TWAI_BUS_TIMING_0_REG (0x18)**

| Bit 31-16 | Bit 15 | Bit 14 | Bit 13 | Bit 12 | ..... | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| Reserved | SJW.1 | SJW.0 | Reserved | BRP.12 | ..... | BRP.1 | BRP.0 |

**Notes:**

- BRP: The TWAI Time Quanta clock is derived from the APB clock that is usually 80 MHz. The Baud Rate Prescaler (BRP) field is used to define the prescaler according to the equation below, where $t_{Tq}$ is the Time Quanta clock cycle and $t_{CLK}$ is APB clock cycle:
  $t_{Tq} = 2 \times t_{CLK} \times (2^{12} \times BRP.12 + 2^{11} \times BRP.11 + ... + 2^{1} \times BRP.1 + 2^{0} \times BRP.0 + 1)$
- SJW: Synchronization Jump Width (SJW) is configured in SJW.0 and SJW.1 where SJW = (2 x SJW.1 + SJW.0 + 1).

The following Table 31.4-2 illustrates the bit fields of TWAI_BUS_TIMING_1_REG.

**Table 31.4-2. Bit Information of TWAI_BUS_TIMING_1_REG (0x1c)**

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | SAM | PBS2.2 | PBS2.1 | PBS2.0 | PBS1.3 | PBS1.2 | PBS1.1 | PBS1.0 |

**Notes:**

- PBS1: The number of Time Quanta in Phase Buffer Segment 1 is defined according to the following equation: (8 x PBS1.3 + 4 x PBS1.2 + 2 x PBS1.1 + PBS1.0 + 1).
- PBS2: The number of Time Quanta in Phase Buffer Segment 2 is defined according to the following equation: (4 x PBS2.2 + 2 x PBS2.1 + PBS2.0 + 1).
- SAM: Enables triple sampling if set to 1. This is useful for low/medium speed buses to filter spikes on the bus line.

### 31.4.3 Interrupt Management

The ESP32-C3 TWAI controller provides eight interrupts, each represented by a single bit in the TWAI_INT_RAW_REG. For a particular interrupt to be triggered, the corresponding enable bit in TWAI_INT ENA_REG must be set.

The TWAI controller provides the following interrupts:

- Receive Interrupt
- Transmit Interrupt
- Error Warning Interrupt
- Data Overrun Interrupt
- Error Passive Interrupt
- Arbitration Lost Interrupt
- Bus Error Interrupt
- Bus Status Interrupt

The TWAI controller's interrupt signal to the interrupt matrix will be asserted whenever one or more interrupt bits are set in the TWAI_INT_RAW_REG, and deasserted when all bits in TWAI_INT_RAW_REG are cleared. The majority of interrupt bits in TWAI_INT_RAW_REG are automatically cleared when the register is read, except for the Receive Interrupt which can only be cleared when all the messages are released by setting the TWAI_RELEASE_BUF bit.

#### 31.4.3.1 Receive Interrupt (RXI)

The Receive Interrupt (RXI) is asserted whenever the TWAI controller has received messages that are pending to be read from the Receive Buffer (i.e., when TWAI_RX_MESSAGE_CNT_REG > 0). Pending received messages includes valid messages in the Receive FIFO and also overrun messages. The RXI will not be deasserted until all pending received messages are cleared using the TWAI_RELEASE_BUF command bit.

#### 31.4.3.2 Transmit Interrupt (TXI)

The Transmit Interrupt (TXI) is triggered whenever Transmit Buffer becomes free, indicating another message can be loaded into the Transmit Buffer to be transmitted. The Transmit Buffer becomes free under the following scenarios:

- A message transmission has completed successfully, i.e., acknowledged without any errors. (Any failed messages will automatically be resent.)
- A single shot transmission has completed (successfully or unsuccessfully, indicated by the TWAI_TX_COMPLETE bit).
- A message transmission was aborted using the TWAI_ABORT_TX command bit.

#### 31.4.3.3 Error Warning Interrupt (EWI)

The Error Warning Interrupt (EWI) is triggered whenever there is a change to the TWAI_ERR_ST and TWAI_BUS_OFF_ST bits of the TWAI_STATUS_REG (i.e., transition from 0 to 1 or vice versa). Thus, an EWI

could indicate one of the following events, depending on the values TWAI_ERR_ST and TWAI_BUS_OFF_ST at the moment when the EWI is triggered.

- If TWAI_ERR_ST = 0 and TWAI_BUS_OFF_ST = 0:
  - If the TWAI controller was in the Error Active state, it indicates both the TEC and REC have returned below the threshold value set by TWAI_ERR_WARNING_LIMIT_REG.
  - If the TWAI controller was previously in the Bus Off Recovery state, it indicates that Bus Recovery has completed successfully.
- If TWAI_ERR_ST = 1 and TWAI_BUS_OFF_ST = 0: The TEC or REC error counters have exceeded the threshold value set by TWAI_ERR_WARNING_LIMIT_REG.
- If TWAI_ERR_ST = 1 and TWAI_BUS_OFF_ST = 1: The TWAI controller has entered the BUS_OFF state (due to the TEC >= 256).
- If TWAI_ERR_ST = 0 and TWAI_BUS_OFF_ST = 1: The TWAI controller's TEC has dropped below the threshold value set by TWAI_ERR_WARNING_LIMIT_REG during BUS_OFF recovery.

#### 31.4.3.4 Data Overrun Interrupt (DOI)

The Data Overrun Interrupt (DOI) is triggered whenever the Receive FIFO has overrun. The DOI indicates that the Receive FIFO is full and should be cleared immediately to prevent any further overrun messages.

The DOI is only triggered by the first message that causes the Receive FIFO to overrun (i.e., the transition from the Receive FIFO not being full to the Receive FIFO overrunning). Any subsequent overrun messages will not trigger the DOI again. The DOI could be triggered again when all received messages (valid or overrun) have been cleared.

#### 31.4.3.5 Error Passive Interrupt (TXI)

The Error Passive Interrupt (EPI) is triggered whenever the TWAI controller switches from Error Active to Error Passive, or vice versa.

#### 31.4.3.6 Arbitration Lost Interrupt (ALI)

The Arbitration Lost Interrupt (ALI) is triggered whenever the TWAI controller is attempting to transmit a message and loses arbitration. The bit position where the TWAI controller lost arbitration is automatically recorded in Arbitration Lost Capture register (TWAI_ARB LOST CAP_REG). When the ALI occurs again, the Arbitration Lost Capture register will no longer record new bit location until it is cleared (via CPU reading this register).

#### 31.4.3.7 Bus Error Interrupt (BEI)

The Bus Error Interrupt (BEI) is triggered whenever TWAI controller detects an error on the TWAI bus. When a bus error occurs, the Bus Error type and its bit position are automatically recorded in the Error Code Capture register (TWAI_ERR_CODE_CAP_REG). When the BEI occurs again, the Error Code Capture register will no longer record new error information until it is cleared (via a read from the CPU).

#### 31.4.3.8 Bus Status Interrupt (BSI)

The Bus Status Interrupt (BSI) is triggered whenever TWAI controller is switching between receive/transmit status and idle status. When a BSI occurs, the current status of TWAI controller can be measured by reading TWAI_RX_ST and TWAI_TX_ST in TWAI_STATUS_REG register.

### 31.4.4 Transmit and Receive Buffers

#### 31.4.4.1 Overview of Buffers

Table 31.4-3. Buffer Layout for Standard Frame Format and Extended Frame Format

| Standard Frame Format (SFF) | | Extended Frame Format (EFF) | |
|---|---|---|---|
| TWAI Address | Content | TWAI Address | Content |
| 0x40 | TX/RX frame information | 0x40 | TX/RX frame information |
| 0x44 | TX/RX identifier 1 | 0x44 | TX/RX identifier 1 |
| 0x48 | TX/RX identifier 2 | 0x48 | TX/RX identifier 2 |
| 0x4c | TX/RX data byte 1 | 0x4c | TX/RX identifier 3 |
| 0x50 | TX/RX data byte 2 | 0x50 | TX/RX identifier 4 |
| 0x54 | TX/RX data byte 3 | 0x54 | TX/RX data byte 1 |
| 0x58 | TX/RX data byte 4 | 0x58 | TX/RX data byte 2 |
| 0x5c | TX/RX data byte 5 | 0x5c | TX/RX data byte 3 |
| 0x60 | TX/RX data byte 6 | 0x60 | TX/RX data byte 4 |
| 0x64 | TX/RX data byte 7 | 0x64 | TX/RX data byte 5 |
| 0x68 | TX/RX data byte 8 | 0x68 | TX/RX data byte 6 |
| 0x6c | reserved | 0x6c | TX/RX data byte 7 |
| 0x70 | reserved | 0x70 | TX/RX data byte 8 |

Table 31.4-3 illustrates the layout of the Transmit Buffer and Receive Buffer registers. Both the Transmit and Receive Buffer registers share the same address space and are only accessible when the TWAI controller is in Operation Mode. The CPU accesses Transmit Buffer registers for write operations, and Receive Buffer registers for read operations . Both buffers share the exact same register layout and fields to represent a message (received or to be transmitted). The Transmit Buffer registers are used to configure a TWAI message to be transmitted. The CPU would write to the Transmit Buffer registers specifying the message's frame type, frame format, frame ID, and frame data (payload). Once the Transmit Buffer is configured, the CPU would then initiate the transmission by setting the TWAI_TX_REQ bit in TWAI_CMD_REG.

- For a self-reception request, set the TWAI_SELF_RX_REQ bit instead.
- For a single-shot transmission, set both the TWAI_TX_REQ and the TWAI_ABORT_TX simultaneously.

The Receive Buffer registers map the first message in the Receive FIFO. The CPU would read the Receive Buffer registers to obtain the first message's frame type, frame format, frame ID, and frame data (payload). Once the message has been read from the Receive Buffer registers, the CPU can set the TWAI_RELEASE_BUF bit in TWAI_CMD_REG to clear the Receive Buffer registers. If there are still messages in the Receive FIFO, the Receive Buffer registers will map the first message again.

#### 31.4.4.2 Frame Information

The frame information is one byte long and specifies a message's frame type, frame format, and length of data. The frame information fields are shown in Table 31.4-4.

Table 31.4-4. TX/RX Frame Information (SFF/EFF); TWAI Address 0x40

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | FF[1] | RTR[2] | X[3] | X[3] | DLC.3[4] | DLC.2[4] | DLC.1[4] | DLC.0[4] |

Notes:

1. FF: The Frame Format (FF) bit specifies whether the message is Extended Frame Format (EFF) or Standard Frame Format (SFF). The message is EFF when FF bit is 1, and SFF when FF bit is 0.
2. RTR: The Remote Transmission Request (RTR) bit specifies whether the message is a data frame or a remote frame. The message is a remote frame when the RTR bit is 1, and a data frame when the RTR bit is 0.
3. X: Don't care, can be any value.
4. DLC: The Data Length Code (DLC) field specifies the number of data bytes for a data frame, or the number of data bytes to request in a remote frame. TWAI data frames are limited to a maximum payload of 8 data bytes, and thus the DLC should range anywhere from 0 to 8.

#### 31.4.4.3 Frame Identifier

The Frame Identifier fields is two-byte (11-bit) long if the message is SFF, and four-byte (29-bit) long if the message is EFF.

The Frame Identifier fields for an SFF (11-bit) message is shown in Table 31.4-5 ~ 31.4-6.

Table 31.4-5. TX/RX Identifier 1 (SFF); TWAI Address 0x44

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | ID.10 | ID.9 | ID.8 | ID.7 | ID.6 | ID.5 | ID.4 | ID.3 |

Table 31.4-6. TX/RX Identifier 2 (SFF); TWAI Address 0x48

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | ID.2 | ID.1 | ID.0 | X[1] | X[2] | X[2] | X[2] | X[2] |

Notes:

1. Don't care. Recommended to be compatible with receive buffer (i.e., set to RTR ) in case of using the self reception functionality (or together with self-test functionality).
2. Don't care. Recommended to be compatible with receive buffer (i.e., set to 0 ) in case of using the self reception functionality (or together with self-test functionality).

The Frame Identifier fields for an EFF (29-bits) message is shown in Table 31.4-7 ~ 31.4-10.

Table 31.4-7. TX/RX Identifier 1 (EFF); TWAI Address 0x44

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | ID.28 | ID.27 | ID.26 | ID.25 | ID.24 | ID.23 | ID.22 | ID.21 |

Table 31.4-8. TX/RX Identifier 2 (EFF); TWAI Address 0x48

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | ID.20 | ID.19 | ID.18 | ID.17 | ID.16 | ID.15 | ID.14 | ID.13 |

Table 31.4-9. TX/RX Identifier 3 (EFF); TWAI Address 0x4c

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | ID.12 | ID.11 | ID.10 | ID.9 | ID.8 | ID.7 | ID.6 | ID.5 |

Table 31.4-10. TX/RX Identifier 4 (EFF); TWAI Address 0x50

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | ID.4 | ID.3 | ID.2 | ID.1 | ID.0 | X[1] | X[2] | X[2] |

Notes:

1. Don't care. Recommended to be compatible with receive buffer (i.e., set to RTR ) in case of using the self reception functionality (or together with self-test functionality).
2. Don't care. Recommended to be compatible with receive buffer (i.e., set to 0 ) in case of using the self reception functionality (or together with self-test functionality).

#### 31.4.4.4 Frame Data

The Frame Data field contains the payloads of transmitted or received data frame, and can range from 0 to eight bytes. The number of valid bytes should be equal to the DLC. However, if the DLC is larger than eight bytes, the number of valid bytes would still be limited to eight. Remote frames do not have data payloads, so their Frame Data fields will be unused.

For example, when transmitting a data frame with five bytes, the CPU should write five to the DLC field, and then write data to the corresponding register of the first to the fifth data field. Likewise, when the CPU receives a data frame with a DLC of five data bytes, only the first to the fifth data byte will contain valid payload data for the CPU to read.

### 31.4.5 Receive FIFO and Data Overruns

The Receive FIFO is a 64-byte internal buffer used to store received messages in First In First Out order. A single received message can occupy between 3 to 13 bytes of space in the Receive FIFO, and their endianness is identical to the register layout of the Receive Buffer registers. The Receive Buffer registers are mapped to the bytes of the first message in the Receive FIFO.

When the TWAI controller receives a message, it will increment the value of TWAI_RX_MESSAGE_COUNTER up to a maximum of 64. If there is adequate space in the Receive FIFO, the message contents will be written into

the Receive FIFO. Once a message has been read from the Receive Buffer, the TWAI_RELEASE_BUF bit should be set. This will decrement TWAI_RX_MESSAGE_COUNTER and free the space occupied by the first message in the Receive FIFO. The Receive Buffer will then map to the next message in the Receive FIFO.

A data overrun occurs when the TWAI controller receives a message, but the Receive FIFO lacks the adequate free space to store the received message in its entirety (either due to the message contents being larger than the free space in the Receive FIFO, or the Receive FIFO being completely full).

When a data overrun occurs:

- The free space left in the Receive FIFO is filled with the partial contents of the overrun message. If the Receive FIFO is already full, then none of the overrun message's contents will be stored.
- When data in the Receive FIFO overruns for the first time, a Data Overrun Interrupt will be triggered.
- Each overrun message will still increment the TWAI_RX_MESSAGE_COUNTER up to a maximum of 64.
- The Receive FIFO will internally mark overrun messages as invalid. The TWAI_MISS_ST bit can be used to determine whether the message currently mapped to by the Receive Buffer is valid or overrun.

To clear an overrun Receive FIFO, the TWAI_RELEASE_BUF must be called repeatedly until TWAI_RX_MESSAGE_COUNTER is 0. This has the effect of reading all valid messages in the Receive FIFO and clearing all overrun messages.

### 31.4.6 Acceptance Filter

The Acceptance Filter allows the TWAI controller to filter out received messages based on their ID (and optionally their first data byte and frame type). Only accepted messages are passed on to the Receive FIFO. The use of Acceptance Filters allows a more lightweight operation of the TWAI controller (e.g., less use of Receive FIFO, fewer Receive Interrupts) since the TWAI Controller only need to handle a subset of messages.

The Acceptance Filter configuration registers can only be accessed whilst the TWAI controller is in Reset Mode, since they share the same address spaces with the Transmit Buffer and Receive Buffer registers.

The configuration registers consist of a 32-bit Acceptance Code Value and a 32-bit Acceptance Mask Value. The Acceptance Code value specifies a bit pattern which each filtered bit of the message must match in order for the message to be accepted. The Acceptance Mask Value is able to mask out certain bits of the Code value (i.e., set as "Don't Care" bits). Each filtered bit of the message must either match the acceptance code or be masked in order for the message to be accepted, as demonstrated in Figure 31.4-1.

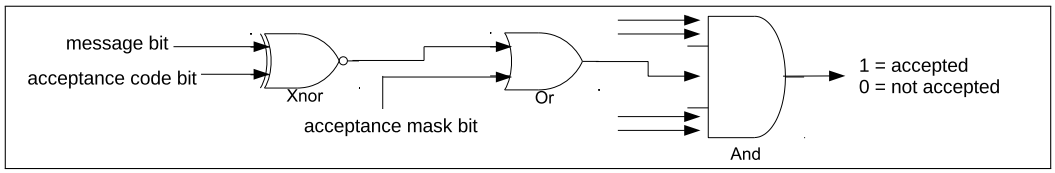



**Figure 31.4-1. Acceptance Filter**

The TWAI controller Acceptance Filter allows the 32-bit Acceptance Code and Mask Values to either define a single filter (i.e., Single Filter Mode), or two filters (i.e., Dual Filter Mode). How the Acceptance Filter interprets the 32-bit code and mask values is dependent on filter mode and the format of received messages (i.e., SFF or EFF).

#### 31.4.6.1 Single Filter Mode

Single Filter Mode is enabled by setting the TWAI_RX_FILTER_MODE bit to 1. This will cause the 32-bit code and mask values to define a single filter. The single filter can filter the following bits of a data or remote frame:

- SFF
  - The entire 11-bit ID
  - RTR bit
  - Data byte 1 and Data byte 2
- EFF
  - The entire 29-bit ID
  - RTR bit

The following Figure 31.4-2 illustrates how the 32-bit code and mask values will be interpreted under Single Filter Mode.

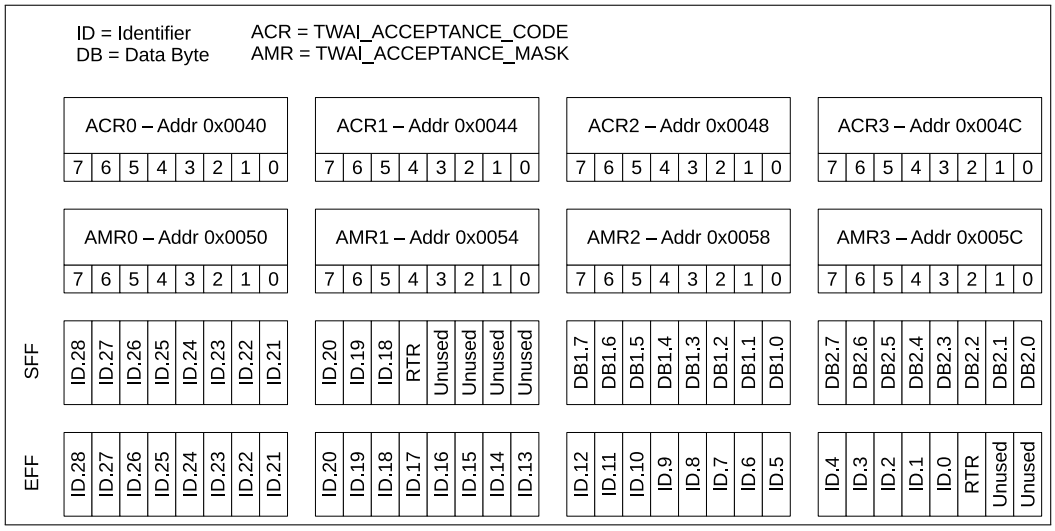



**Figure 31.4-2. Single Filter Mode**

#### 31.4.6.2 Dual Filter Mode

Dual Filter Mode is enabled by clearing the TWAI_RX_FILTER_MODE bit to 0. This will cause the 32-bit code and mask values to define a two separate filters referred to as filter 1 or filter 2. Under Dual Filter Mode, a message will be accepted if it is accepted by one of the two filters.

The two filters can filter the following bits of a data or remote frame:

- SFF
  - The entire 11-bit ID
  - RTR bit
  - Data byte 1 (for filter 1 only)
- EFF

- The first 16 bits of the 29-bit ID

The following Figure 31.4-3 illustrates how the 32-bit code and mask values will be interpreted in Dual Filter Mode.

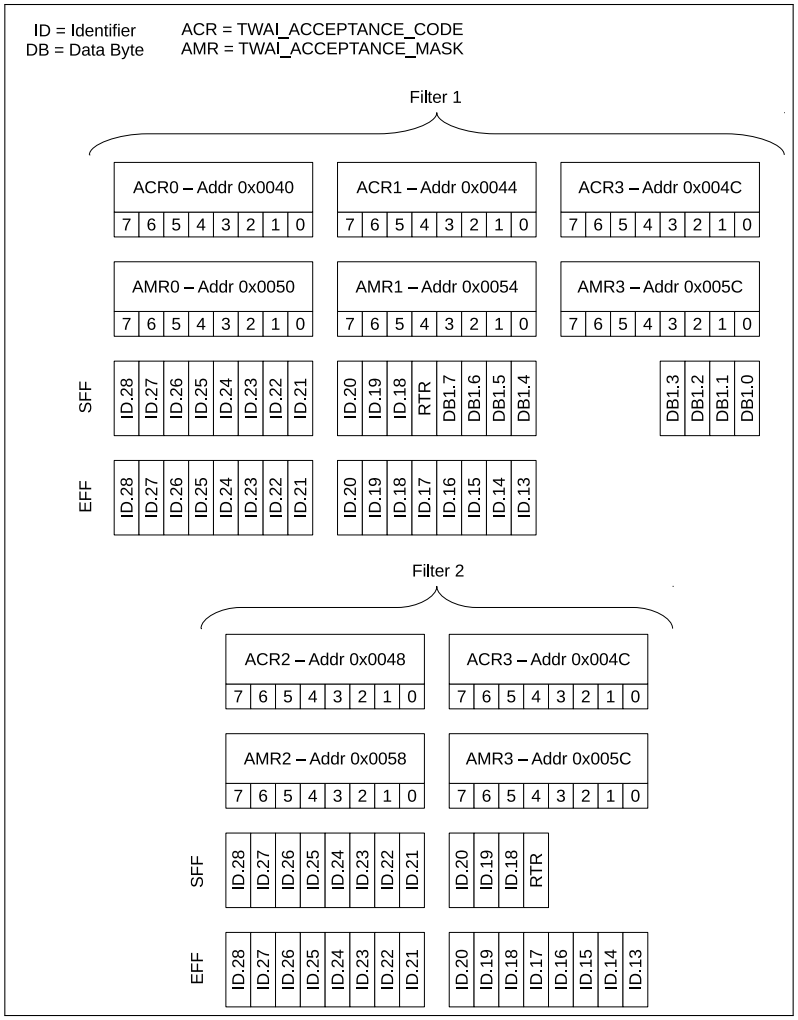



Figure 31.4-3. Dual Filter Mode

## 31.4.7 Error Management

The TWAI protocol requires that each TWAI node maintains the Transmit Error Counter (TEC) and Receive Error Counter (REC). The value of both error counters determines the current error state of the TWAI controller (i.e., Error Active, Error Passive, Bus-Off). The TWAI controller stores the TEC and REC values in TWAI_TX_ERR_CNT_REG and TWAI_RX_ERR_CNT_REG respectively, and they can be read by the CPU anytime. In addition to the error states, the TWAI controller also offers an Error Warning Limit (EWL) feature that can warn users of the occurrence of severe bus errors before the TWAI controller enters the Error Passive state.

The current error state of the TWAI controller is indicated via a combination of the following values and status bits: TEC, REC, TWAI_ERR_ST, and TWAI_BUS_OFF_ST. Certain changes to these values and bits will also trigger interrupts, thus allowing the users to be notified of error state transitions (see section 31.4.3). The following figure 31.4-4 shows the relation between the error states, values and bits, and error state related interrupts.

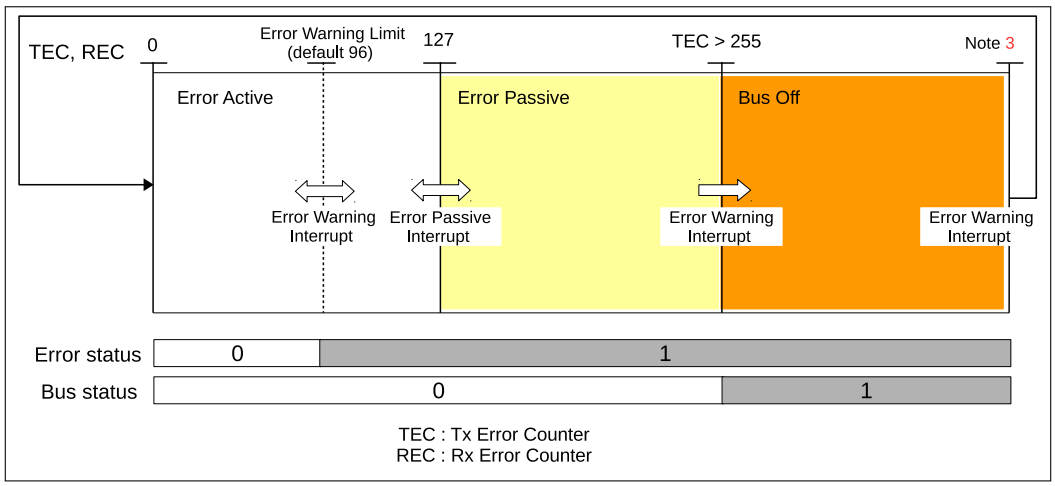



Figure 31.4-4. Error State Transition

#### 31.4.7.1 Error Warning Limit

The Error Warning Limit (EWL) is a configurable threshold value for the TEC and REC, which will trigger an interrupt when exceeded. The EWL is intended to serve as a warning about severe TWAI bus errors, and is triggered before the TWAI controller enters the Error Passive state. The EWL is configured in TWAI_ERR_WARNING_LIMIT_REG and can only be configured whilst the TWAI controller is in Reset Mode. The TWAI_ERR_WARNING_LIMIT_REG has a default value of 96. When the values of TEC and/or REC are larger than or equal to the EWL value, the TWAI_ERR_ST bit is immediately set to 1. Likewise, when the values of both the TEC and REC are smaller than the EWL value, the TWAI_ERR_ST bit is immediately reset to 0. The Error Warning Interrupt is triggered whenever the value of the TWAI_ERR_ST bit (or the TWAI_BUS_OFF_ST) changes.

#### 31.4.7.2 Error Passive

The TWAI controller is in the Error Passive state when the TEC or REC value exceeds 127. Likewise, when both the TEC and REC are less than or equal to 127, the TWAI controller enters the Error Active state. The Error Passive Interrupt is triggered whenever the TWAI controller transitions from the Error Active state to the Error Passive state or vice versa.

#### 31.4.7.3 Bus-Off and Bus-Off Recovery

The TWAI controller enters the Bus-Off state when the TEC value exceeds 255. On entering the Bus-Off state, the TWAI controller will automatically do the following:

- Set REC to 0
- Set TEC to 127
- Set the TWAI_BUS_OFF_ST bit to 1
- Enter Reset Mode

The Error Warning Interrupt is triggered whenever the value of the TWAI_BUS_OFF_ST bit (or the TWAI_ERR_ST bit) changes.

To return to the Error Active state, the TWAI controller must undergo Bus-Off Recovery. Bus-Off Recovery requires the TWAI controller to observe 128 occurrences of 11 consecutive recessive bits on the bus. To initiate Bus-Off Recovery (after entering the Bus-Off state), the TWAI controller should enter Operation Mode

by setting the TWAI_RESET_MODE bit to 0. The TEC tracks the progress of Bus-Off Recovery by decrementing the TEC each time when the TWAI controller observes 11 consecutive recessive bits. When Bus-Off Recovery has completed (i.e., TEC has decremented from 127 to 0), the TWAI_BUS_OFF_ST bit will automatically be reset to 0, thus triggering the Error Warning Interrupt.

### 31.4.8 Error Code Capture

The Error Code Capture (ECC) feature allows the TWAI controller to record the error type and bit position of a TWAI bus error in the form of an error code. Upon detecting a TWAI bus error, the Bus Error Interrupt is triggered and the error code is recorded in TWAI_ERR_CODE_CAP_REG. Subsequent bus errors will trigger the Bus Error Interrupt, but their error codes will not be recorded until the current error code is read from the TWAI_ERR_CODE_CAP_REG.

The following Table 31.4-11 shows the fields of the TWAI_ERR_CODE_CAP_REG:

**Table 31.4-11. Bit Information of TWAI_ERR_CODE_CAP_REG (0x30)**

| Bit 31-8 | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Reserved | ERRC.1[1] | ERRC.0[1] | DIR[2] | SEG.4[3] | SEG.3[3] | SEG.2[3] | SEG.1[3] | SEG.0[3] |

**Notes:**

- ERRC: The Error Code (ERRC) indicates the type of bus error: 00 for bit error, 01 for format error, 10 for stuff error, and 11 for other types of error.
- DIR: The Direction (DIR) indicates whether the TWAI controller was transmitting or receiving when the bus error occurred: 0 for transmitter, 1 for receiver.
- SEG: The Error Segment (SEG) indicates which segment of the TWAI message (i.e., bit position) the bus error occurred at.

The following Table 31.4-12 shows how to interpret the SEG.0 to SEG.4 bits.

**Table 31.4-12. Bit Information of Bits SEG.4 - SEG.0**

| Bit SEG.4 | Bit SEG.3 | Bit SEG.2 | Bit SEG.1 | Bit SEG.0 | Description |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | start of frame |
| 0 | 0 | 0 | 1 | 0 | ID.28 ~ ID.21 |
| 0 | 0 | 1 | 1 | 0 | ID.20 ~ ID.18 |
| 0 | 0 | 1 | 0 | 0 | bit SRTR |
| 0 | 0 | 1 | 0 | 1 | bit IDE |
| 0 | 0 | 1 | 1 | 1 | ID.17 ~ ID.13 |
| 0 | 1 | 1 | 1 | 1 | ID.12 ~ ID.5 |
| 0 | 1 | 1 | 1 | 0 | ID.4 ~ ID.0 |
| 0 | 1 | 1 | 0 | 0 | bit RTR |
| 0 | 1 | 1 | 0 | 1 | reserved bit 1 |
| 0 | 1 | 0 | 0 | 1 | reserved bit 0 |
| 0 | 1 | 0 | 1 | 1 | data length code |
| 0 | 1 | 0 | 1 | 0 | data field |
| 0 | 1 | 0 | 0 | 0 | CRC sequence |

Cont'd on next page

Table 31.4-12 – cont'd from previous page

| Bit SEG.4 | Bit SEG.3 | Bit SEG.2 | Bit SEG.1 | Bit SEG.0 | Description |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | CRC delimiter |
| 1 | 1 | 0 | 0 | 1 | ACK slot |
| 1 | 1 | 0 | 1 | 1 | ACK delimiter |
| 1 | 1 | 0 | 1 | 0 | end of frame |
| 1 | 0 | 0 | 1 | 0 | intermission |
| 1 | 0 | 0 | 0 | 1 | active error flag |
| 1 | 0 | 1 | 1 | 0 | passive error flag |
| 1 | 0 | 0 | 1 | 1 | tolerate dominant bits |
| 1 | 0 | 1 | 1 | 1 | error delimiter |
| 1 | 1 | 1 | 0 | 0 | overload flag |

Notes:

- Bit SRTR: under Standard Frame Format.
- Bit IDE: Identifier Extension Bit, 0 for Standard Frame Format.

### 31.4.9 Arbitration Lost Capture

The Arbitration Lost Capture (ALC) feature allows the TWAI controller to record the bit position where it loses arbitration. When the TWAI controller loses arbitration, the bit position is recorded in TWAI_ARB LOST CAP_REG and the Arbitration Lost Interrupt is triggered.

Subsequent losses in arbitration will trigger the Arbitration Lost Interrupt, but will not be recorded in TWAI_ARB LOST CAP_REG until the current Arbitration Lost Capture is read from the TWAI_ERR_CODE_CAP_REG.

Table 31.4-13 illustrates bits and fields of TWAI_ERR_CODE_CAP_REG whilst Figure 31.4-5 illustrates the bit positions of a TWAI message.

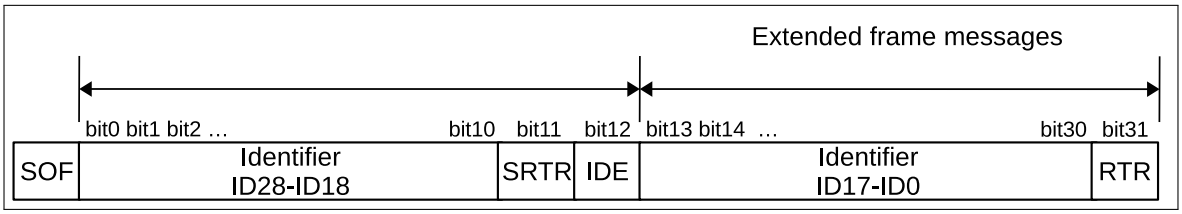



Figure 31.4-5. Positions of Arbitration Lost Bits

Table 31.4-13. Bit Information of TWAI_ARB LOST CAP_REG (0x2c)

| Bit 31-5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|
| Reserved | BITNO.4[1] | BITNO.3[1] | BITNO.2[1] | BITNO.1[1] | BITNO.0[1] |

Notes:

- BITNO: Bit Number (BITNO) indicates the nth bit of a TWAI message where arbitration was lost.

## 31.5 Register Summary

'|' here means separate line to distinguish between TWAI working modes discussed in Section 31.4.1 *Modes*. The left describes the access in Operation Mode. The right belongs to Reset Mode and is marked in red. The addresses in this section are relative to Two-wire Automotive Interface base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Registers** | | | |
| TWAI_MODE_REG | Mode Register | 0x0000 | R/W |
| TWAI_BUS_TIMING_0_REG | Bus Timing Register 0 | 0x0018 | RO \| R/W |
| TWAI_BUS_TIMING_1_REG | Bus Timing Register 1 | 0x001C | RO \| R/W |
| TWAI_ERR_WARNING_LIMIT_REG | Error Warning Limit Register | 0x0034 | RO \| R/W |
| TWAI_DATA_0_REG | Data Register 0 | 0x0040 | WO \| R/W |
| TWAI_DATA_1_REG | Data Register 1 | 0x0044 | WO \| R/W |
| TWAI_DATA_2_REG | Data Register 2 | 0x0048 | WO \| R/W |
| TWAI_DATA_3_REG | Data Register 3 | 0x004C | WO \| R/W |
| TWAI_DATA_4_REG | Data Register 4 | 0x0050 | WO \| R/W |
| TWAI_DATA_5_REG | Data Register 5 | 0x0054 | WO \| R/W |
| TWAI_DATA_6_REG | Data Register 6 | 0x0058 | WO \| R/W |
| TWAI_DATA_7_REG | Data Register 7 | 0x005C | WO \| R/W |
| TWAI_DATA_8_REG | Data Register 8 | 0x0060 | WO \| RO |
| TWAI_DATA_9_REG | Data Register 9 | 0x0064 | WO \| RO |
| TWAI_DATA_10_REG | Data Register 10 | 0x0068 | WO \| RO |
| TWAI_DATA_11_REG | Data Register 11 | 0x006C | WO \| RO |
| TWAI_DATA_12_REG | Data Register 12 | 0x0070 | WO \| RO |
| TWAI_CLOCK_DIVIDER_REG | Clock Divider Register | 0x007C | varies |
| **Contro Registers** | | | |
| TWAI_CMD_REG | Command Register | 0x0004 | WO |
| **Status Register** | | | |
| TWAI_STATUS_REG | Status Register | 0x0008 | RO |
| TWAI_ARB LOST CAP_REG | Arbitration Lost Capture Register | 0x002C | RO |
| TWAI_ERR_CODE_CAP_REG | Error Code Capture Register | 0x0030 | RO |
| TWAI_RX_ERR_CNT_REG | Receive Error Counter Register | 0x0038 | RO \| R/W |
| TWAI_TX_ERR_CNT_REG | Transmit Error Counter Register | 0x003C | RO \| R/W |
| TWAI_RX_MESSAGE_CNT_REG | Receive Message Counter Register | 0x0074 | RO |
| **Interrupt Registers** | | | |
| TWAI_INT_RAW_REG | Interrupt Register | 0x000C | RO |
| TWAI_INT ENA_REG | Interrupt Enable Register | 0x0010 | R/W |

## 31.6 Registers

'|' here means separate line. The left describes the access in Operation Mode. The right belongs to Reset Mode with red color. The addresses in this section are relative to Two-wire Automotive Interface base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 31.1. TWAI_MODE_REG (0x0000)**

| (reserved) | TWAI_RX_FILTER_MODE | TWAI_SELF_TEST_MODE | TWAI_LISTEN_ONLY_MODE | TWAI_RESET_MODE | |
|---|---|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 1 | Reset |

**TWAI_RESET_MODE** This bit is used to configure the operation mode of the TWAI Controller. 1: Reset mode; 0: Operation mode (R/W)

**TWAI_LISTEN_ONLY_MODE** 1: Listen only mode. In this mode the nodes will only receive messages from the bus, without generating the acknowledge signal nor updating the RX error counter. (R/W)

**TWAI_SELF_TEST_MODE** 1: Self test mode. In this mode the TX nodes can perform a successful transmission without receiving the acknowledge signal. This mode is often used to test a single node with the self reception request command. (R/W)

**TWAI_RX_FILTER_MODE** This bit is used to configure the filter mode. 0: Dual filter mode; 1: Single filter mode (R/W)

**Register 31.2. TWAI_BUS_TIMING_0_REG (0x0018)**

| (reserved) | TWAI_SYNC_JUMP_WIDTH | (reserved) | TWAI_BAUD_PRESC | |
|---|---|---|---|---|
| 31 … 16 | 15 … 14 | 13 | 12 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | 0x0 | 0x00 | Reset |

**TWAI_BAUD_PRESC** Baud Rate Prescaler value, determines the frequency dividing ratio. (RO | R/W)

**TWAI_SYNC_JUMP_WIDTH** Synchronization Jump Width (SJW), 1 ~ 14 Tq wide. (RO | R/W)

**Register 31.3. TWAI_BUS_TIMING_1_REG (0x001C)**

| (reserved) | TWAI_TIME_SAMP | TWAI_TIME_SEG2 | TWAI_TIME_SEG1 | |
|---|---|---|---|---|
| 31 … 8 | 7 | 6 … 4 | 3 … 0 | |
| 0 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 | 0x0 | Reset |

**TWAI_TIME_SEG1** The width of PBS1. (RO | R/W)

**TWAI_TIME_SEG2** The width of PBS2. (RO | R/W)

**TWAI_TIME_SAMP** The number of sample points. 0: the bus is sampled once; 1: the bus is sampled three times (RO | R/W)

**Register 31.4. TWAI_ERR_WARNING_LIMIT_REG (0x0034)**

| (reserved) | TWAI_ERR_WARNING_LIMIT | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x60 | Reset |

**TWAI_ERR_WARNING_LIMIT** Error warning threshold. In the case when any of an error counter value exceeds the threshold, or all the error counter values are below the threshold, an error warning interrupt will be triggered (given the enable signal is valid). (RO | R/W)

**Register 31.5. TWAI_DATA_0_REG (0x0040)**

| (reserved) | TWAI_TX_BYTE_0 \| TWAI_ACCEPTANCE_CODE_0 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_0** Stored the 0th byte information of the data to be transmitted in operation mode. (WO)

**TWAI_ACCEPTANCE_CODE_0** Stored the 0th byte of the filter code in reset mode. (R/W)

**Register 31.6. TWAI_DATA_1_REG (0x0044)**

| (reserved) | TWAI_TX_BYTE_1 \| TWAI_ACCEPTANCE_CODE_1 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_1** Stored the 1st byte information of the data to be transmitted in operation mode. (WO)

**TWAI_ACCEPTANCE_CODE_1** Stored the 1st byte of the filter code in reset mode. (R/W)

Register 31.7. TWAI_DATA_2_REG (0x0048)

| (reserved) | TWAI_TX_BYTE_2 \| TWAI_ACCEPTANCE_CODE_2 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_2** Stored the 2nd byte information of the data to be transmitted in operation mode. (WO)

**TWAI_ACCEPTANCE_CODE_2** Stored the 2nd byte of the filter code in reset mode. (R/W)

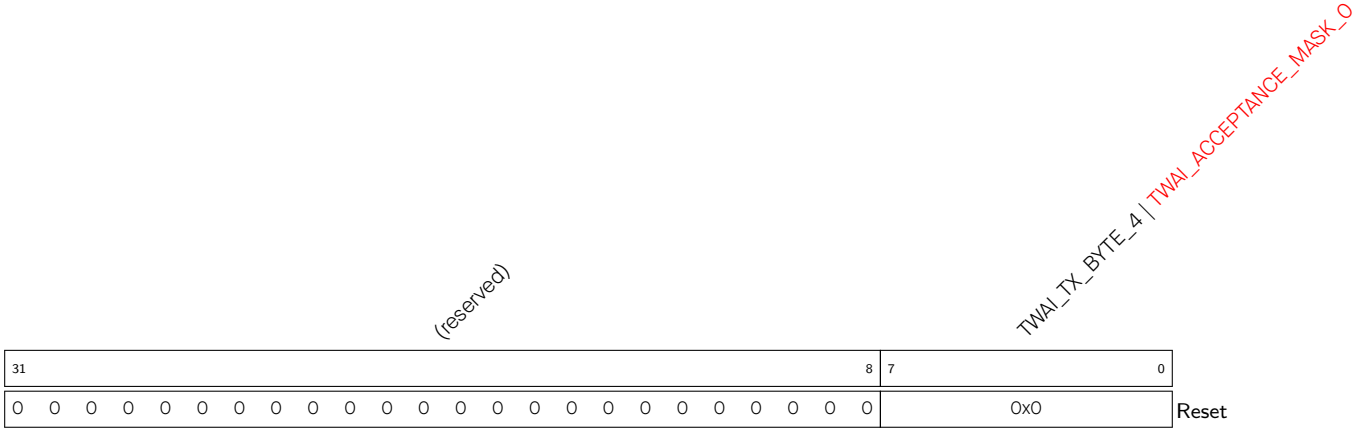
Register 31.8. TWAI_DATA_3_REG (0x004C)

| (reserved) | TWAI_TX_BYTE_3 \| TWAI_ACCEPTANCE_CODE_3 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_3** Stored the 3rd byte information of the data to be transmitted in operation mode. (WO)

**TWAI_ACCEPTANCE_CODE_3** Stored the 3rd byte of the filter code in reset mode. (R/W)

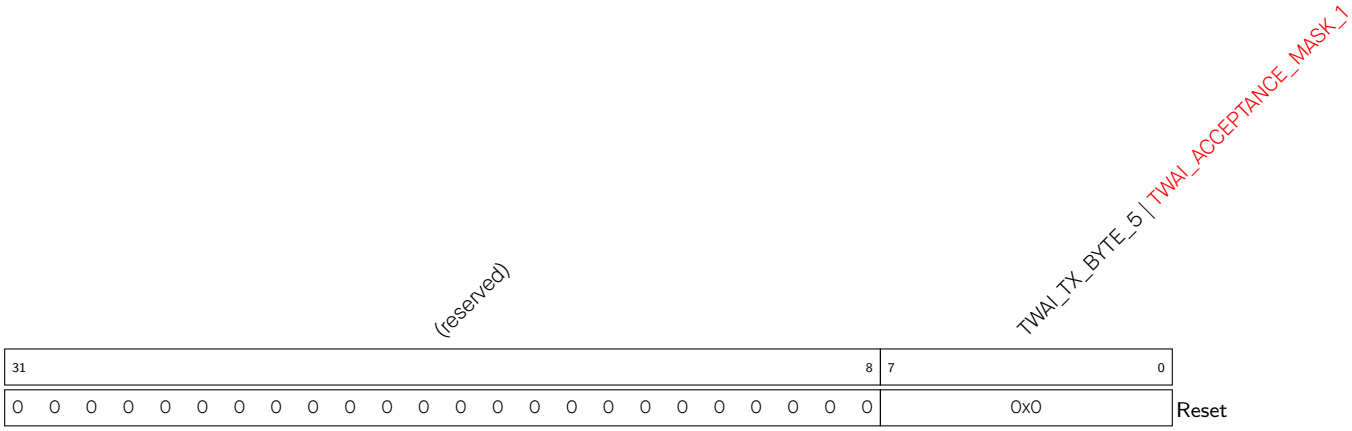
### Register 31.9. TWAI_DATA_4_REG (0x0050)




**TWAI_TX_BYTE_4** Stored the 4th byte information of the data to be transmitted in operation mode. (WO)

**TWAI_ACCEPTANCE_MASK_0** Stored the 0th byte of the filter code in reset mode. (R/W)

### Register 31.10. TWAI_DATA_5_REG (0x0054)




**TWAI_TX_BYTE_5** Stored the 5th byte information of the data to be transmitted in operation mode. (WO)

**TWAI_ACCEPTANCE_MASK_1** Stored the 1st byte of the filter code in reset mode. (R/W)

**Register 31.11. TWAI_DATA_6_REG (0x0058)**

| (reserved) | TWAI_TX_BYTE_6 \| TWAI_ACCEPTANCE_MASK_2 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_6** Stored the 6th byte information of the data to be transmitted in operation mode. (WO)

**TWAI_ACCEPTANCE_MASK_2** Stored the 2nd byte of the filter code in reset mode. (R/W)

**Register 31.12. TWAI_DATA_7_REG (0x005C)**

| (reserved) | TWAI_TX_BYTE_7 \| TWAI_ACCEPTANCE_MASK_3 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_7** Stored the 7th byte information of the data to be transmitted in operation mode. (WO)

**TWAI_ACCEPTANCE_MASK_3** Stored the 3rd byte of the filter code in reset mode. (R/W)

**Register 31.13. TWAI_DATA_8_REG (0x0060)**

| (reserved) | TWAI_TX_BYTE_8 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_8** Stored the 8th byte information of the data to be transmitted in operation mode. (WO)

**Register 31.14. TWAI_DATA_9_REG (0x0064)**

| (reserved) | TWAI_TX_BYTE_9 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_9** Stored the 9th byte information of the data to be transmitted in operation mode. (WO)

**Register 31.15. TWAI_DATA_10_REG (0x0068)**

| (reserved) | TWAI_TX_BYTE_10 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_10** Stored the 10th byte information of the data to be transmitted in operation mode. (WO)

### Register 31.16. TWAI_DATA_11_REG (0x006C)

| (reserved) | TWAI_TX_BYTE_11 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_11** Stored the 11th byte information of the data to be transmitted in operation mode. (WO)

### Register 31.17. TWAI_DATA_12_REG (0x0070)

| (reserved) | TWAI_TX_BYTE_12 | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

**TWAI_TX_BYTE_12** Stored the 12th byte information of the data to be transmitted in operation mode. (WO)

### Register 31.18. TWAI_CLOCK_DIVIDER_REG (0x007C)

| (reserved) | TWAI_CLOCK_OFF | TWAI_CD | |
|---|---|---|---|
| 31 … 9 | 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0x0 | Reset |

**TWAI_CD** These bits are used to configure the divisor of the external CLKOUT pin. (R/W)

**TWAI_CLOCK_OFF** This bit can be configured in reset mode. 1: Disable the external CLKOUT pin; 0: Enable the external CLKOUT pin (RO | R/W)

**Register 31.19. TWAI_CMD_REG (0x0004)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | | | | | TWAI_SELF_RX_REQ | TWAI_CLR_OVERRUN | TWAI_RELEASE_BUF | TWAI_ABORT_TX | TWAI_TX_REQ | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**TWAI_TX_REQ** Set the bit to 1 to drive nodes to start transmission. (WO)

**TWAI_ABORT_TX** Set the bit to 1 to cancel a pending transmission request. (WO)

**TWAI_RELEASE_BUF** Set the bit to 1 to release the RX buffer. (WO)

**TWAI_CLR_OVERRUN** Set the bit to 1 to clear the data overrun status bit. (WO)

**TWAI_SELF_RX_REQ** Self reception request command. Set the bit to 1 to allow a message be transmitted and received simultaneously. (WO)

**Register 31.20. TWAI_STATUS_REG (0x0008)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | TWAI_MISS_ST | TWAI_BUS_OFF_ST | TWAI_ERR_ST | TWAI_TX_ST | TWAI_RX_ST | TWAI_TX_COMPLETE | TWAI_TX_BUF_ST | TWAI_OVERRUN_ST | TWAI_RX_BUF_ST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 | Reset |

**TWAI_RX_BUF_ST** 1: The data in the RX buffer is not empty, with at least one received data packet. (RO)

**TWAI_OVERRUN_ST** 1: The RX FIFO is full and data overrun has occurred. (RO)

**TWAI_TX_BUF_ST** 1: The TX buffer is empty, the CPU may write a message into it. (RO)

**TWAI_TX_COMPLETE** 1: The TWAI controller has successfully received a packet from the bus. (RO)

**TWAI_RX_ST** 1: The TWAI Controller is receiving a message from the bus. (RO)

**TWAI_TX_ST** 1: The TWAI Controller is transmitting a message to the bus. (RO)

**TWAI_ERR_ST** 1: At least one of the RX/TX error counter has reached or exceeded the value set in register TWAI_ERR_WARNING_LIMIT_REG. (RO)

**TWAI_BUS_OFF_ST** 1: In bus-off status, the TWAI Controller is no longer involved in bus activities. (RO)

**TWAI_MISS_ST** This bit reflects whether the data packet in the RX FIFO is complete. 1: The current packet is missing; 0: The current packet is complete (RO)

**Register 31.21. TWAI_ARB LOST CAP_REG (0x002C)**

| (reserved) | TWAI_ARB_LOST_CAP | |
|---|---|---|
| 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 | 0x0 | Reset |

**TWAI_ARB_LOST_CAP** This register contains information about the bit position of lost arbitration. (RO)

**Register 31.22. TWAI_ERR_CODE_CAP_REG (0x0030)**

| (reserved) | TWAI_ECC_TYPE | TWAI_ECC_DIRECTION | TWAI_ECC_SEGMENT | |
|---|---|---|---|---|
| 31 … 8 | 7 … 6 | 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 | 0x0 | 0 | 0x0 | Reset |

**TWAI_ECC_SEGMENT** This register contains information about the location of errors, see Table 31.4-11 for details. (RO)

**TWAI_ECC_DIRECTION** This register contains information about transmission direction of the node when error occurs. 1: Error occurs when receiving a message; 0: Error occurs when transmitting a message (RO)

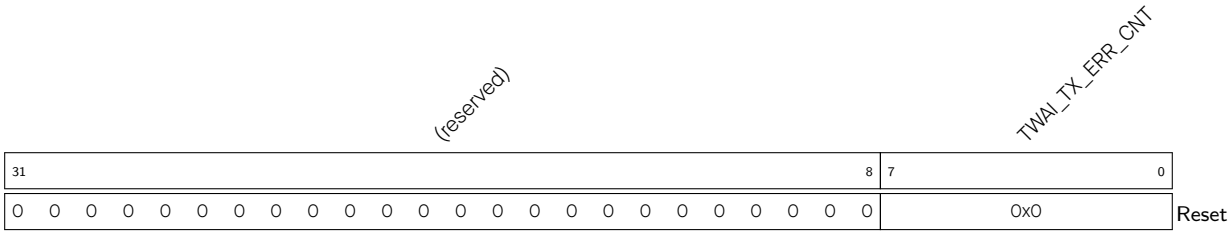
**TWAI_ECC_TYPE** This register contains information about error types: 00: bit error; 01: form error; 10: stuff error; 11: other type of error (RO)

**Register 31.23. TWAI_RX_ERR_CNT_REG (0x0038)**

| (reserved) | TWAI_RX_ERR_CNT | |
|---|---|---|
| 31 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x0 | Reset |

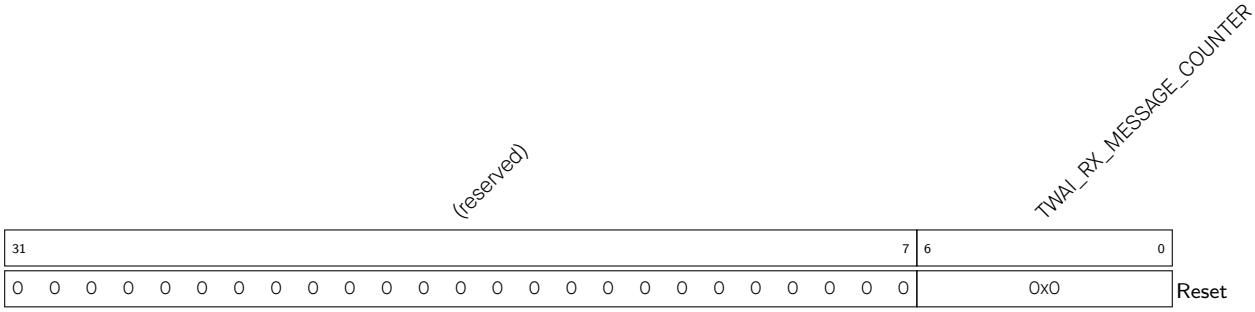
**TWAI_RX_ERR_CNT** The RX error counter register, reflects value changes in reception status. (RO | R/W)

**Register 31.24. TWAI_TX_ERR_CNT_REG (0x003C)**




**TWAI_TX_ERR_CNT** The TX error counter register, reflects value changes in transmission status. (RO | R/W)

**Register 31.25. TWAI_RX_MESSAGE_CNT_REG (0x0074)**




**TWAI_RX_MESSAGE_COUNTER** This register reflects the number of messages available within the RX FIFO. (RO)

**Register 31.26. TWAI_INT_RAW_REG (0x000C)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | TWAI_BUS_STATE_INT_ST | TWAI_BUS_ERR_INT_ST | TWAI_ARB_LOST_INT_ST | TWAI_ERR_PASSIVE_INT_ST | (reserved) | TWAI_OVERRUN_INT_ST | TWAI_ERR_WARN_INT_ST | TWAI_TX_INT_ST | TWAI_RX_INT_ST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**TWAI_RX_INT_ST** Receive interrupt. If this bit is set to 1, it indicates there are messages to be handled in the RX FIFO. (RO)

**TWAI_TX_INT_ST** Transmit interrupt. If this bit is set to 1, it indicates the message transmission is finished and a new transmission is able to start. (RO)

**TWAI_ERR_WARN_INT_ST** Error warning interrupt. If this bit is set to 1, it indicates the error status signal and the bus-off status signal of Status register have changed (e.g., switched from 0 to 1 or from 1 to 0). (RO)

**TWAI_OVERRUN_INT_ST** Data overrun interrupt. If this bit is set to 1, it indicates a data overrun interrupt is generated in the RX FIFO. (RO)

**TWAI_ERR_PASSIVE_INT_ST** Error passive interrupt. If this bit is set to 1, it indicates the TWAI Controller is switched between error active status and error passive status due to the change of error counters. (RO)

**TWAI_ARB_LOST_INT_ST** Arbitration lost interrupt. If this bit is set to 1, it indicates an arbitration lost interrupt is generated. (RO)

**TWAI_BUS_ERR_INT_ST** Error interrupt. If this bit is set to 1, it indicates an error is detected on the bus. (RO)

**TWAI_BUS_STATE_INT_ST** Bus state interrupt. If this bit is set to 1, it indicates the status of TWAI controller has changed. (RO)

**Register 31.27. TWAI_INT ENA_REG (0x0010)**

| 31 | | | | | | | | | | | | | | | | | | | | | | | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | | | | | | | TWAI_BUS_STATE_INT_ENA | TWAI_BUS_ERR_INT_ENA | TWAI_ARB_LOST_INT_ENA | TWAI_ERR_PASSIVE_INT_ENA | (reserved) | TWAI_OVERRUN_INT_ENA | TWAI_ERR_WARN_INT_ENA | TWAI_TX_INT_ENA | TWAI_RX_INT_ENA | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**TWAI_RX_INT_ENA** Set this bit to 1 to enable receive interrupt. (R/W)

**TWAI_TX_INT_ENA** Set this bit to 1 to enable transmit interrupt. (R/W)

**TWAI_ERR_WARN_INT_ENA** Set this bit to 1 to enable error warning interrupt. (R/W)

**TWAI_OVERRUN_INT_ENA** Set this bit to 1 to enable data overrun interrupt. (R/W)

**TWAI_ERR_PASSIVE_INT_ENA** Set this bit to 1 to enable error passive interrupt. (R/W)

**TWAI_ARB_LOST_INT_ENA** Set this bit to 1 to enable arbitration lost interrupt. (R/W)

**TWAI_BUS_ERR_INT_ENA** Set this bit to 1 to enable bus error interrupt. (R/W)

**TWAI_BUS_STATE_INT_ENA** Set this bit to 1 to enable bus state interrupt. (R/W)

# Chapter 32

# LED PWM Controller (LEDC)

## 32.1 Overview

The LED PWM Controller is a peripheral designed to generate PWM signals for LED control. It has specialized features such as automatic duty cycle fading. However, the LED PWM Controller can also be used to generate PWM signals for other purposes.

## 32.2 Features

The LED PWM Controller has the following features:

- Six independent PWM generators (i.e. six channels)
- Four independent timers that support division by fractions
- Automatic duty cycle fading (i.e. gradual increase/decrease of a PWM's duty cycle without interference from the processor) with interrupt generation on fade completion
- Adjustable phase of PWM signal output
- PWM signal output in low-power mode (Light-sleep mode)
- Maximum PWM resolution: 14 bits

Note that the four timers are identical regarding their features and operation. The following sections refer to the timers collectively as Timer*x* (where *x* ranges from 0 to 3). Likewise, the six PWM generators are also identical in features and operation, and thus are collectively referred to as PWM*n* (where *n* ranges from 0 to 5).

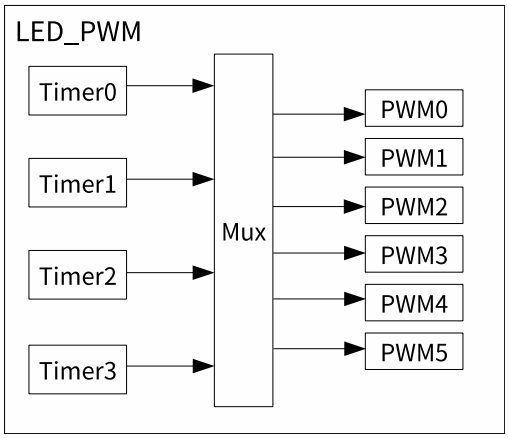



Figure 32.2-1. LED PWM Architecture

# 32.3 Functional Description

## 32.3.1 Architecture

Figure 32.2-1 shows the architecture of the LED PWM Controller.

The four timers can be independently configured (i.e. configurable clock divider, and counter overflow value) and each internally maintains a timebase counter (i.e. a counter that counts on cycles of a reference clock). Each PWM generator selects one of the timers and uses the timer's counter value as a reference to generate its PWM signal.

Figure 32.3-1 illustrates the main functional blocks of the timer and the PWM generator.

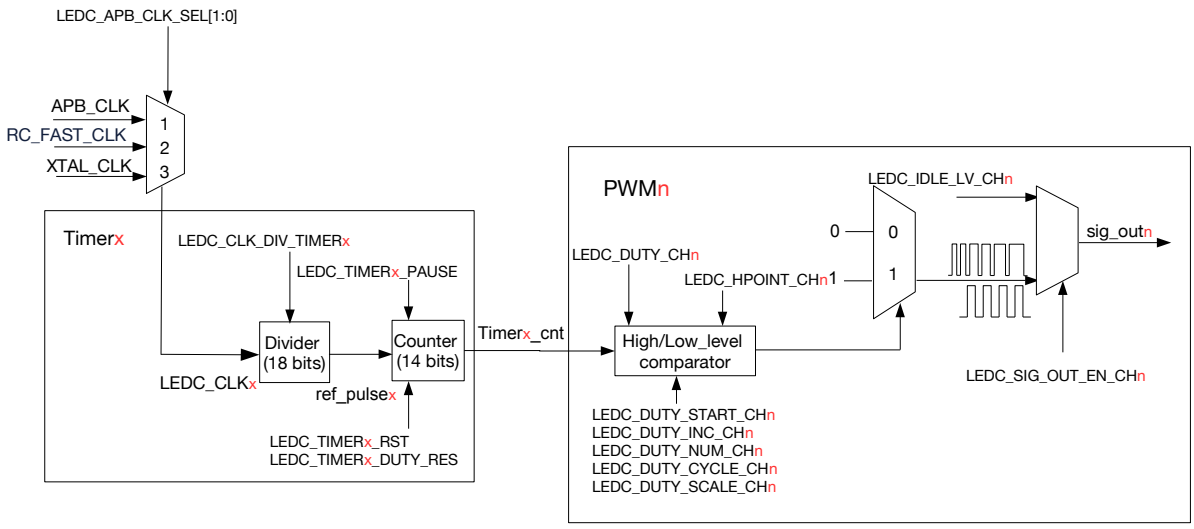



Figure 32.3-1. LED PWM Generator Diagram

## 32.3.2 Timers

Each timer in LED PWM Controller internally maintains a timebase counter. Referring to Figure 32.3-1, this clock signal used by the timebase counter is named ref_pulse*x*. All timers use the same clock source LEDC_CLK*x*, which is then passed through a clock divider to generate ref_pulse*x* for the counter.

### 32.3.2.1 Clock Source

LED PWM registers configured by software are clocked by APB_CLK. For more information about APB_CLK, see Chapter 6 *Reset and Clock*. To use the LED PWM peripheral, the APB_CLK signal to the LED PWM has to be enabled. The APB_CLK signal to LED PWM can be enabled by setting the SYSTEM_LEDC_CLK_EN field in the register SYSTEM_PERIP_CLK_EN0_REG and be reset via software by setting the SYSTEM_LEDC_RST field in the register SYSTEM_PERIP_RST_EN0_REG. For more information, please refer to Table 16.3-1 in Chapter 16 *System Registers (SYSREG)*.

Timers in the LED PWM Controller choose their common clock source from one of the following clock signals: APB_CLK, RC_FAST_CLK and XTAL_CLK (see Chapter 6 *Reset and Clock* for more details about each clock signal). The procedure for selecting a clock source signal for LEDC_CLK*x* is described below:

- APB_CLK: Set LEDC_APB_CLK_SEL[1:0] to 1
- RC_FAST_CLK: Set LEDC_APB_CLK_SEL[1:0] to 2
- XTAL_CLK: Set LEDC_APB_CLK_SEL[1:0] to 3

The LEDC_CLK*x* signal will then be passed through the clock divider.

#### 32.3.2.2 Clock Divider Configuration

The LEDC_CLK*x* signal is passed through a clock divider to generate the ref_pulse*x* signal for the counter. The frequency of ref_pulse*x* is equal to the frequency of LEDC_CLK*x* divided by the divisor LEDC_CLK_DIV (see Figure 32.3-1).

The divisor LEDC_CLK_DIV is a fractional value. Thus, it can be a non-integer. LEDC_CLK_DIV is configured according to the following equation.

$$LEDC_CLK_DIV = A + \frac{B}{256}$$

- *A* corresponds to the most significant 10 bits of LEDC_CLK_DIV_TIMER*x* (i.e. LEDC_TIMER*x*_CONF_REG[21:12])
- The fractional part *B* corresponds to the least significant 8 bits of LEDC_CLK_DIV_TIMER*x* (i.e. LEDC_TIMER*x*_CONF_REG[11:4])

When the fractional part *B* is zero, LEDC_CLK_DIV is equivalent to an integer divisor (i.e. an integer prescaler). In other words, a ref_pulse*x* clock pulse is generated after every *A* number of LEDC_CLK*x* clock pulses.

However, when *B* is nonzero, LEDC_CLK_DIV becomes a non-integer divisor. The clock divider implements non-integer frequency division by alternating between *A* and (*A*+1) LEDC_CLK*x* clock pulses per ref_pulse*x* clock pulse. This will result in the average frequency of ref_pulse*x* clock pulse being the desired frequency (i.e. the non-integer divided frequency). For every 256 ref_pulse*x* clock pulses:

- A number of *B* ref_pulse*x* clock pulses will consist of (*A*+1) LEDC_CLK*x* clock pulses
- A number of (256-*B*) ref_pulse*x* clock pulses will consist of *A* LEDC_CLK*x* clock pulses
- The ref_pulse*x* clock pulses consisting of (*A*+1) pulses are evenly distributed amongst those consisting of *A* pulses

Figure 32.3-2 illustrates the relation between LEDC_CLK*x* clock pulses and ref_pulse*x* clock pulses when dividing by a non-integer LEDC_CLK_DIV .

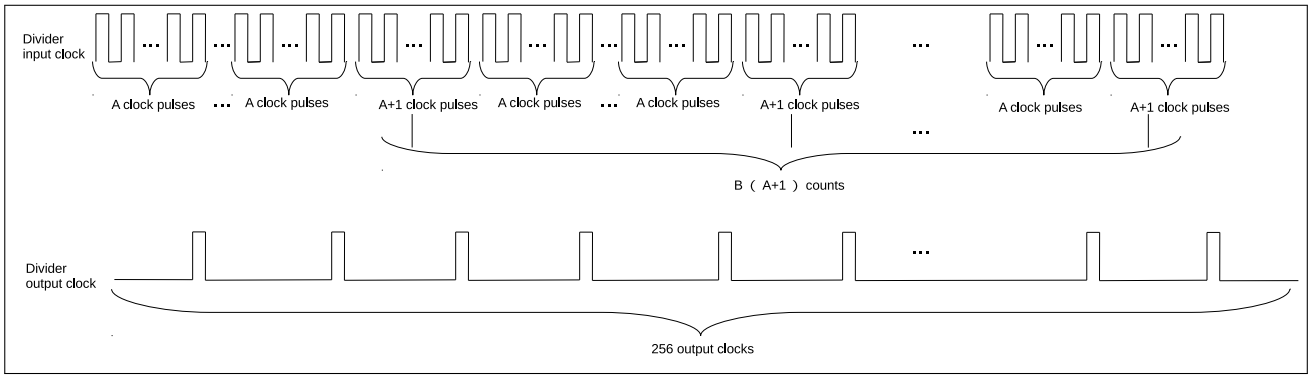



Figure 32.3-2. Frequency Division When LEDC_CLK_DIV is a Non-Integer Value

To change the timer's clock divisor at runtime, first configure the LEDC_CLK_DIV_TIMERx field, and then set the LEDC_TIMERx_PARA_UP field to apply the new configuration. This will cause the newly configured values to take effect upon the next overflow of the counter. The LEDC_TIMERx_PARA_UP field will be automatically cleared by hardware.

#### 32.3.2.3 14-bit Counter

Each timer contains a 14-bit timebase counter that uses ref_pulsex as its reference clock (see Figure 32.3-1). The LEDC_TIMERx_DUTY_RES field configures the overflow value of this 14-bit counter. Hence, the maximum resolution of the PWM signal is 14 bits. The counter counts up to $2^{LEDC_TIMERx_DUTY_RES} - 1$, overflows and begins counting from 0 again. The counter's value can be read, reset, and suspended by software.

The counter can trigger LEDC_TIMERx_OVF_INT interrupt (generated automatically by hardware without configuration) every time the counter overflows. It can also be configured to trigger LEDC_OVF_CNT_CHn_INT interrupt after the counter overflows $LEDC_OVF_NUM_CHn + 1$ times. To configure LEDC_OVF_CNT_CHn_INT interrupt, please:

1. Configure LEDC_TIMER_SEL_CHn as the counter for the PWM generator
2. Enable the counter by setting LEDC_OVF_CNT_EN_CHn
3. Set LEDC_OVF_NUM_CHn to the number of counter overflows to generate an interrupt, minus 1
4. Enable the overflow interrupt by setting LEDC_OVF_CNT_CHn_INT_ENA
5. Set LEDC_TIMERx_DUTY_RES to enable the timer and wait for a LEDC_OVF_CNT_CHn_INT interrupt

Referring to Figure 32.3-1, the frequency of a PWM generator output signal (sig_outn) is dependent on the frequency of the timer's clock source LEDC_CLKx, the clock divisor LEDC_CLK_DIV, and the duty resolution (counter width) LEDC_TIMERx_DUTY_RES:

$$f_{\text{PWM}} = \frac{f_{\text{LEDC_CLKx}}}{\text{LEDC_CLK_DIV} \cdot 2^{\text{LEDC_TIMERx_DUTY_RES}}}$$

Based on the formula above, the desired duty resolution can be calculated as follows:

$$\text{LEDC_TIMERx_DUTY_RES} = \log_2\left(\frac{f_{\text{LEDC_CLKx}}}{f_{\text{PWM}} \cdot \text{LEDC_CLK_DIV}}\right)$$

Table 32.3-1 lists the commonly-used frequencies and their corresponding resolutions.

**Table 32.3-1. Commonly-used Frequencies and Resolutions**

| LEDC_CLKx | PWM Frequency | Highest Resolution (bit) [1] | Lowest Resolution (bit) [2] |
|---|---|---|---|
| APB_CLK (80 MHz) | 1 kHz | 14 | 7 |
| APB_CLK (80 MHz) | 5 kHz | 13 | 4 |
| APB_CLK (80 MHz) | 10 kHz | 12 | 3 |
| XTAL_CLK (40 MHz) | 1 kHz | 14 | 6 |
| XTAL_CLK (40 MHz) | 4 kHz | 13 | 4 |

[1] The highest resolution is calculated when the clock divisor LEDC_CLK_DIV is 1 and rounded down. If the highest resolution calculated by the formula is higher than the counter's width 14 bits, then the highest resolution should be 14 bits.

[2] The lowest resolution is calculated when the clock divisor LEDC_CLK_DIV is $1023 + \frac{255}{256}$ and rounded up. If the lowest resolution calculated by the formula is lower than 0, then the lowest resolution should be 1.

Table 32.3-1. Commonly-used Frequencies and Resolutions

| LEDC_CLK*x* | PWM Frequency | Highest Resolution (bit) [1] | Lowest Resolution (bit) [2] |
|---|---|---|---|
| RC_FAST_CLK (17.5 MHz) | 1 kHz | 14 | 5 |
| RC_FAST_CLK (17.5 MHz) | 1.75 kHz | 13 | 4 |

[1] The highest resolution is calculated when the clock divisor LEDC_CLK_DIV is 1 and rounded down. If the highest resolution calculated by the formula is higher than the counter's width 14 bits, then the highest resolution should be 14 bits.

[2] The lowest resolution is calculated when the clock divisor LEDC_CLK_DIV is $1023 + \frac{255}{256}$ and rounded up. If the lowest resolution calculated by the formula is lower than 0, then the lowest resolution should be 1.

To change the overflow value at runtime, first set the LEDC_TIMER*x*_DUTY_RES field, and then set the LEDC_TIMER*x*_PARA_UP field. This will cause the newly configured values to take effect upon the next overflow of the counter. If LEDC_OVF_CNT_EN_CH*n* field is reconfigured, LEDC_PARA_UP_CH*n* should be set to apply the new configuration. In summary, these configuration values need to be updated by setting LEDC_TIMER*x*_PARA_UP or LEDC_PARA_UP_CH*n*. LEDC_TIMER*x*_PARA_UP and LEDC_PARA_UP_CH*n* will be automatically cleared by hardware.

### 32.3.3 PWM Generators

To generate a PWM signal, a PWM generator (PWM*n*) selects a timer (Timer*x*). Each PWM generator can be configured separately by setting LEDC_TIMER_SEL_CH*n* to use one of four timers to generate the PWM output.

As shown in Figure 32.3-1, each PWM generator has a comparator and two multiplexers. A PWM generator compares the timer's 14-bit counter value (Timer*x*_cnt) to two trigger values Hpoint*n* and Lpoint*n*. When the timer's counter value is equal to Hpoint*n* or Lpoint*n*, the PWM signal is high or low, respectively, as described below:

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

Figure 32.3-3 illustrates how Hpoint*n* or Lpoint*n* are used to generate a fixed duty cycle PWM output signal.

For a particular PWM generator (PWM*n*), its Hpoint*n* is sampled from the LEDC_HPOINT_CH*n* field each time the selected timer's counter overflows. Likewise, Lpoint*n* is also sampled on every counter overflow and is calculated from the sum of the LEDC_DUTY_CH*n*[18:4] and LEDC_HPOINT_CH*n* fields. By setting Hpoint*n* and Lpoint*n* via the LEDC_HPOINT_CH*n* and LEDC_DUTY_CH*n*[18:4] fields, the relative phase and duty cycle of the PWM output can be set.

The PWM output signal (sig_out*n*) is enabled by setting LEDC_SIG_OUT_EN_CH*n*. When LEDC_SIG_OUT_EN_CH*n* is cleared, PWM signal output is disabled, and the output signal (sig_out*n*) will output a constant level as specified by LEDC_IDLE_LV_CH*n*.

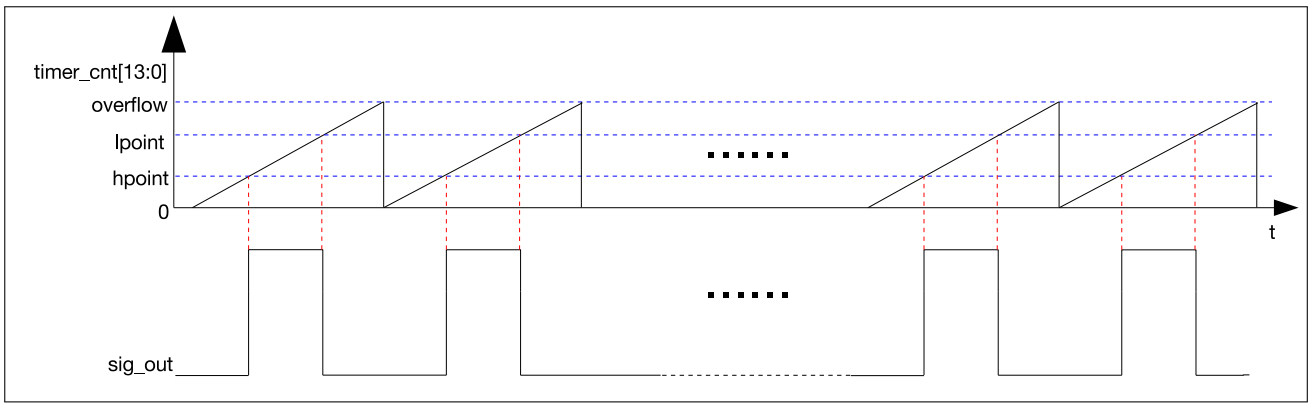



Figure 32.3-3. LED_PWM Output Signal Diagram

The bits LEDC_DUTY_CH*n*[3:0] are used to dither the duty cycles of the PWM output signal (sig_out*n*) by periodically altering the duty cycle of sig_out*n*. When LEDC_DUTY_CH*n*[3:0] is set to a non-zero value, then for every 16 cycles of sig_out*n*, LEDC_DUTY_CH*n*[3:0] of those cycles will have PWM pulses that are one timer tick longer than the other (16- LEDC_DUTY_CH*n*[3:0]) cycles. For instance, if LEDC_DUTY_CH*n*[18:4] is set to 10 and LEDC_DUTY_CH*n*[3:0] is set to 5, then 5 of 16 cycles will have a PWM pulse with a duty value of 11 and the rest of the 16 cycles will have a PWM pulse with a duty value of 10. The average duty cycle after 16 cycles is 10.3125.

If fields LEDC_TIMER_SEL_CH*n*, LEDC_HPOINT_CH*n*, LEDC_DUTY_CH*n*[18:4] and LEDC_SIG_OUT_EN_CH*n* are reconfigured, LEDC_PARA_UP_CH*n* must be set to apply the new configuration. This will cause the newly configured values to take effect upon the next overflow of the counter. LEDC_PARA_UP_CH*n* field will be automatically cleared by hardware.

## 32.3.4 Duty Cycle Fading

The PWM generators can fade the duty cycle of a PWM output signal (i.e. gradually change the duty cycle from one value to another). If Duty Cycle Fading is enabled, the value of Lpoint*n* will be incremented/decremented after a fixed number of counter overflows has occurred. Figure 32.3-4 illustrates Duty Cycle Fading.

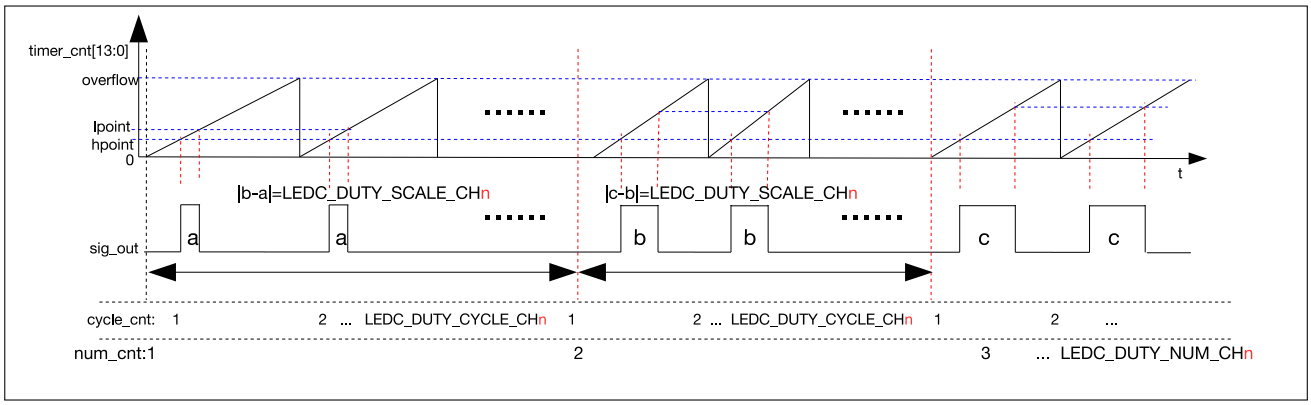



Figure 32.3-4. Output Signal Diagram of Fading Duty Cycle

Duty Cycle Fading is configured using the following register fields:

- LEDC_DUTY_CH*n* is used to set the initial value of Lpoint*n*.

- LEDC_DUTY_START_CH*n* will enable/disable duty cycle fading when set/cleared.
- LEDC_DUTY_CYCLE_CH*n* sets the number of counter overflow cycles for every Lpoint*n* increment/decrement. In other words, Lpoint*n* will be incremented/decremented after LEDC_DUTY_CYCLE_CH*n* counter overflows.
- LEDC_DUTY_INC_CH*n* configures whether Lpoint*n* is incremented/decremented if set/cleared.
- LEDC_DUTY_SCALE_CH*n* sets the amount that Lpoint*n* is incremented/decremented.
- LEDC_DUTY_NUM_CH*n* sets the maximum number of increments/decrements before duty cycle fading stops.

If the fields LEDC_DUTY_CH*n*, LEDC_DUTY_START_CH*n*, LEDC_DUTY_CYCLE_CH*n*, LEDC_DUTY_INC_CH*n*, LEDC_DUTY_SCALE_CH*n*, and LEDC_DUTY_NUM_CH*n* are reconfigured, LEDC_PARA_UP_CH*n* must be set to apply the new configuration. After this field is set, the values for duty cycle fading will take effect at once. LEDC_PARA_UP_CH*n* field will be automatically cleared by hardware.

### 32.3.5 Interrupts

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

## 32.4 Register Summary

The addresses in this section are relative to the LED PWM Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Register** | | | |
| LEDC_CH0_CONF0_REG | Configuration register 0 for channel 0 | 0x0000 | varies |
| LEDC_CH0_CONF1_REG | Configuration register 1 for channel 0 | 0x000C | varies |
| LEDC_CH1_CONF0_REG | Configuration register 0 for channel 1 | 0x0014 | varies |
| LEDC_CH1_CONF1_REG | Configuration register 1 for channel 1 | 0x0020 | varies |
| LEDC_CH2_CONF0_REG | Configuration register 0 for channel 2 | 0x0028 | varies |
| LEDC_CH2_CONF1_REG | Configuration register 1 for channel 2 | 0x0034 | varies |
| LEDC_CH3_CONF0_REG | Configuration register 0 for channel 3 | 0x003C | varies |
| LEDC_CH3_CONF1_REG | Configuration register 1 for channel 3 | 0x0048 | varies |
| LEDC_CH4_CONF0_REG | Configuration register 0 for channel 4 | 0x0050 | varies |
| LEDC_CH4_CONF1_REG | Configuration register 1 for channel 4 | 0x005C | varies |
| LEDC_CH5_CONF0_REG | Configuration register 0 for channel 5 | 0x0064 | varies |
| LEDC_CH5_CONF1_REG | Configuration register 1 for channel 5 | 0x0070 | varies |
| LEDC_CONF_REG | Global LEDC configuration register | 0x00D0 | R/W |
| **Hpoint Register** | | | |
| LEDC_CH0_HPOINT_REG | High point register for channel 0 | 0x0004 | R/W |
| LEDC_CH1_HPOINT_REG | High point register for channel 1 | 0x0018 | R/W |
| LEDC_CH2_HPOINT_REG | High point register for channel 2 | 0x002C | R/W |
| LEDC_CH3_HPOINT_REG | High point register for channel 3 | 0x0040 | R/W |
| LEDC_CH4_HPOINT_REG | High point register for channel 4 | 0x0054 | R/W |
| LEDC_CH5_HPOINT_REG | High point register for channel 5 | 0x0068 | R/W |
| **Duty Cycle Register** | | | |
| LEDC_CH0_DUTY_REG | Initial duty cycle for channel 0 | 0x0008 | R/W |
| LEDC_CH0_DUTY_R_REG | Current duty cycle for channel 0 | 0x0010 | RO |
| LEDC_CH1_DUTY_REG | Initial duty cycle for channel 1 | 0x001C | R/W |
| LEDC_CH1_DUTY_R_REG | Current duty cycle for channel 1 | 0x0024 | RO |
| LEDC_CH2_DUTY_REG | Initial duty cycle for channel 2 | 0x0030 | R/W |
| 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 |
| **Timer Register** | | | |
| LEDC_TIMER0_CONF_REG | Timer 0 configuration | 0x00A0 | varies |
| LEDC_TIMER0_VALUE_REG | Timer 0 current counter value | 0x00A4 | RO |

| Name | Description | Address | Access |
|---|---|---|---|
| LEDC_TIMER1_CONF_REG | Timer 1 configuration | 0x00A8 | varies |
| LEDC_TIMER1_VALUE_REG | Timer 1 current counter value | 0x00AC | RO |
| LEDC_TIMER2_CONF_REG | Timer 2 configuration | 0x00B0 | varies |
| LEDC_TIMER2_VALUE_REG | Timer 2 current counter value | 0x00B4 | RO |
| LEDC_TIMER3_CONF_REG | Timer 3 configuration | 0x00B8 | varies |
| LEDC_TIMER3_VALUE_REG | Timer 3 current counter value | 0x00BC | RO |
| **Interrupt Register** | | | |
| LEDC_INT_RAW_REG | Raw interrupt status | 0x00C0 | R/WTC/SS |
| 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 | WT |
| **Version Register** | | | |
| LEDC_DATE_REG | Version control register | 0x00FC | R/W |

## 32.5 Registers

The addresses in this section are relative to LED PWM Controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 32.1. LEDC_CH*n*_CONF0_REG (*n*: 0-5) (0x0000+20**n*)**

| (reserved) | LEDC_OVF_CNT_RESET_CH*n* | LEDC_OVF_CNT_EN_CH*n* | LEDC_OVF_NUM_CH*n* | LEDC_PARA_UP_CH*n* | LEDC_IDLE_LV_CH*n* | LEDC_SIG_OUT_EN_CH*n* | LEDC_TIMER_SEL_CH*n* | |
|---|---|---|---|---|---|---|---|---|
| 31 … 17 | 16 | 15 | 14 … 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 | Reset |

**LEDC_TIMER_SEL_CH*n*** This field is used to select one of the 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 (when LEDC_SIG_OUT_EN_CH*n* is 0). (R/W)

**LEDC_PARA_UP_CH*n*** This bit is used to update the listed fields below for channel *n*, and will be automatically cleared by hardware. (WT)

- LEDC_HPOINT_CH*n*
- LEDC_DUTY_START_CH*n*
- LEDC_SIG_OUT_EN_CH*n*
- LEDC_TIMER_SEL_CH*n*
- LEDC_DUTY_NUM_CH*n*
- LEDC_DUTY_CYCLE_CH*n*
- LEDC_DUTY_SCALE_CH*n*
- LEDC_DUTY_INC_CH*n*
- LEDC_OVF_CNT_EN_CH*n*

**LEDC_OVF_NUM_CH*n*** This field 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 count the number of times when the timer selected by channel *n* overflows. (R/W)

**LEDC_OVF_CNT_RESET_CH*n*** Set this bit to reset the timer-overflow counter of channel *n*. (WT)

**Register 32.2. LEDC_CH*n*_CONF1_REG (*n*: 0-5) (0x000C+20**n*)**

| LEDC_DUTY_START_CH*n* / LEDC_DUTY_INC_CH*n* | | LEDC_DUTY_NUM_CH*n* | LEDC_DUTY_CYCLE_CH*n* | LEDC_DUTY_SCALE_CH*n* | |
|---|---|---|---|---|---|
| 31 | 30 | 29 … 20 | 19 … 10 | 9 … 0 | |
| 0 | 1 | 0x0 | 0x0 | 0x0 | Reset |

**LEDC_DUTY_SCALE_CH*n*** This field configures the step size of the duty cycle change during fading. (R/W)

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

**LEDC_DUTY_NUM_CH*n*** This field controls the number of times the duty cycle will be changed. (R/W)

**LEDC_DUTY_INC_CH*n*** This bit determines whether the duty cycle of the output signal on channel *n* increases or decreases. 1: Increase; 0: Decrease. (R/W)

**LEDC_DUTY_START_CH*n*** If this bit is set to 1, other configured fields in LEDC_CH*n*_CONF1_REG will take effect upon the next timer overflow. (R/W/SC)

**Register 32.3. LEDC_CONF_REG (0x00D0)**

| LEDC_CLK_EN | (reserved) | LEDC_APB_CLK_SEL | |
|---|---|---|---|
| 31 | 30 … 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 | Reset |

**LEDC_APB_CLK_SEL** This field is used to select the common clock source for all the 4 timers.

1: APB_CLK; 2: RC_FAST_CLK; 3: XTAL_CLK. (R/W)

**LEDC_CLK_EN** This bit is used to control the clock.

1: Force clock on for register. 0: Support clock only when application writes registers. (R/W)

**Register 32.4. LEDC_CH*n*_HPOINT_REG (*n*: 0-5) (0x0004+20**n*)**

| (reserved) | LEDC_HPOINT_CH*n* | |
|---|---|---|
| 31 … 14 | 13 … 0 | |
| 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 timer for this channel has reached the value specified by this field. (R/W)

**Register 32.5. LEDC_CH*n*_DUTY_REG (*n*: 0-5) (0x0008+20**n*)**

| (reserved) | LEDC_DUTY_CH*n* | |
|---|---|---|
| 31 … 19 | 18 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x000 | Reset |

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

**Register 32.6. LEDC_CH*n*_DUTY_R_REG (*n*: 0-5) (0x0010+20**n*)**

| (reserved) | LEDC_DUTY_R_CH*n* | |
|---|---|---|
| 31 … 19 | 18 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0x000 | Reset |

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

**Register 32.7. LEDC_TIMERx_CONF_REG (x: 0-3) (0x00A0+8*x)**

| (reserved) | LEDC_TIMERx_PARA_UP | (reserved) | LEDC_TIMERx_RST | LEDC_TIMERx_PAUSE | LEDC_CLK_DIV_TIMERx | LEDC_TIMERx_DUTY_RES | |
|---|---|---|---|---|---|---|---|
| 31 26 | 25 | 24 | 23 | 22 | 21 4 | 3 0 | |
| 0 0 0 0 0 0 | 0 | 0 | 1 | 0 | 0x000 | 0x0 | Reset |

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

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

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

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

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

**Register 32.8. LEDC_TIMERx_VALUE_REG (x: 0-3) (0x00A4+8*x)**

| (reserved) | LEDC_TIMERx_CNT | |
|---|---|---|
| 31 14 | 13 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

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

**Register 32.9. LEDC_INT_RAW_REG (0x00C0)**

| 31–16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | LEDC_OVF_CNT_CH5_INT_RAW | LEDC_OVF_CNT_CH4_INT_RAW | LEDC_OVF_CNT_CH3_INT_RAW | LEDC_OVF_CNT_CH2_INT_RAW | LEDC_OVF_CNT_CH1_INT_RAW | LEDC_OVF_CNT_CH0_INT_RAW | LEDC_DUTY_CHNG_END_CH5_INT_RAW | LEDC_DUTY_CHNG_END_CH4_INT_RAW | LEDC_DUTY_CHNG_END_CH3_INT_RAW | LEDC_DUTY_CHNG_END_CH2_INT_RAW | LEDC_DUTY_CHNG_END_CH1_INT_RAW | LEDC_DUTY_CHNG_END_CH0_INT_RAW | LEDC_TIMER3_OVF_INT_RAW | LEDC_TIMER2_OVF_INT_RAW | LEDC_TIMER1_OVF_INT_RAW | LEDC_TIMER0_OVF_INT_RAW | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**LEDC_TIMER*x*_OVF_INT_RAW** Triggered when the timer*x* has reached its maximum counter value. (R/WTC/SS)

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

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

**Register 32.10. LEDC_INT_ST_REG (0x00C4)**

| 31–16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | LEDC_OVF_CNT_CH5_INT_ST | LEDC_OVF_CNT_CH4_INT_ST | LEDC_OVF_CNT_CH3_INT_ST | LEDC_OVF_CNT_CH2_INT_ST | LEDC_OVF_CNT_CH1_INT_ST | LEDC_OVF_CNT_CH0_INT_ST | LEDC_DUTY_CHNG_END_CH5_INT_ST | LEDC_DUTY_CHNG_END_CH4_INT_ST | LEDC_DUTY_CHNG_END_CH3_INT_ST | LEDC_DUTY_CHNG_END_CH2_INT_ST | LEDC_DUTY_CHNG_END_CH1_INT_ST | LEDC_DUTY_CHNG_END_CH0_INT_ST | LEDC_TIMER3_OVF_INT_ST | LEDC_TIMER2_OVF_INT_ST | LEDC_TIMER1_OVF_INT_ST | LEDC_TIMER0_OVF_INT_ST | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

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

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

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

### Register 32.11. LEDC_INT_ENA_REG (0x00C8)

| 31–16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | LEDC_OVF_CNT_CH5_INT_ENA | LEDC_OVF_CNT_CH4_INT_ENA | LEDC_OVF_CNT_CH3_INT_ENA | LEDC_OVF_CNT_CH2_INT_ENA | LEDC_OVF_CNT_CH1_INT_ENA | LEDC_OVF_CNT_CH0_INT_ENA | LEDC_DUTY_CHNG_END_CH5_INT_ENA | LEDC_DUTY_CHNG_END_CH4_INT_ENA | LEDC_DUTY_CHNG_END_CH3_INT_ENA | LEDC_DUTY_CHNG_END_CH2_INT_ENA | LEDC_DUTY_CHNG_END_CH1_INT_ENA | LEDC_DUTY_CHNG_END_CH0_INT_ENA | LEDC_TIMER3_OVF_INT_ENA | LEDC_TIMER2_OVF_INT_ENA | LEDC_TIMER1_OVF_INT_ENA | LEDC_TIMER0_OVF_INT_ENA | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**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 32.12. LEDC_INT_CLR_REG (0x00CC)

| 31–16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | LEDC_OVF_CNT_CH5_INT_CLR | LEDC_OVF_CNT_CH4_INT_CLR | LEDC_OVF_CNT_CH3_INT_CLR | LEDC_OVF_CNT_CH2_INT_CLR | LEDC_OVF_CNT_CH1_INT_CLR | LEDC_OVF_CNT_CH0_INT_CLR | LEDC_DUTY_CHNG_END_CH5_INT_CLR | LEDC_DUTY_CHNG_END_CH4_INT_CLR | LEDC_DUTY_CHNG_END_CH3_INT_CLR | LEDC_DUTY_CHNG_END_CH2_INT_CLR | LEDC_DUTY_CHNG_END_CH1_INT_CLR | LEDC_DUTY_CHNG_END_CH0_INT_CLR | LEDC_TIMER3_OVF_INT_CLR | LEDC_TIMER2_OVF_INT_CLR | LEDC_TIMER1_OVF_INT_CLR | LEDC_TIMER0_OVF_INT_CLR | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

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

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

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

**Register 32.13. LEDC_DATE_REG (0x00FC)**

| LEDC_LEDC_DATE | |
|---|---|
| 31 … 0 | |
| 0x19061700 | Reset |

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

# Chapter 33

# Remote Control Peripheral (RMT)

## 33.1 Overview

The RMT (Remote Control) module is designed to send and receive infrared remote control signals. A variety of remote control protocols are supported. 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. Optionally, the RMT module modulates its output signals with a carrier wave, or demodulates and filters its input signals.

The RMT module has four channels, numbered from zero to three. Channels 0 ~ 1 (TX channels) are dedicated to transmit signals, and channels 2 ~ 3 (RX channels) to receive signals. Each TX/RX channel has the same functionality controlled by a dedicated set of registers and is able to independently either transmit or receive data. TX channels are indicated by *n* which is used as a placeholder for the channel number, and by *m* for RX channels.

## 33.2 Features

- Two TX channels
- Two RX channels
- Support multiple channels (programmable) transmitting data simultaneously
- Four channels share a 192 x 32-bit RAM
- Support modulation on TX pulses
- Support filtering and demodulation on RX pulses
- Wrap TX mode
- Wrap RX mode
- Continuous TX mode

## 33.3 Functional Description

## 33.3.1 RMT Architecture

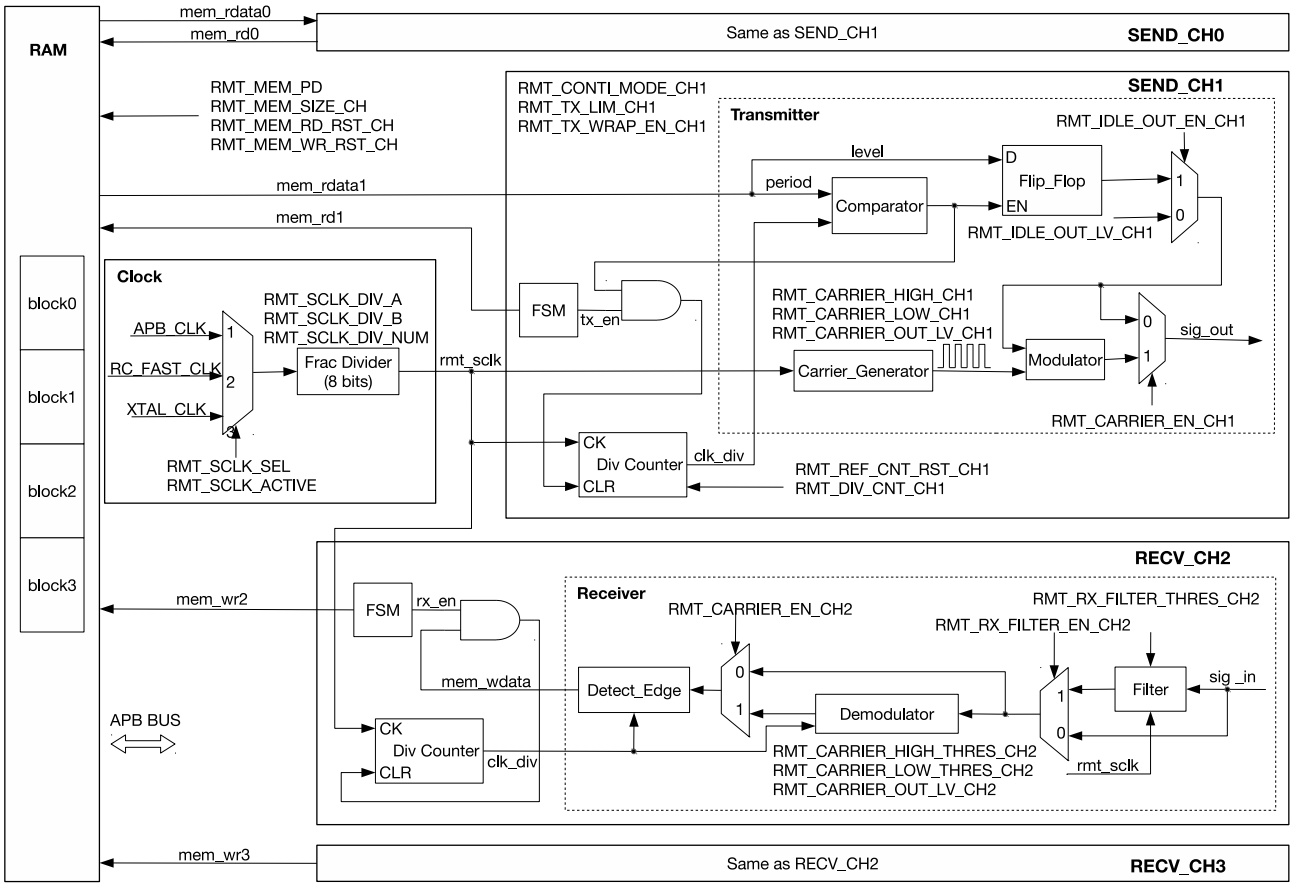



Figure 33.3-1. RMT Architecture

The RMT module has four independent channels, two of which are TX channels and the other two are RX channels. Each TX channel has its own clock-divider counter, state machine, and transmitter. Each RX channel also has its own clock-divider counter, state machine, and receiver. The four channels share a 192 x 32-bit RAM.

## 33.3.2 RMT RAM

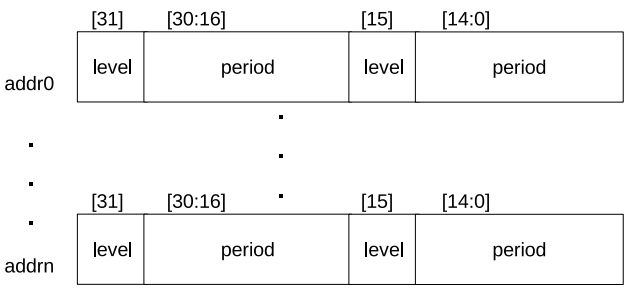



Figure 33.3-2. Format of Pulse Code in RAM

Figure 33.3-2 shows the format of pulse code in RAM. Each pulse code contains a 16-bit entry with two fields, level and period.

- Level (0 or 1): indicates a low-/high-level value was received or is going to be sent.
- Period: points out how many clk_div clock cycles the level lasts for, see Figure 33.3-1.

A zero (0) period is interpreted as a transmission end-marker. If the period is not an end-marker, its value is limited by APB clock and RMT clock:

$$3 \times T_{apb_clk} + 5 \times T_{\text{rmt_sclk}} < period \times T_{\text{clk_div}} (1)$$

The RAM is divided into four 48 x 32-bit blocks. By default, each channel uses one block, block zero for channel zero, block one for channel one, and so on.

If the data size of one single transfer is larger than this block size of TX channel *n* or RX channel *m*, users can configure the channel

- to enable wrap mode by setting RMT_MEM_TX_WRAP_EN_CH*n*/*m*.
- or to use more blocks by configuring RMT_MEM_SIZE_CH*n*/*m*.

Setting RMT_MEM_SIZE_CH*n*/*m* > 1 allows channel *n*/*m* to use the memory of subsequent channels, block (*n*/*m*) ~ block (*n*/*m* + RMT_MEM_SIZE_CH*n*/*m* -1). If so, the subsequent channels *n*/*m* + 1 ~ *n*/*m* + RMT_MEM_SIZE_CH*n*/*m* - 1 can not 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_CH0, but channel 3 can not use the blocks for channels 0, 1, or 2. Therefore, the maximum value of RMT_MEM_SIZE_CH*n* should not exceed (4 - *n*) and the maximum value of RMT_MEM_SIZE_CH*m* should not exceed (2 - *m*).

The RMT RAM can be accessed via APB bus, or read by the transmitter and written by the receiver. To avoid any possible access conflict between the receiver and the APB bus, RMT can be configured to designate the RAM block's owner, be it the receiver or the APB bus, by configuring RMT_MEM_OWNER_CH*m*. If this ownership is violated, a flag signal RMT_CH*m*_OWNER_ERR will be generated.

APB bus is able to access RAM in FIFO mode and in Direct Address (NONFIFO) mode, depending on the configuration of RMT_FIFO_MASK:

- 1: use NONFIFO mode;
- 0: use FIFO mode.

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

### 33.3.3 Clock

The clock source of RMT can be APB_CLK, RC_FAST_CLK or XTAL_CLK, depending on the configuration of RMT_SCLK_SEL. RMT clock can be enabled by setting RMT_SCLK_ACTIVE. RMT working clock (rmt_sclk) is obtained by dividing the selected clock source with a fractional divider, see Figure 33.3-1. The divider is:

$$RMT_SCLK_DIV_NUM + 1 + RMT_SCLK_DIV_A/RMT_SCLK_DIV_B$$

For more information, please check Chapter 6 *Reset and Clock*.

RMT_DIV_CNT_CH*n*/*m* is used to configure the divider coefficient of internal clock divider for RMT channels. The coefficient is normally equal to the value of RMT_DIV_CNT_CH*n*/*m*, except value 0 that represents coefficient 256. The clock divider can be reset by clearing RMT_REF_CNT_RST_CH*n*/*m*. The clock generated from the divider can be used by the counter (see Figure 33.3-1).

## 33.3.4 Transmitter

### 33.3.4.1 Normal TX Mode

When RMT_TX_START_CH*n* is set, the transmitter of channel *n* starts reading and sending pulse codes from the starting address of its RAM block. The codes are sent starting from low-address entry.

When an end-marker (a zero period) is encountered, the transmitter stops the transmission, returns to idle state and generates an RMT_CH*n*_TX_END_INT interrupt. Setting RMT_TX_STOP_CH*n* to 1 also 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 implement the above-mentioned configurations, please set RMT_CONF_UPDATE_CH*n* first. For more information, see Section 33.3.6.

### 33.3.4.2 Wrap TX Mode

To transmit more pulse codes than can be fitted in the channel's RAM, users can enable wrap TX mode by setting RMT_MEM_TX_WRAP_EN_CH*n*. In this mode, the transmitter sends the data from RAM in loops till an end-marker is encountered.

For example, if RMT_MEM_SIZE_CH*n* = 1, the transmitter starts sending data from the address 48 * *n*, and then the data from higher RAM address. Once the transmitter finishes sending the data from (48 * (*n* + 1) - 1), it continues sending data from 48 * *n* again till an end-marker is encountered. Wrap mode is also applicable for RMT_MEM_SIZE_CH*n* > 1.

When the size of transmitted pulse codes is larger than or equal to the value set by RMT_TX_LIM_CH*n*, an RMT_CH*n*_TX_THR_EVENT_INT interrupt is triggered. 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 by software, the already used RAM region can be updated with new pulse codes. In this way the transmitter can seamlessly send unlimited pulse codes in wrap mode.

To update the configuration of RMT_MEM_TX_WRAP_EN_CH*n*, RMT_MEM_SIZE_CH*n*, and RMT_TX_LIM_CH*n*, please set RMT_CONF_UPDATE_CH*n* first. For more information, see Section 33.3.6.

### 33.3.4.3 TX Modulation

Transmitter output can be modulated with a carrier wave by setting RMT_CARRIER_EN_CH*n*. The carrier waveform is configurable.

In a carrier cycle, high level lasts for (RMT_CARRIER_HIGH_CH*n* + 1) rmt_sclk cycles, while low level lasts for (RMT_CARRIER_LOW_CH*n* + 1) rmt_sclk cycles. When RMT_CARRIER_OUT_LV_CH*n* is set, carrier wave is added on the high-level of output signals; while RMT_CARRIER_OUT_LV_CH*n* is cleared, carrier wave is added on the low-level of output signals.

Carrier wave can be added on all output signals during modulation, or just added on valid pulse codes (the data stored in RAM), depending on the configuration of RMT_CARRIER_EFF_EN_CH*n*:

- 0: add carrier wave on all output signals;
- 1: add carrier wave only on valid signals.

To implement the modulation configuration, please set RMT_CONF_UPDATE_CH*n* first. For more information, see Section 33.3.6.

#### 33.3.4.4 Continuous TX Mode

This continuous TX mode can be enabled by setting RMT_TX_CONTI_MODE_CH*n*. In this mode, the transmitter sends the pulse codes from RAM in loops.

- If an end-marker is encountered, the transmitter starts transmitting the first data again.
- If no end-marker is encountered, the transmitter starts transmitting the first data again after the last data is transmitted.

If RMT_TX_LOOP_CNT_EN_CH*n* is set, the loop counting is incremented by 1 each time an end-marker is encountered. If the counting reaches the value set in RMT_TX _LOOP_NUM_CH*n*, an RMT_CH*n*_TX_LOOP_INT is generated.

In an end-marker, if its period[14:0] is 0, then the period of the previous data must satisfy the following requirement:

$$6 \times T_{apb_clk} + 12 \times T_{rmt_sclk} < period \times T_{clk_div} (2)$$

The period of the other data only need to satisfy relation (1).

To implement the above-mentioned configuration, please set RMT_CONF_UPDATE_CH*n* first. For more information, see Section 33.3.6.

#### 33.3.4.5 Simultaneous TX Mode

RMT module supports multiple channels transmitting data simultaneously. To use this function, follow the steps below.

1. Configure RMT_TX_SIM_CH*n* to choose which multiple channels are used to transmit data simultaneously.
2. Set RMT_TX_SIM_EN to enable this transmission mode.
3. Set RMT_TX_START_CH*n* for each selected channel, to start data transmitting.

Once the last channel is configured, these channels start transmitting data simultaneously. Due to hardware limitations, there is no guarantee that two channels can start sending data exactly at the same time. The interval between two channels starting transmitting data is within 3 x $T_{clk_div}$.

To configure RMT_TX_SIM_EN, please set RMT_CONF_UPDATE_CH*n* first. For more information, see Section 33.3.6.

### 33.3.5 Receiver

#### 33.3.5.1 Normal RX Mode

The receiver of channel *m* is controlled by RMT_RX_EN_CH*m*:

- RMT_RX_EN_CH*m* = 1, the receiver starts working.
- RMT_RX_EN_CH*m* = 0, the receiver stops receiving data.

When the receiver becomes active, it starts counting from the first edge of the signal, detecting signal levels and counting clock cycles the level lasts for. Each cycle count is then written back to RAM.

When the receiver detects no change in a signal level for a number of clock cycles more than the value set by RMT_IDLE_THRES_CH*m*, the receiver will stop receiving data, return to idle state, and generate an RMT_CH*m*_RX_END_INT interrupt.

Please note that RMT_IDLE_THRES_CH*m* should be configured to a maximum value according to your application, otherwise a valid received level may be mistaken as a level in idle state.

If RAM block of this RX channel is used up by the received data, the receiver will stop receiving data, and generate an RMT_CH*m*_ERR_INT interrupt triggered by RAM FULL event.

To implement configuration above, please set RMT_CONF_UPDATE_CH*m* first. For more information, see Section 33.3.6.

#### 33.3.5.2 Wrap RX Mode

To receive more pulse codes than can be fitted in the channel's RAM, users can enable wrap RX mode for channel *m* by configuring RMT_MEM_RX_WRAP_EN_CH*m*. In wrap mode, the receiver stores the received data to RAM block of this channel in loops.

Receiving ends, when the receiver detects no change in a signal level for a number of clock cycles more than the value set by RMT_IDLE_THRES_CH*m*. The receiver then returns to idle state and generates an RMT_CH*m*_RX_END_INT interrupt.

For example, if RMT_MEM_SIZE_CH*m* is set to 1, the receiver starts receiving data and stores the data to address 48 * *m*, and then to higher RAM address. When the receiver finishes storing the received data to address (48 * (*m* + 1) - 1), the receiver continues receiving data and storing data to the address 48 * *m* again, till no change is detected on a signal level for more than RMT_IDLE_THRES_CH*m* clock cycles. Wrap mode is also applicable for RMT_MEM_SIZE_CH*m* > 1.

An RMT_CH*m*_RX_THR_EVENT_INT is generated when the size of received pulse codes is larger than or equal to the value set by RMT_RX_LIM_CH*m*. In wrap mode, RMT_RX_LIM_CH*M* can be set to a half or a fraction of the size of the channel's RAM block. When an RMT_CH*m*_RX_THR_EVENT_INT interrupt is detected by software, the system will be notified to copy out data stored in already used RMT RAM region, and then the region can be updated by subsequent data. In this way an arbitrary amount of data can be seamlessly received.

To implement the configuration above, please set RMT_CONF_UPDATE_CH*m* first. For more information, see Section 33.3.6.

#### 33.3.5.3 RX Filtering

Users can enable the receiver to filter input signals by setting RMT_RX_FILTER_EN_CH*m* for each channel. The filter samples input signals continuously, and detects the signals which remain unchanged for a

continuous RMT_RX_FILTER_THRES_CH*m* rmt_sclk cycles as valid, otherwise, the signals are rejected. Only the valid signals can pass through this filter. The filter removes pulses with a length of less than RMT_RX_FILTER_THRES_CH*n* rmt_sclk cycles.

To implement the configuration above, please set RMT_CONF_UPDATE_CH*m* first. For more information, see Section 33.3.6.

#### 33.3.5.4 RX Demodulation

Users can enable demodulation function on input signals or on filtered output signals by setting RMT_CARRIER_EN_CH*m*. RX demodulation can be applied to high-level carrier wave or low-level carrier wave, depending on the configuration of RMT_CARRIER_OUT_LV_CH*m*:

- 1: demodulate high-level carrier wave
- 0: demodulate low-level carrier wave

Users can configure RMT_CARRIER_HIGH_THRES_CH*m* and RMT_CARRIER_LOW_THRES_CH*m* to set the thresholds to demodulate high-level carrier wave or low-level carrier wave.

If the high-level of a signal lasts for less than RMT_CARRIER_HIGH_THRES_CH*m* clk_div cycles, or the low-level lasts for less than RMT_CARRIER_LOW_THRES_CH*m* clk_div cycles, such level is detected as a carrier wave and then is filtered out.

To implement the configuration above, please set RMT_CONF_UPDATE_CH*m* first. For more information, see Section 33.3.6.

### 33.3.6 Configuration Update

To update RMT registers configuration, please set RMT_CONF_UPDATE_CH*n*/*m* for each channel first.

All the bits/fields listed in the second column of Table 33.3-1 should follow this rule.

Table 33.3-1. Configuration Update

| Register | Bit/Field Configuration Update |
|---|---|
| TX Channels | |
| RMT_CH*n*CONFO_REG | RMT_CARRIER_OUT_LV_CH*n* |
| | RMT_CARRIER_EN_CH*n* |
| | RMT_CARRIER_EFF_EN_CH*n* |
| | RMT_DIV_CNT_CH*n* |
| | RMT_TX_STOP_CH*n* |
| | RMT_IDLE_OUT_EN_CH*n* |
| | RMT_IDLE_OUT_LV_CH*n* |
| | RMT_TX_CONTI_MODE_CH*n* |
| RMT_CH*n*CARRIER_DUTY_REG | RMT_CARRIER_HIGH_CH*n* |
| | RMT_CARRIER_LOW_CH*n* |
| RMT_CH*n*_TX_LIM_REG | RMT_TX_LOOP_CNT_EN_CH*n* |
| | RMT_TX_LOOP_NUM_CH*n* |
| | RMT_TX_LIM_CH*n* |
| RMT_CH*n*_TX_SIM_REG | RMT_TX_SIM_EN |

Cont'd on next page

Table 33.3-1 – cont'd from previous page

| Register | Bit/Field Configuration Update |
|---|---|
| RX Channels | |
| RMT_CH*m*CONF0_REG | RMT_CARRIER_OUT_LV_CH*m* |
| | RMT_CARRIER_EN_CH*m* |
| | RMT_IDLE_THRES_CH*m* |
| | RMT_DIV_CNT_CH*m* |
| RMT_CH*m*CONF1_REG | RMT_RX_FILTER_THRES_CH*m* |
| | RMT_RX_EN_CH*m* |
| RMT_CH*m*_RX_CARRIER_RM_REG | RMT_CARRIER_HIGH_THRES_CH*m* |
| | RMT_CARRIER_LOW_THRES_CH*m* |
| RMT_CH*m*_RX_LIM_REG | RMT_RX_LIM_CH*m* |
| RMT_REF_CNT_RST_REG | RMT_REF_CNT_RST_CH*m* |

### 33.3.7 Interrupts

- RMT_CH*n*/*m*_ERR_INT: triggered when channel *n*/*m* 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_CH*n*_TX_LIM_REG.
- RMT_CH*m*_RX_THR_EVENT_INT: triggered each time when the amount of data received by the receiver reaches the value set in RMT_CH*m*_RX_LIM_REG.
- RMT_CH*n*_TX_END_INT: Triggered when the transmitter has finished transmitting signals.
- RMT_CH*m*_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*.

## 33.4 Register Summary

The addresses in this section are relative to RMT base address provided in Table 3.3-3 in Chapter 3 *System and Memory* .

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

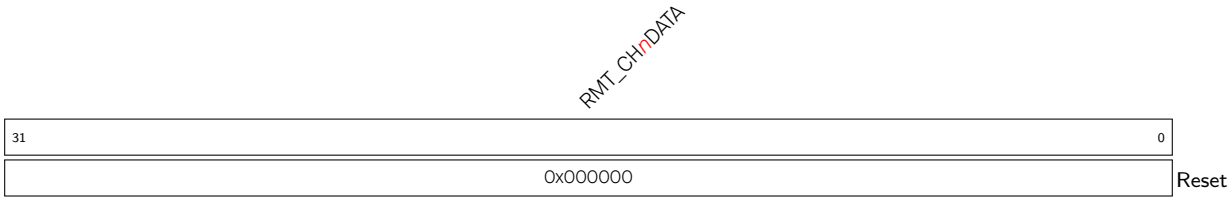
| Name | Description | Address | Access |
|---|---|---|---|
| **FIFO R/W Registers** | | | |
| RMT_CH0DATA_REG | The read and write data register for channel 0 by APB FIFO access. | 0x0000 | RO |
| RMT_CH1DATA_REG | The read and write data register for channel 1 by APB FIFO access. | 0x0004 | RO |
| RMT_CH2DATA_REG | The read and write data register for channel 2 by APB FIFO access. | 0x0008 | RO |
| RMT_CH3DATA_REG | The read and write data register for channel 3 by APB FIFO access. | 0x000C | RO |
| **Configuration Registers** | | | |
| RMT_CH0CONF0_REG | Configuration register 0 for channel 0 | 0x0010 | varies |
| RMT_CH1CONF0_REG | Configuration register 0 for channel 1 | 0x0014 | varies |
| RMT_CH2CONF0_REG | Configuration register 0 for channel 2 | 0x0018 | R/W |
| RMT_CH2CONF1_REG | Configuration register 1 for channel 2 | 0x001C | varies |
| RMT_CH3CONF0_REG | Configuration register 0 for channel 3 | 0x0020 | R/W |
| RMT_CH3CONF1_REG | Configuration register 1 for channel 3 | 0x0024 | varies |
| RMT_SYS_CONF_REG | Configuration register for RMT APB | 0x0068 | R/W |
| RMT_REF_CNT_RST_REG | Reset register for RMT clock divider | 0x0070 | WT |
| **Status Registers** | | | |
| RMT_CH0STATUS_REG | Channel 0 status register | 0x0028 | RO |
| RMT_CH1STATUS_REG | Channel 1 status register | 0x002C | RO |
| RMT_CH2STATUS_REG | Channel 2 status register | 0x0030 | RO |
| RMT_CH3STATUS_REG | Channel 3 status register | 0x0034 | RO |
| **Interrupt Registers** | | | |
| RMT_INT_RAW_REG | Raw interrupt status | 0x0038 | R/WTC/SS |
| RMT_INT_ST_REG | Masked interrupt status | 0x003C | RO |
| RMT_INT_ENA_REG | Interrupt enable bits | 0x0040 | R/W |
| RMT_INT_CLR_REG | Interrupt clear bits | 0x0044 | WT |
| **Carrier Wave Duty Cycle Registers** | | | |
| RMT_CH0CARRIER_DUTY_REG | Duty cycle configuration register for channel 0 | 0x0048 | R/W |
| RMT_CH1CARRIER_DUTY_REG | Duty cycle configuration register for channel 1 | 0x004C | R/W |
| RMT_CH2_RX_CARRIER_RM_REG | Carrier remove register for channel 2 | 0x0050 | R/W |
| RMT_CH3_RX_CARRIER_RM_REG | Carrier remove register for channel 3 | 0x0054 | R/W |
| **TX Event Configuration Registers** | | | |
| RMT_CH0_TX_LIM_REG | Configuration register for channel 0 TX event | 0x0058 | varies |
| RMT_CH1_TX_LIM_REG | Configuration register for channel 1 TX event | 0x005C | varies |
| RMT_TX_SIM_REG | RMT TX synchronous register | 0x006C | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| **RX Event Configuration Registers** | | | |
| RMT_CH2_RX_LIM_REG | Configuration register for channel 2 RX event | 0x0060 | R/W |
| RMT_CH3_RX_LIM_REG | Configuration register for channel 3 RX event | 0x0064 | R/W |
| **Version Register** | | | |
| RMT_DATE_REG | Version control register | 0x00CC | R/W |

## 33.5 Registers

The addresses in this section are relative to RMT base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 33.1. RMT_CH*n*DATA_REG (*n* = 0, 1) (0x0000, 0x0004)**




**RMT_CH*n*DATA** Read and write data for channel *n* via APB FIFO. (RO)

**Register 33.2. RMT_CH*m*DATA_REG (*m* = 2, 3) (0x0008, 0x000C)**

**RMT_CH*m*DATA** Read and write data for channel *m* via APB FIFO. (RO)

**Register 33.3. RMT_CH*n*CONF0_REG (*n* = 0, 1) (0x0010, 0x0014)**

| (reserved) | RMT_CONF_UPDATE_CH*n* | (reserved) | RMT_CARRIER_OUT_LV_CH*n* | RMT_CARRIER_EN_CH*n* | RMT_CARRIER_EFF_EN_CH*n* | (reserved) | RMT_MEM_SIZE_CH*n* | RMT_DIV_CNT_CH*n* | RMT_TX_STOP_CH*n* | RMT_IDLE_OUT_EN_CH*n* | RMT_IDLE_OUT_LV_CH*n* | RMT_MEM_TX_WRAP_EN_CH*n* | RMT_TX_CONTI_MODE_CH*n* | RMT_APB_MEM_RST_CH*n* | RMT_MEM_RD_RST_CH*n* | RMT_TX_START_CH*n* | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 … 16 | 15 … 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 0 0 0 0 0 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0x1 | 0x2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**RMT_TX_START_CH*n*** Set this bit to start sending data in channel *n*. (WT)

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

**RMT_APB_MEM_RST_CH*n*** Set this bit to reset RAM W/R address accessed by APB FIFO for channel *n*. (WT)

**RMT_TX_CONTI_MODE_CH*n*** Set this bit to enable continuous TX mode for channel *n*. (R/W)

In this mode, the transmitter starts its transmission from the first data, and in the following transmission:

- if an end-marker is encountered, the transmitter starts transmitting data from the first data again;
- if no end-marker is encountered, the transmitter starts transmitting the first data again when the last data is transmitted.

**RMT_MEM_TX_WRAP_EN_CH*n*** Set this bit to enable wrap TX mode for channel *n*. In this mode, if the TX data size is larger than the channel's RAM block size, the transmitter continues transmitting the first data to the last data in loops. (R/W)

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

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

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

Continued on the next page...

**Register 33.3. RMT_CH*n*CONF0_REG (*n* = 0, 1) (0x0010, 0x0014)**

**Continued from the previous page...**

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

**RMT_MEM_SIZE_CH*n*** This register is used to configure the maximum number of memory blocks allocated to channel *n*. (R/W)

**RMT_CARRIER_EFF_EN_CH*n*** 1: Add carrier modulation on the output signal only at data-sending state for channel *n*. 0: Add carrier modulation on the output signal at data-sending state and idle state for channel *n*. Only valid when RMT_CARRIER_EN_CH*n* is 1. (R/W)

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

**RMT_CARRIER_OUT_LV_CH*n*** This bit is used to configure the position of carrier wave for channel *n*. (R/W)

1'h0: add carrier wave on low level.

1'h1: add carrier wave on high level.

**RMT_CONF_UPDATE_CH*n*** Synchronization bit for channel *n* (WT)

**Register 33.4. RMT_CH*m*CONF0_REG (*m* = 2, 3) (0x0018, 0x0020)**

| (reserved) | RMT_CARRIER_OUT_LV_CH*m* | RMT_CARRIER_EN_CH*m* | (reserved) | RMT_MEM_SIZE_CH*m* | RMT_IDLE_THRES_CH*m* | RMT_DIV_CNT_CH*m* | |
|---|---|---|---|---|---|---|---|
| 31 30 | 29 | 28 | 27 26 | 25 23 | 22 8 | 7 0 | |
| 0 0 | 1 | 1 | 0 0 | 0x1 | 0x7fff | 0x2 | Reset |

**RMT_DIV_CNT_CH*m*** This field is used to configure the clock divider of channel *m*. (R/W)

**RMT_IDLE_THRES_CH*m*** This field is used to configure RX threshold. When no edge is detected on the input signal for continuous clock cycles longer than this field value, the receiver stops receiving data. (R/W)

**RMT_MEM_SIZE_CH*m*** This field is used to configure the maximum number of memory blocks allocated to channel *m*. (R/W)

**RMT_CARRIER_EN_CH*m*** This is the carrier modulation enable-bit for channel *m*. 1: Add carrier modulation on output signal. 0: No carrier modulation is added on output signal. (R/W)

**RMT_CARRIER_OUT_LV_CH*m*** This bit is used to configure the position of carrier wave for channel *m*. (R/W)

1'h0: add carrier wave on low level.

1'h1: add carrier wave on high level.

**Register 33.5. RMT_CH*m*CONF1_REG(*m* = 2, 3) (0x001C, 0x0024)**

| (reserved) | RMT_CONF_UPDATE_CH*m* | (reserved) | RMT_MEM_RX_WRAP_EN_CH*m* | RMT_RX_FILTER_THRES_CH*m* | RMT_RX_FILTER_EN_CH*m* | RMT_MEM_OWNER_CH*m* | RMT_APB_MEM_RST_CH*m* | RMT_MEM_WR_RST_CH*m* | RMT_RX_EN_CH*m* | |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 16 | 15 | 14 | 13 | 12 … 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 | 0xf | 0 | 1 | 0 | 0 | 0 | Reset |

**RMT_RX_EN_CH*m*** Set this bit to enable the receiver to start receiving data in channel *m*. (R/W)

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

**RMT_APB_MEM_RST_CH*m*** Set this bit to reset RAM W/R address accessed by APB FIFO for channel *m*. (WT)

**RMT_MEM_OWNER_CH*m*** This bit marks the ownership of channel *m*'s RAM block. (R/W/SC)

1'h1: Receiver is using the RAM.

1'h0: APB bus is using the RAM.

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

**RMT_RX_FILTER_THRES_CH*m*** When receiving data, the receiver ignores the input pulse when its width is shorter than this register value in units of rmt_sclk cycles. (R/W)

**RMT_MEM_RX_WRAP_EN_CH*m*** Set this bit to enable wrap RX mode for channel *m*. In this mode, if the RX data size is larger than channel *m*'s RAM block size, the receiver stores the RX data from the first address to the last address in loops. (R/W)

**RMT_CONF_UPDATE_CH*m*** Synchronization bit for channel *m*. (WT)

**Register 33.6. RMT_SYS_CONF_REG (0x0068)**

| RMT_CLK_EN | (reserved) | RMT_SCLK_ACTIVE | RMT_SCLK_SEL | RMT_SCLK_DIV_B | RMT_SCLK_DIV_A | RMT_SCLK_DIV_NUM | RMT_MEM_FORCE_PU | RMT_MEM_FORCE_PD | RMT_MEM_CLK_FORCE_ON | RMT_APB_FIFO_MASK | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 27 | 26 | 25 24 | 23 18 | 17 12 | 11 4 | 3 | 2 | 1 | 0 | |
| 0 | 0 0 0 0 | 1 | 0x1 | 0x0 | 0x0 | 0x1 | 0 | 0 | 0 | 0 | Reset |

**RMT_APB_FIFO_MASK** 1'h1: Access memory directly. 1'h0: Access memory by FIFO. (R/W)

**RMT_MEM_CLK_FORCE_ON** Set this bit to enable the clock for RMT memory. (R/W)

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

**RMT_MEM_FORCE_PU** 1: Disable the power-down function of RMT memory in Light-sleep. 0: Power down RMT memory when RMT is in Light-sleep mode. (R/W)

**RMT_SCLK_DIV_NUM** The integral part of the fractional divider. (R/W)

**RMT_SCLK_DIV_A** The numerator of the fractional part of the fractional divider. (R/W)

**RMT_SCLK_DIV_B** The denominator of the fractional part of the fractional divider. (R/W)

**RMT_SCLK_SEL** Choose the clock source of rmt_sclk. 1: APB_CLK; 2: RC_FAST_CLK; 3: XTAL_CLK. (R/W)

**RMT_SCLK_ACTIVE** rmt_sclk switch. (R/W)

**RMT_CLK_EN** The enable signal of RMT register clock gate. 1: Power up the drive clock of registers. 0: Power down the drive clock of registers. (R/W)

**Register 33.7. RMT_REF_CNT_RST_REG (0x0070)**

| (reserved) | RMT_REF_CNT_RST_CH3 | RMT_REF_CNT_RST_CH2 | RMT_REF_CNT_RST_CH1 | RMT_REF_CNT_RST_CH0 | |
|---|---|---|---|---|---|
| 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 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0 | Reset |

**RMT_REF_CNT_RST_CH0** This bit is used to reset the clock divider of channel 0. (WT)

**RMT_REF_CNT_RST_CH1** This bit is used to reset the clock divider of channel 1. (WT)

**RMT_REF_CNT_RST_CH2** This bit is used to reset the clock divider of channel 2. (WT)

**RMT_REF_CNT_RST_CH3** This bit is used to reset the clock divider of channel 3. (WT)

**Register 33.8. RMT_CH*n*STATUS_REG (*n* = 0, 1) (0x0028, 0x002C)**

| RMT_APB_MEM_RADDR_CH*n* | RMT_APB_MEM_WR_ERR_CH*n* | RMT_MEM_EMPTY_CH*n* | RMT_APB_MEM_RD_ERR_CH*n* | RMT_APB_MEM_WADDR_CH*n* | RMT_STATE_CH*n* | RMT_MEM_RADDR_EX_CH*n* | |
|---|---|---|---|---|---|---|---|
| 31 – 24 | 23 | 22 | 21 | 20 – 12 | 11 – 9 | 8 – 0 | |
| 0x0 | 0 | 0 | 0 | 0 | 0 | 0 | Reset |

**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_APB_MEM_WADDR_CH*n*** This field records the memory address offset when writes RAM over 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 (overflows) when reads RAM via APB bus. (RO)

**RMT_MEM_EMPTY_CH*n*** This status bit will be set when the TX data size is larger than the memory size and the wrap TX 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 (overflows) when writes via APB bus. (RO)

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

**Register 33.9. RMT_CH*m*STATUS_REG (*m* = 2, 3) (0x0030, 0x0034)**

| (reserved) | RMT_APB_MEM_RD_ERR_CH*m* | RMT_MEM_FULL_CH*m* | RMT_MEM_OWNER_ERR_CH*m* | RMT_STATE_CH*m* | (reserved) | RMT_APB_MEM_RADDR_CH*m* | (reserved) | RMT_MEM_WADDR_EX_CH*m* | |
|---|---|---|---|---|---|---|---|---|---|
| 31 28 | 27 | 26 | 25 | 24 22 | 21 | 20 12 | 11 9 | 8 0 | |
| 0 0 0 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 0 0 | 0 | Reset |

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

**RMT_APB_MEM_RADDR_CH*m*** This field records the memory address offset when reads RAM over APB bus. (RO)

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

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

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

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

**Register 33.10. RMT_INT_RAW_REG (0x0038)**

| (reserved) | RMT_CH1_TX_LOOP_INT_RAW | RMT_CH0_TX_LOOP_INT_RAW | RMT_CH3_RX_THR_EVENT_INT_RAW | RMT_CH2_RX_THR_EVENT_INT_RAW | RMT_CH1_TX_THR_EVENT_INT_RAW | RMT_CH0_TX_THR_EVENT_INT_RAW | RMT_CH3_ERR_INT_RAW | RMT_CH2_ERR_INT_RAW | RMT_CH1_ERR_INT_RAW | RMT_CH0_ERR_INT_RAW | RMT_CH3_RX_END_INT_RAW | RMT_CH2_RX_END_INT_RAW | RMT_CH1_TX_END_INT_RAW | RMT_CH0_TX_END_INT_RAW | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 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 | Reset | |

**RMT_CH0_TX_END_INT_RAW** The interrupt raw bit of RMT_CH0_TX_END_INT. (R/WTC/SS)

**RMT_CH1_TX_END_INT_RAW** The interrupt raw bit of RMT_CH1_TX_END_INT. (R/WTC/SS)

**RMT_CH2_RX_END_INT_RAW** The interrupt raw bit of RMT_CH2_RX_END_INT. (R/WTC/SS)

**RMT_CH3_RX_END_INT_RAW** The interrupt raw bit of RMT_CH3_RX_END_INT. (R/WTC/SS)

**RMT_CH0_ERR_INT_RAW** The interrupt raw bit of RMT_CH0_ERR_INT. (R/WTC/SS)

**RMT_CH1_ERR_INT_RAW** The interrupt raw bit of RMT_CH1_ERR_INT. (R/WTC/SS)

**RMT_CH2_ERR_INT_RAW** The interrupt raw bit of RMT_CH2_ERR_INT. (R/WTC/SS)

**RMT_CH3_ERR_INT_RAW** The interrupt raw bit of RMT_CH3_ERR_INT. (R/WTC/SS)

**RMT_CH0_TX_THR_EVENT_INT_RAW** The interrupt raw bit of RMT_CH0_TX_THR_EVENT_INT. (R/WTC/SS)

**RMT_CH1_TX_THR_EVENT_INT_RAW** The interrupt raw bit of RMT_CH0_TX_THR_EVENT_INT. (R/WTC/SS)

**RMT_CH2_RX_THR_EVENT_INT_RAW** The interrupt raw bit of RMT_CH2_RX_THR_EVENT_INT. (R/WTC/SS)

**RMT_CH3_RX_THR_EVENT_INT_RAW** The interrupt raw bit of RMT_CH3_RX_THR_EVENT_INT. (R/WTC/SS)

**RMT_CH0_TX_LOOP_INT_RAW** The interrupt raw bit of RMT_CH0_TX_LOOP_INT. (R/WTC/SS)

**RMT_CH1_TX_LOOP_INT_RAW** The interrupt raw bit of RMT_CH1_TX_LOOP_INT. (R/WTC/SS)

**Register 33.11. RMT_INT_ST_REG (0x003C)**

| 31 | | | | | | | | | | | | | | | | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | RMT_CH1_TX_LOOP_INT_ST | RMT_CH0_TX_LOOP_INT_ST | RMT_CH3_RX_THR_EVENT_INT_ST | RMT_CH2_RX_THR_EVENT_INT_ST | RMT_CH1_TX_THR_EVENT_INT_ST | RMT_CH0_TX_THR_EVENT_INT_ST | RMT_CH3_ERR_INT_ST | RMT_CH2_ERR_INT_ST | RMT_CH1_ERR_INT_ST | RMT_CH0_ERR_INT_ST | RMT_CH3_RX_END_INT_ST | RMT_CH2_RX_END_INT_ST | RMT_CH1_TX_END_INT_ST | RMT_CH0_TX_END_INT_ST | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**RMT_CH0_TX_END_INT_ST** The masked interrupt status bit for RMT_CH0_TX_END_INT. (RO)

**RMT_CH1_TX_END_INT_ST** The masked interrupt status bit for RMT_CH1_TX_END_INT. (RO)

**RMT_CH2_RX_END_INT_ST** The masked interrupt status bit for RMT_CH2_RX_END_INT. (RO)

**RMT_CH3_RX_END_INT_ST** The masked interrupt status bit for RMT_CH3_RX_END_INT. (RO)

**RMT_CH0_ERR_INT_ST** The masked interrupt status bit for RMT_CH0_ERR_INT. (RO)

**RMT_CH1_ERR_INT_ST** The masked interrupt status bit for RMT_CH1_ERR_INT. (RO)

**RMT_CH2_ERR_INT_ST** The masked interrupt status bit for RMT_CH2_ERR_INT. (RO)

**RMT_CH3_ERR_INT_ST** The masked interrupt status bit for RMT_CH3_ERR_INT. (RO)

**RMT_CH0_TX_THR_EVENT_INT_ST** The masked interrupt status bit for RMT_CH0_TX_THR_EVENT_INT. (RO)

**RMT_CH1_TX_THR_EVENT_INT_ST** The masked interrupt status bit for RMT_CH1_TX_THR_EVENT_INT. (RO)

**RMT_CH2_RX_THR_EVENT_INT_ST** The masked interrupt status bit for RMT_CH2_RX_THR_EVENT_INT. (RO)

**RMT_CH3_RX_THR_EVENT_INT_ST** The masked interrupt status bit for RMT_CH3_RX_THR_EVENT_INT. (RO)

**RMT_CH0_TX_LOOP_INT_ST** The masked interrupt status bit for RMT_CH0_TX_LOOP_INT. (RO)

**RMT_CH1_TX_LOOP_INT_ST** The masked interrupt status bit for RMT_CH1_TX_LOOP_INT. (RO)

**Register 33.12. RMT_INT_ENA_REG (0x0040)**

| (reserved) | RMT_CH1_TX_LOOP_INT_ENA | RMT_CH0_TX_LOOP_INT_ENA | RMT_CH3_RX_THR_EVENT_INT_ENA | RMT_CH2_RX_THR_EVENT_INT_ENA | RMT_CH1_TX_THR_EVENT_INT_ENA | RMT_CH0_TX_THR_EVENT_INT_ENA | RMT_CH3_ERR_INT_ENA | RMT_CH2_ERR_INT_ENA | RMT_CH1_ERR_INT_ENA | RMT_CH0_ERR_INT_ENA | RMT_CH3_RX_END_INT_ENA | RMT_CH2_RX_END_INT_ENA | RMT_CH1_TX_END_INT_ENA | RMT_CH0_TX_END_INT_ENA | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 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 | Reset |

**RMT_CH0_TX_END_INT_ENA** The interrupt enable bit for RMT_CH0_TX_END_INT. (R/W)

**RMT_CH1_TX_END_INT_ENA** The interrupt enable bit for RMT_CH1_TX_END_INT. (R/W)

**RMT_CH2_RX_END_INT_ENA** The interrupt enable bit for RMT_CH2_RX_END_INT. (R/W)

**RMT_CH3_RX_END_INT_ENA** The interrupt enable bit for RMT_CH3_RX_END_INT. (R/W)

**RMT_CH0_ERR_INT_ENA** The interrupt enable bit for RMT_CH0_ERR_INT. (R/W)

**RMT_CH1_ERR_INT_ENA** The interrupt enable bit for RMT_CH1_ERR_INT. (R/W)

**RMT_CH2_ERR_INT_ENA** The interrupt enable bit for RMT_CH2_ERR_INT. (R/W)

**RMT_CH3_ERR_INT_ENA** The interrupt enable bit for RMT_CH3_ERR_INT. (R/W)

**RMT_CH0_TX_THR_EVENT_INT_ENA** The interrupt enable bit for RMT_CH0_TX_THR_EVENT_INT. (R/W)

**RMT_CH1_TX_THR_EVENT_INT_ENA** The interrupt enable bit for RMT_CH1_TX_THR_EVENT_INT. (R/W)

**RMT_CH2_RX_THR_EVENT_INT_ENA** The interrupt enable bit for RMT_CH2_RX_THR_EVENT_INT. (R/W)

**RMT_CH3_RX_THR_EVENT_INT_ENA** The interrupt enable bit for RMT_CH3_RX_THR_EVENT_INT. (R/W)

**RMT_CH0_TX_LOOP_INT_ENA** The interrupt enable bit for RMT_CH0_TX_LOOP_INT. (R/W)

**RMT_CH1_TX_LOOP_INT_ENA** The interrupt enable bit for RMT_CH1_TX_LOOP_INT. (R/W)

**Register 33.13. RMT_INT_CLR_REG (0x0044)**

| 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 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (reserved) | | | | | | | | | | | | | | | | | | RMT_CH1_TX_LOOP_INT_CLR | RMT_CH0_TX_LOOP_INT_CLR | RMT_CH3_RX_THR_EVENT_INT_CLR | RMT_CH2_RX_THR_EVENT_INT_CLR | RMT_CH1_TX_THR_EVENT_INT_CLR | RMT_CH0_TX_THR_EVENT_INT_CLR | RMT_CH3_ERR_INT_CLR | RMT_CH2_ERR_INT_CLR | RMT_CH1_ERR_INT_CLR | RMT_CH0_ERR_INT_CLR | RMT_CH3_RX_END_INT_CLR | RMT_CH2_RX_END_INT_CLR | RMT_CH1_TX_END_INT_CLR | RMT_CH0_TX_END_INT_CLR | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 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 |

**RMT_CH0_TX_END_INT_CLR** Set this bit to clear the RMT_CH0_TX_END_INT interrupt. (WT)

**RMT_CH1_TX_END_INT_CLR** Set this bit to clear the RMT_CH1_TX_END_INT interrupt. (WT)

**RMT_CH2_RX_END_INT_CLR** Set this bit to clear the RMT_CH2_RX_END_IN interrupt. (WT)

**RMT_CH3_RX_END_INT_CLR** Set this bit to clear the RMT_CH3_RX_END_IN interrupt. (WT)

**RMT_CH0_ERR_INT_CLR** Set this bit to clear the RMT_CH0_ERR_INT interrupt. (WT)

**RMT_CH1_ERR_INT_CLR** Set this bit to clear the RMT_CH1_ERR_INT interrupt. (WT)

**RMT_CH2_ERR_INT_CLR** Set this bit to clear the RMT_CH2_ERR_INT interrupt. (WT)

**RMT_CH3_ERR_INT_CLR** Set this bit to clear the RMT_CH3_ERR_INT interrupt. (WT)

**RMT_CH0_TX_THR_EVENT_INT_CLR** Set this bit to clear the RMT_CH0_TX_THR_EVENT_INT interrupt. (WT)

**RMT_CH1_TX_THR_EVENT_INT_CLR** Set this bit to clear the RMT_CH1_TX_THR_EVENT_INT interrupt. (WT)

**RMT_CH2_RX_THR_EVENT_INT_CLR** Set this bit to clear the RMT_CH2_RX_THR_EVENT_INT interrupt. (WT)

**RMT_CH3_RX_THR_EVENT_INT_CLR** Set this bit to clear the RMT_CH3_RX_THR_EVENT_INT interrupt. (WT)

**RMT_CH0_TX_LOOP_INT_CLR** Set this bit to clear the RMT_CH0_TX_LOOP_INT interrupt. (WT)

**RMT_CH1_TX_LOOP_INT_CLR** Set this bit to clear the RMT_CH1_TX_LOOP_INT interrupt. (WT)

**Register 33.14. RMT_CH*n*CARRIER_DUTY_REG (*n* = 0, 1) (0x0048, 0x004C)**

| RMT_CARRIER_HIGH_CH*n* (31–16) | RMT_CARRIER_LOW_CH*n* (15–0) | |
|---|---|---|
| 0x40 | 0x40 | Reset |

**RMT_CARRIER_LOW_CH*n*** This field is used to configure carrier wave's low level clock period for channel *n*. (R/W)

**RMT_CARRIER_HIGH_CH*n*** This field is used to configure carrier wave's high level clock period for channel *n*. (R/W)

**Register 33.15. RMT_CH*m*_RX_CARRIER_RM_REG (*m* = 2, 3) (0x0050, 0x0054)**

| RMT_CARRIER_HIGH_THRES_CH*m* (31–16) | RMT_CARRIER_LOW_THRES_CH*m* (15–0) | |
|---|---|---|
| 0x00 | 0x00 | Reset |

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

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

**Register 33.16. RMT_CH*n*_TX_LIM_REG (*n* = 0, 1) (0x0058, 0x005C)**

| (reserved) | RMT_LOOP_COUNT_RESET_CH*n* | RMT_TX_LOOP_CNT_EN_CH*n* | RMT_TX_LOOP_NUM_CH*n* | RMT_TX_LIM_CH*n* | |
|---|---|---|---|---|---|
| 31 … 21 | 20 | 19 | 18 … 9 | 8 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0 | 0x80 | Reset |

**RMT_TX_LIM_CH*n*** This field is used to configure the maximum entries that channel *n* can send out. (R/W)

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

**RMT_TX_LOOP_CNT_EN_CH*n*** This bit is the enable bit for loop counting. (R/W)

**RMT_LOOP_COUNT_RESET_CH*n*** This bit is used to reset the loop count when continuous TX mode is enabled. (WT)

**Register 33.17. RMT_TX_SIM_REG (0x006C)**

| (reserved) | RMT_TX_SIM_EN | RMT_TX_SIM_CH1 | RMT_TX_SIM_CH0 | |
|---|---|---|---|---|
| 31 … 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 |

**RMT_TX_SIM_CH0** Set this bit to enable channel 0 to start sending data synchronously with other enabled channels. (R/W)

**RMT_TX_SIM_CH1** Set this bit to enable channel 1 to start sending data synchronously with other enabled channels. (R/W)

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

**Register 33.18. RMT_CH*m*_RX_LIM_REG (*m* = 2, 3) (0x0060, 0x0064)**

| (reserved) | RMT_CH*m*_RX_LIM_REG | |
|---|---|---|
| 31 … 9 | 8 … 0 | |
| 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 | Reset |

**RMT_RX_LIM_CH*m*** This field is used to configure the maximum entries that channel *m* can receive. (R/W)

**Register 33.19. RMT_DATE_REG (0x00CC)**

| (reserved) | RMT_RMT_DATE | |
|---|---|---|
| 31 … 28 | 27 … 0 | |
| 0 0 0 0 | 0x2006231 | Reset |

**RMT_DATE** Version control register. (R/W)

# Part VI

# Analog Signal Processing

This part describes components related to analog-to-digital conversion, on-chip sensors, and features such as temperature sensing, demonstrating the system's capabilities in handling analog signals.

# Chapter 34

# On-Chip Sensor and Analog Signal Processing

## 34.1 Overview

ESP32-C3 provides the following on-chip sensor and analog signal processing peripherals:

- Two 12-bit Successive Approximation ADCs (SAR ADCs): SAR ADC1 and SAR ADC2, for measuring analog signals from six channels.
- One temperature sensor for measuring the internal temperature of the ESP32-C3 chip.

## 34.2 SAR ADCs

### 34.2.1 Overview

ESP32-C3 integrates two 12-bit SAR ADCs, which are able to measure analog signals from up to six pins. The SAR ADCs are managed by two dedicated controllers:

- DIG ADC controller: drives Digital_Reader0 and Digital_Reader1 to sample channel voltages of SAR ADC1 and SAR ADC2, respectively. This DIG ADC controller supports high-performance multi-channel scanning and DMA continuous conversion.
- PWDET controller: monitors RF power. Note this controller is only for RF internal use.

**Note:**
The DIG ADC controller of SAR ADC2 for ESP32-C3 does not work properly and it is suggested to use SAR ADC1. For more information, please refer to ESP32-C3 Series SoC Errata.

### 34.2.2 Features

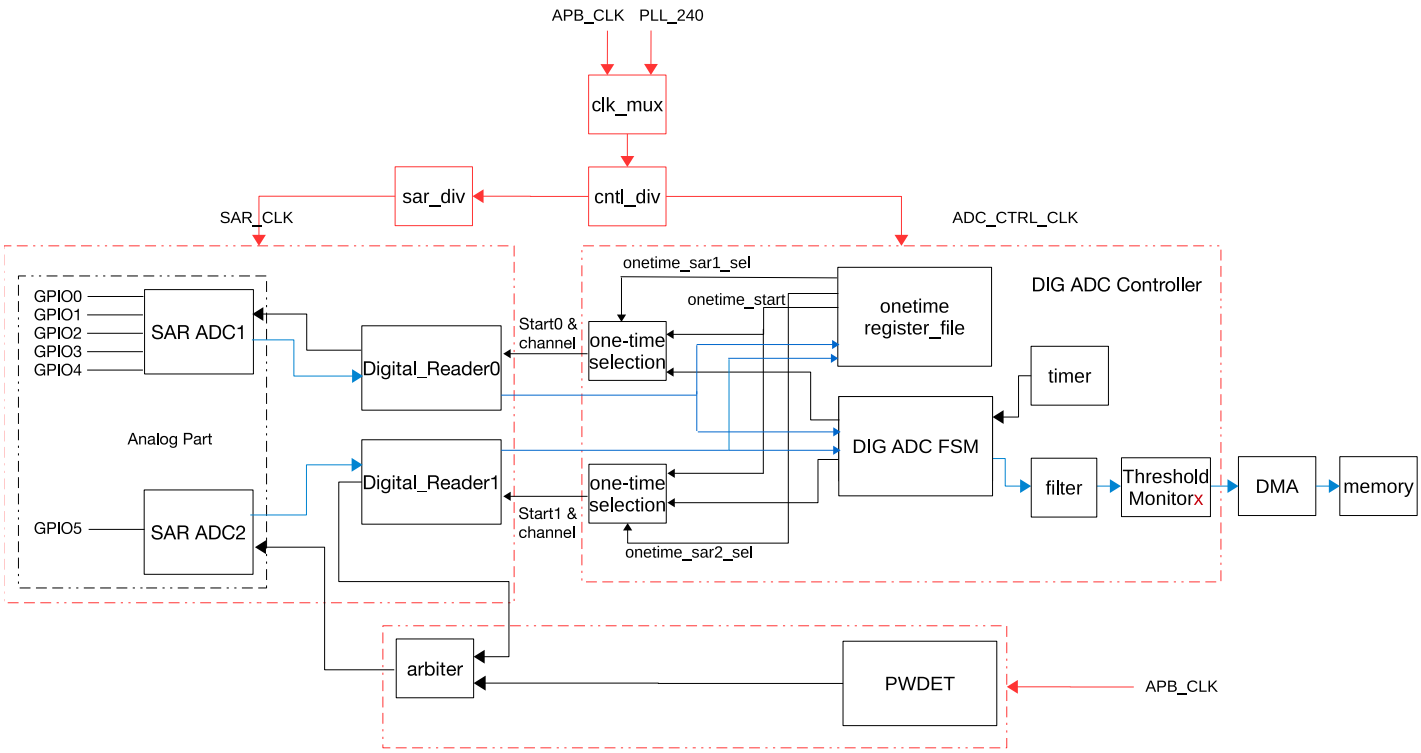
- Each SAR ADC has its own ADC Reader module (Digital_Reader0 or Digital_Reader1), which can be configured and operated separately.
- Support 12-bit sampling resolution
- Support sampling the analog voltages from up to six pins
- DIG ADC controller:
  - Provides separate control modules for one-time sampling and multi-channel scanning.
  - One-time sampling and multi-channel scanning can be run independently on each ADC.
  - Channel scanning sequence in multi-channel scanning mode is user-defined.

- Provides two filters with configurable filter coefficient.
- Supports threshold monitoring. An interrupt will be triggered when the sampled value is greater than the pre-set high threshold or less than the pre-set low threshold.
- Supports DMA

- PWDET controller: monitors RF power (for internal use only)

### 34.2.3 Functional Description

The major components of SAR ADCs and their interconnections are shown in Figure 34.2-1.




—: data flow; —: clock signal; —: ADC control signal

**Figure 34.2-1. SAR ADCs Function Overview**

As shown in Figure 34.2-1, the SAR ADC module consists of the following components:

- SAR ADC1: measures voltages from up to five channels.
- SAR ADC2: measures the voltage from one channel.
- Clock management: selects clock sources and their dividers:
  - Clock sources: can be APB_CLK or PLL_240.
  - Divided Clocks:
    * SAR_CLK: operating clock for SAR ADC1, SAR ADC2, Digital_Reader0, and Digital_Reader1. Note that the divider (sar_div) of SAR_ADC must be no less than 2.
    * ADC_CTRL_CLK: operating clock for DIG ADC FSM.

- Arbiter: this arbiter determines which controller is selected as the ADC2's working controller, DIG ADC controller or PWDET controller.
- Digital_Reader0 (driven by DIG ADC FSM): reads data from SAR ADC1.
- Digital_Reader1 (driven by DIG ADC FSM): reads data from SAR ADC2.
- DIG ADC FSM: generates the signals required throughout the ADC sampling process.
- Threshold monitor*x*: threshold monitor 1 and threshold monitor 2. The monitor*x* will trigger a interrupt when the sampled value is greater than the pre-set high threshold or less than the pre-set low threshold.

The following sections describe the individual components in details.

#### 34.2.3.1 Input Signals

In order to sample an analog signal, an SAR ADC must first select the analog pin to measure via an internal multiplexer. A summary of all the analog signals that may be sent to the SAR ADC module for processing by either ADC1 or ADC2 are presented in Table 34.2-1.

**Table 34.2-1. SAR ADC Input Signals**

| Signal | Channel | ADC Selection |
|---|---|---|
| GPIO0 | 0 | SAR ADC1 |
| GPIO1 | 1 | |
| GPIO2 | 2 | |
| GPIO3 | 3 | |
| GPIO4 | 4 | |
| GPIO5 | 0 | SAR ADC2 |

#### 34.2.3.2 ADC Conversion and Attenuation

When the SAR ADCs convert an analog voltage, the resolution (12-bit) of the conversion spans voltage range from 0 mV to $V_{ref}$. $V_{ref}$ is the SAR ADC's internal reference voltage (1100 mV by design). The output value of the conversion (data) is mapped to analog voltage $V_{data}$ using the following formula:

$$V_{data} = \frac{V_{ref}}{4095} \times data$$

In order to convert voltages larger than $V_{ref}$, input signals can be attenuated before being input into the SAR ADCs. The attenuation can be configured to 0 dB, 2.5 dB, 6 dB, and 12 dB.

#### 34.2.3.3 DIG ADC Controller

The clock of the DIG ADC controller is quite fast, thus the sample rate is high. For more information, see Section ADC Characteristics in ESP32-C3 Series Datasheet.

This controller supports:

- up to 12-bit sampling resolution
- software-triggered one-time sampling
- timer-triggered multi-channel scanning

The configuration of a one-time sampling triggered by the software is as follows:

- Select SAR ADC1 or SAR ADC2 to perform a one-time sampling:
  - if APB_SARADC1_ONETIME_SAMPLE is set, SAR ADC1 is selected.
  - if APB_SARADC2_ONETIME_SAMPLE is set, SAR ADC2 is selected.
- Configure APB_SARADC_ONETIME_CHANNEL to select one channel to sample.
- Configure APB_SARADC_ONETIME_ATTEN to set attenuation.
- Configure APB_SARADC_ONETIME_START to start this one-time sampling.
- On completion of sampling, APB_SARADC_ADC*x*_DONE_INT_RAW interrupt is generated. Software can use this interrupt to initiate reading of the sample values from APB_SARADC_ADC*x*_DATA. *x* can be 1 or 2. 1: SAR ADC1; 2: SAR ADC2.

If the timer-triggered multi-channel scanning is selected, follow the configuration below. Note that in this mode, the scan sequence is performed according to the configuration entered into pattern table.

- Configure APB_SARADC_TIMER_TARGET to set the trigger target for DIG ADC timer. When the timer counting reaches two times of the pre-configured cycle number, a sampling operation is triggered. For the working clock of the timer, see Section 34.2.3.4.
- Configure APB_SARADC_TIMER_EN to enable the timer.
- When the timer times out, it drives DIG ADC FSM to start sampling according to the pattern table;
- Sampled data is automatically stored in memory via DMA. An interrupt is triggered once the scan is completed.

**Note:**
Any SAR ADC can not be configured to perform both one-time sampling and multi-channel scanning at the same time. Therefore, if a pattern table is configured to use any SAR ADC for multi-channel scanning, then this SAR ADC can not be configured to perform one-time sampling.

#### 34.2.3.4 DIG ADC Clock

Two clocks can be used as the working clock of DIG ADC controller, depending on the configuration of APB_SARADC_CLK_SEL:

- 1: Select the clock (ADC_CTRL_CLK) divided from PLL_240.
- 0: Select APB_CLK.

If ADC_CTRL_CLK is selected, users can configure the divider by APB_SARADC_CLKM_DIV_NUM. Note that due to speed limits of SAR ADCs, the operating clock of Digital_Reader0, SAR ADC1, Digital_Reader1, and SAR ADC2 is SAR_CLK, the frequency of which affects the sampling precision. The lower the frequency, the higher the precision. SAR_CLK is divided from ADC_CTRL_CLK. The divider coefficient is configured by APB_SARADC_SAR_CLK_DIV.

The ADC needs 25 SAR_CLK clock cycles per sample, so the maximum sampling rate is limited by the SAR_CLK frequency.

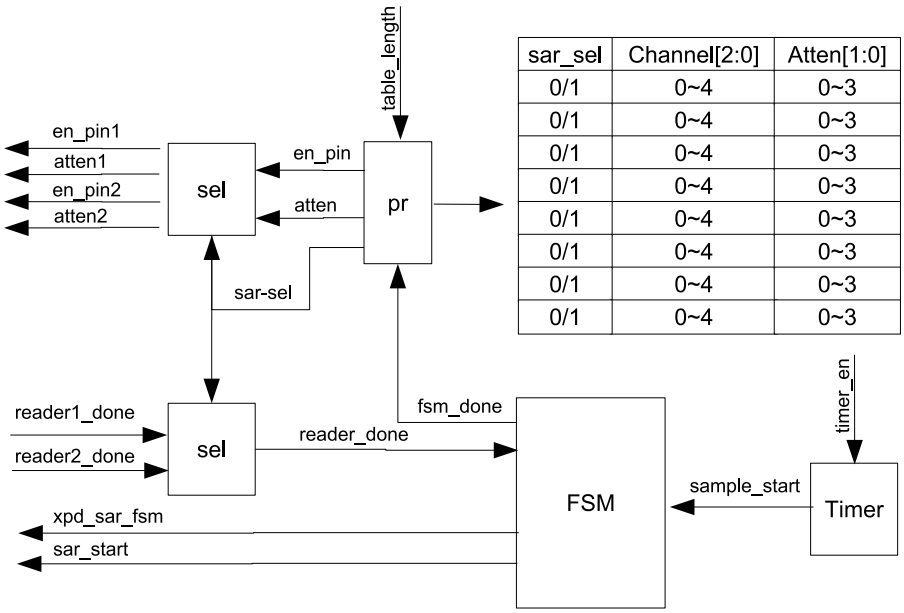
#### 34.2.3.5 DMA Support

DIG ADC controller supports direct memory access via peripheral DMA, which is triggered by DIG ADC timer. Users can switch the DMA data path to DIG ADC by configuring APB_SARADC_APB_ADC_TRANS via software. For specific DMA configuration, please refer to Chapter *2 GDMA Controller (GDMA)*.

#### 34.2.3.6 DIG ADC FSM

**Overview**

Figure 34.2-2 shows the diagram of DIG ADC FSM.




| sar_sel | Channel[2:0] | Atten[1:0] |
|---|---|---|
| 0/1 | 0~4 | 0~3 |
| 0/1 | 0~4 | 0~3 |
| 0/1 | 0~4 | 0~3 |
| 0/1 | 0~4 | 0~3 |
| 0/1 | 0~4 | 0~3 |
| 0/1 | 0~4 | 0~3 |
| 0/1 | 0~4 | 0~3 |
| 0/1 | 0~4 | 0~3 |

Figure 34.2-2. Diagram of DIG ADC FSM

Wherein:

- Timer: a dedicated timer for DIG ADC controller, to generate a sample_start signal.
- pr: the pointer to pattern table entries. FSM sends out corresponding signals based on the configuration of the pattern table entry that the pointer points to.

The execution process is as follows:

- Configure APB_SARADC_TIMER_EN to enable the DIG ADC timer. The timeout event of this timer triggers an sample_start signal. This signal drives the FSM module to start sampling.
- When the FSM module receives the sample_start signal, it starts the following operations:
  - Power up SAR ADC.
  - Select SAR ADC1 or SAR ADC2 as the working ADC, configure the ADC channel and attenuation, based on the pattern table entry that the current pr points to.
  - According to the configuration information, output the corresponding en_pad and atten signals to the analog side.
  - Initiate the sar_start signal and start sampling.

- When the FSM receives the reader_done signal from ADC Reader (Digital_Reader0 or Digital_Reader1), it will
  - stop sampling,
  - transfer the data to the filter, and then threshold monitor transfers the data to memory via DMA,
  - update the pattern table pointer pr and wait for the next sampling. Note that if the pointer pr is smaller than APB_SARADC_SAR_PATT_LEN (table_length), then pr = pr + 1, otherwise, pr is cleared.

### Pattern Table

There is one pattern table in the controller, consisting of the APB_SARADC_SAR_PATT_TAB1_REG and APB_SARADC_SAR_PATT_TAB2_REG registers, see Figure 34.2-3 and Figure 34.2-4:

| (reserved) | cmd3 | cmd2 | cmd1 | cmd0 |
|---|---|---|---|---|
| 31 – 24 | 23 – 18 | 17 – 12 | 11 – 6 | 5 – 0 |
| 0 0 0 0 0 0 0 0 | 0x0000 | 0x0000 | 0x0000 | 0x0000 |

**cmd** *x* represents pattern table entries. *x* here is the index, 0 ~ 3.

**Figure 34.2-3. APB_SARADC_SAR_PATT_TAB1_REG and Pattern Table Entry 0 - Entry 3**

| (reserved) | cmd7 | cmd6 | cmd5 | cmd4 |
|---|---|---|---|---|
| 31 – 24 | 23 – 18 | 17 – 12 | 11 – 6 | 5 – 0 |
| 0 0 0 0 0 0 0 0 | 0x0000 | 0x0000 | 0x0000 | 0x0000 |

**cmd** *x* represents pattern table entries. *x* here is the index, 4 ~ 7.

**Figure 34.2-4. APB_SARADC_SAR_PATT_TAB2_REG and Pattern Table Entry 4 - Entry 7**

Each register consists of four 6-bit pattern table entries. Each entry is composed of three fields that contain working ADC, ADC channel and attenuation information, as shown in Table 34.2-5.

| sar_sel | ch_sel | atten |
|---|---|---|
| 5 | 4 – 2 | 1 – 0 |
| X | XX | X X |

**Figure 34.2-5. Pattern Table Entry**

**atten** Attenuation. 0: 0 dB; 1: 2.5 dB; 2: 6 dB; 3: 12 dB.

**ch_sel** ADC channel, see Table 34.2-1.

**sar_sel** Working ADC. 0: SAR ARC1; 1: SAR ADC2.

### Configuration of multi-channel scanning

In this example, two channels are selected for multi-channel scanning:

- Channel 2 of SAR ADC1, with the attenuation of 12 dB

- Channel 0 of SAR ADC2, with the attenuation of 2.5 dB

The detailed configuration is as follows:

- Configure the first pattern table entry (cmd0):

| sar_sel | ch_sel | atten |
|---|---|---|
| 5 | 4 2 | 1 0 |
| 0 | 2 | 3 |

Figure 34.2-6. cmd0 Configuration

**atten** write the value of 3 to this field, to set the attenuation to 12 dB.

**ch_sel** write the value of 2 to this field, to select channel 2 (see Table 34.2-1).

**sar_sel** write the value of 0 to this bit, to select SAR ADC1 as the working ADC.

- Configure the second pattern table entry (cmd1):

| sar_sel | ch_sel | atten |
|---|---|---|
| 5 | 4 2 | 1 0 |
| 1 | 0 | 1 |

Figure 34.2-7. cmd1 configuration

**atten** write the value of 1 to this field, to set the attenuation to 2.5 dB.

**ch_sel** write the value of 0 to this field, to select channel 0 (see Table 34.2-1).

**sar_sel** write the value of 1 to this bit, to select SAR ADC2 as the working ADC.

- Configure APB_SARADC_SAR_PATT_LEN to 1, i.e., set pattern table length to (this value + 1 = 2). Then pattern table entries cmd0 and cmd1 will be used.
- Enable the timer, then DIG ADC controller starts scanning the two channels in cycles, as configured in the pattern table entries.

### DMA Data Format

The ADC eventually passes 32-bit data to the DMA, see the figure below.

| reserved | sar_sel | ch_sel | reserved | data |
|---|---|---|---|---|
| 31 17 | 16 | 15 13 | 12 | 11 0 |
| xx | x | xxx | x | x x |

Figure 34.2-8. DMA Data Format

**data** SAR ADC read value, 12-bit

**ch_sel** Channel, 3-bit

**sar_sel** SAR ADC selection, 1-bit

#### 34.2.3.7 ADC Filters

The DIG ADC controller provides two filters for automatic filtering of sampled ADC data. Both filters can be configured to any channel of either SAR ADC and then filter the sampled data for the target channel. The filter's formula is shown below:

$$data_{cur} = \frac{(k-1)data_{prev}}{k} + \frac{data_{in}}{k} + 0.5$$

- $data_{cur}$: the filtered data value.
- $data_{in}$: the sampled data value from the ADC.
- $data_{prev}$: the last filtered data value.
- $k$: the filter coefficient.

The filters are configured as follows:

- Configure APB_SARADC_FILTER_CHANNEL*x* to select the ADC channel for filter *x*;
- Configure APB_SARADC_FILTER_FACTOR*x* to set the coefficient for filter *x*;

Note that *x* is used here as the placeholder of filter index. 0: filter 0; 1: filter 1.

#### 34.2.3.8 Threshold Monitoring

DIG ADC controller contains two threshold monitors that can be configured to monitor on any channel of SAR ADC1 and SAR ADC2. A high threshold interrupt is triggered when the ADC sample value is larger than the pre-configured high threshold, and a low threshold interrupt is triggered if the sample value is lower than the pre-configured low threshold.

The configuration of threshold monitoring is as follows:

- Set APB_SARADC_THRES*x*_EN to enable threshold monitor *x*.
- Configure APB_SARADC_THRES*x*_LOW to set a low threshold;
- Configure APB_SARADC_THRES*x*_HIGH to set a high threshold;
- Configure APB_SARADC_THRES*x*_CHANNEL to select the SAR ADC and the channel to monitor.

Note that *x* is used here as the placeholder of monitor index. 0: monitor 0; 1: monitor 1.

#### 34.2.3.9 SAR ADC2 Arbiter

SAR ADC2 can be controlled by two controllers, namely, DIG ADC controller and PWDET controller. To avoid any possible conflicts and to improve the efficiency of SAR ADC2, ESP32-C3 provides an arbiter for SAR ADC2. The arbiter supports fair arbitration and fixed priority arbitration.

- Fair arbitration mode (cyclic priority arbitration) can be enabled by clearing APB_SARADC_ADC_ARB_FIX_PRIORITY.
- In fixed priority arbitration, users can set APB_SARADC_ADC_ARB_APB_PRIORITY (for DIG ADC controller) and APB_SARADC_ADC_ARB_WIFI_PRIORITY (for PWDET controller), to configure the priorities for these controllers. A larger value indicates a higher priority.

The arbiter ensures that a higher priority controller can always start a conversion (sample) when required, regardless of whether a lower priority controller already has a conversion in progress. If a higher priority controller starts a conversion whilst the ADC already has a conversion in progress from a lower priority controller, the conversion in progress will be interrupted (stopped). The higher priority controller will then start its conversion. A lower priority controller will not be able to start a conversion whilst the ADC has a conversion in progress from a higher priority controller.

Therefore, certain data flags are embedded into the output data value to indicate whether the conversion is valid or not.

- The data flag for DIG ADC controller is the {sar_sel, ch_sel} bits in DMA data, see Figure 34.2-8.
  - 4'b1111: Conversion is interrupted.
  - 4'b1110: Conversion is not started.
  - Corresponding channel No.: The data is valid.
- The data flag for PWDET controller is the two higher bits of the sampling result.
  - 2'b10: Conversion is interrupted.
  - 2'b01: Conversion is not started.
  - 2'b00: The data is valid.

Users can configure APB_SARADC_ADC_ARB_GRANT_FORCE to mask the arbiter, and set APB_SARADC_ADC_ARB_WIFI_FORCE or APB_SARADC_ADC_ARB_APB_FORCE to authorize corresponding controllers.

## 34.3 Temperature Sensor

### 34.3.1 Overview

ESP32-C3 provides a temperature sensor to monitor temperature changes inside the chip in real time.

### 34.3.2 Features

The temperature sensor has the following features:

- Supports software triggering and, once triggered, the data can be read continuously
- Configurable temperature offset based on the environment, to improve the accuracy
- Adjustable measurement range

### 34.3.3 Functional Description

The temperature sensor can be started by software as follows:

- Set APB_SARADC_TSENS_PU to start XPD_SAR, and then to enable temperature sensor;
- Set SYSTEM_TSENS_CLK_EN to enable temperature sensor clock;
- Wait for APB_SARADC_TSENS_XPD_WAIT clock cycles till the reset of temperature sensor is released, the sensor starts measuring the temperature;

- Wait for a while and then read the data from APB_SARADC_TSENS_OUT. The output value gradually approaches the actual temperature linearly as the measurement time increases.

The actual temperature (°C) can be obtained by converting the output of temperature sensor via the following formula:

$$T(°C) = 0.4386 * VALUE - 27.88 * offset - 20.52$$

VALUE in the formula is the output of the temperature sensor, and the offset is determined by the temperature offset. The temperature offset varies in different actual environment (the temperature range). For details, refer to Table 34.3-1.

**Table 34.3-1. Temperature Offset**

| Measurement Range (°C) | Temperature Offset (°C) |
|---|---|
| 50 ~ 125 | -2 |
| 20 ~ 100 | -1 |
| -10 ~ 80 | 0 |
| -30 ~ 50 | 1 |
| -40 ~ 20 | 2 |

## 34.4 Interrupts

- APB_SARADC_ADC1_DONE_INT: Triggered when SAR ADC1 completes one data conversion.
- APB_SARADC_ADC2_DONE_INT: Triggered when SAR ADC2 completes one data conversion.
- APB_SARADC_THRES*x*_HIGH_INT: Triggered when the sampling value is higher than the high threshold of monitor *x*.
- APB_SARADC_THRES*x*_LOW_INT: Triggered when the sampling value is lower than the low threshold of monitor *x*.

## 34.5 Register Summary

The addresses in this section are relative to the ADC controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

The abbreviations given in Column **Access** are explained in Section *Access Types for Registers*.

| Name | Description | Address | Access |
|---|---|---|---|
| **Configuration Registers** | | | |
| APB_SARADC_CTRL_REG | SAR ADC control register 1 | 0x0000 | R/W |
| APB_SARADC_CTRL2_REG | SAR ADC control register 2 | 0x0004 | R/W |
| APB_SARADC_FILTER_CTRL1_REG | Filtering control register 1 | 0x0008 | R/W |
| APB_SARADC_SAR_PATT_TAB1_REG | Pattern table register 1 | 0x0018 | R/W |
| APB_SARADC_SAR_PATT_TAB2_REG | Pattern table register 2 | 0x001C | R/W |
| APB_SARADC_ONETIME_SAMPLE_REG | Configuration register for one-time sampling | 0x0020 | R/W |

| Name | Description | Address | Access |
|---|---|---|---|
| APB_SARADC_APB_ADC_ARB_CTRL_REG | SAR ADC2 arbiter configuration register | 0x0024 | R/W |
| APB_SARADC_FILTER_CTRL0_REG | Filtering control register 0 | 0x0028 | R/W |
| APB_SARADC_1_DATA_STATUS_REG | SAR ADC1 sampling data register | 0x002C | RO |
| APB_SARADC_2_DATA_STATUS_REG | SAR ADC2 sampling data register | 0x0030 | RO |
| APB_SARADC_THRES0_CTRL_REG | Sampling threshold control register 0 | 0x0034 | R/W |
| APB_SARADC_THRES1_CTRL_REG | Sampling threshold control register 1 | 0x0038 | R/W |
| APB_SARADC_THRES_CTRL_REG | Sampling threshold control register | 0x003C | R/W |
| APB_SARADC_INT_ENA_REG | Enable register of SAR ADC interrupts | 0x0040 | R/W |
| APB_SARADC_INT_RAW_REG | Raw register of SAR ADC interrupts | 0x0044 | RO |
| APB_SARADC_INT_ST_REG | State register of SAR ADC interrupts | 0x0048 | RO |
| APB_SARADC_INT_CLR_REG | Clear register of SAR ADC interrupts | 0x004C | WO |
| APB_SARADC_DMA_CONF_REG | DMA configuration register for SAR ADC | 0x0050 | R/W |
| APB_SARADC_APB_ADC_CLKM_CONF_REG | SAR ADC clock control register | 0x0054 | R/W |
| APB_SARADC_APB_TSENS_CTRL_REG | Temperature sensor control register 1 | 0x0058 | varies |
| APB_SARADC_APB_TSENS_CTRL2_REG | Temperature sensor control register 2 | 0x005C | R/W |
| APB_SARADC_CALI_REG | SAR ADC calibration register | 0x0060 | R/W |
| APB_SARADC_APB_CTRL_DATE_REG | Version control register | 0x03FC | R/W |

## 34.6 Register

The addresses in this section are relative to the ADC controller base address provided in Table 3.3-3 in Chapter 3 *System and Memory*.

**Register 34.1. APB_SARADC_CTRL_REG (0x0000)**

| APB_SARADC_WAIT_ARB_CYCLE | (reserved) | APB_SARADC_XPD_SAR_FORCE | (reserved) | APB_SARADC_SAR_PATT_P_CLEAR | (reserved) | APB_SARADC_SAR_PATT_LEN | APB_SARADC_SAR_CLK_DIV | APB_SARADC_SAR_CLK_GATED | (reserved) | APB_SARADC_START | APB_SARADC_START_FORCE | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 30 | 29 | 28 27 | 26 24 | 23 | 22 18 | 17 15 | 14 7 | 6 | 5 2 | 1 | 0 | |
| 1 | 0 | 0 | 0 0 0 | 0 | 0 0 0 0 0 | 7 | 4 | 1 | 0 0 0 0 | 0 | 0 | Reset |

**APB_SARADC_START_FORCE** 0: select FSM to start SAR ADC. 1: select software to start SAR ADC. (R/W)

**APB_SARADC_START** Write 1 here to start the SAR ADC by software. Valid only when APB_SARADC_START_FORCE = 1. (R/W)

**APB_SARADC_SAR_CLK_GATED** SAR ADC clock gate enable bit. (R/W)

**APB_SARADC_SAR_CLK_DIV** SAR ADC clock divider. This value should be no less than 2. (R/W)

**APB_SARADC_SAR_PATT_LEN** Configure how many pattern table entries will be used. If this field is set to 1, then pattern table entries (cmd0) and (cmd1) will be used. (R/W)

**APB_SARADC_SAR_PATT_P_CLEAR** Clear the pointer of pattern table entry for DIG ADC controller. (R/W)

**APB_SARADC_XPD_SAR_FORCE** Force select XPD SAR. (R/W)

**APB_SARADC_WAIT_ARB_CYCLE** The clock cycles of waiting arbitration signal stable after SAR_DONE. (R/W)

### Register 34.2. APB_SARADC_CTRL2_REG (0x0004)

| 31–25 | 24 | 23–12 | 11 | 10 | 9 | 8–1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| (reserved) | APB_SARADC_TIMER_EN | APB_SARADC_TIMER_TARGET | (reserved) | APB_SARADC_SAR2_INV | APB_SARADC_SAR1_INV | APB_SARADC_MAX_MEAS_NUM | APB_SARADC_MEAS_NUM_LIMIT | |
| 0 0 0 0 0 0 0 | 0 | 10 | 0 | 0 | 0 | 255 | 0 | Reset |

**APB_SARADC_MEAS_NUM_LIMIT** Enable the limitation of SAR ADCs maximum conversion times. (R/W)

**APB_SARADC_MAX_MEAS_NUM** The SAR ADCs maximum conversion times. (R/W)

**APB_SARADC_SAR1_INV** Write 1 here to invert the data of SAR ADC1. (R/W)

**APB_SARADC_SAR2_INV** Write 1 here to invert the data of SAR ADC2. (R/W)

**APB_SARADC_TIMER_TARGET** Set SAR ADC timer target. (R/W)

**APB_SARADC_TIMER_EN** Enable SAR ADC timer trigger. (R/W)

### Register 34.3. APB_SARADC_FILTER_CTRL1_REG (0x0008)

| 31–29 | 28–26 | 25–0 | |
|---|---|---|---|
| APB_SARADC_FILTER_FACTOR0 | APB_SARADC_FILTER_FACTOR1 | (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 | Reset |

**APB_SARADC_FILTER_FACTOR1** The filter coefficient for SAR ADC filter 1. (R/W)

**APB_SARADC_FILTER_FACTOR0** The filter coefficient for SAR ADC filter 0. (R/W)

### Register 34.4. APB_SARADC_SAR_PATT_TAB1_REG (0x0018)

| (reserved) | APB_SARADC_SAR_PATT_TAB1 | |
|---|---|---|
| 31 … 24 | 23 … 0 | |
| 0 0 0 0 0 0 0 0 | 0x0000 | Reset |

**APB_SARADC_SAR_PATT_TAB1** Pattern table entries 0 ~ 3 (each entry is six bits). (R/W)

### Register 34.5. APB_SARADC_SAR_PATT_TAB2_REG (0x001C)

| (reserved) | APB_SARADC_SAR_PATT_TAB2 | |
|---|---|---|
| 31 … 24 | 23 … 0 | |
| 0 0 0 0 0 0 0 0 | 0x0000 | Reset |

**APB_SARADC_SAR_PATT_TAB2** Pattern table entries 4 ~ 7 (each entry is six bits). (R/W)

### Register 34.6. APB_SARADC_ONETIME_SAMPLE_REG (0x0020)

| APB_SARADC1_ONETIME_SAMPLE | APB_SARADC2_ONETIME_SAMPLE | APB_SARADC_ONETIME_START | APB_SARADC_ONETIME_CHANNEL | APB_SARADC_ONETIME_ATTEN | (reserved) | |
|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 … 25 | 24 … 23 | 22 … 0 | |
| 0 | 0 | 0 | 13 | 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 |

**APB_SARADC_ONETIME_ATTEN** Configure the attenuation for a one-time sampling. (R/W)

**APB_SARADC_ONETIME_CHANNEL** Configure the channel for a one-time sampling. (R/W)

**APB_SARADC_ONETIME_START** Start SAR ADC one-time sampling. (R/W)

**APB_SARADC2_ONETIME_SAMPLE** Enable SAR ADC2 one-time sampling. (R/W)

**APB_SARADC1_ONETIME_SAMPLE** Enable SAR ADC1 one-time sampling. (R/W)

### Register 34.7. APB_SARADC_APB_ADC_ARB_CTRL_REG (0x0024)

| (reserved) | APB_SARADC_ADC_ARB_FIX_PRIORITY | APB_SARADC_ADC_ARB_WIFI_PRIORITY | (reserved) | APB_SARADC_ADC_ARB_APB_PRIORITY | APB_SARADC_ADC_ARB_GRANT_FORCE | APB_SARADC_ADC_ARB_WIFI_FORCE | (reserved) | APB_SARADC_ADC_ARB_APB_FORCE | (reserved) | |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 … 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 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 0 | Reset |

**APB_SARADC_ADC_ARB_APB_FORCE** SAR ADC2 arbiter forces to enable DIG ADC controller. (R/W)

**APB_SARADC_ADC_ARB_WIFI_FORCE** SAR ADC2 arbiter forces to enable PWDET controller. (R/W)

**APB_SARADC_ADC_ARB_GRANT_FORCE** ADC2 arbiter force grant. (R/W)

**APB_SARADC_ADC_ARB_APB_PRIORITY** Set DIG ADC controller priority. (R/W)

**APB_SARADC_ADC_ARB_WIFI_PRIORITY** Set PWDET controller priority. (R/W)

**APB_SARADC_ADC_ARB_FIX_PRIORITY** ADC2 arbiter uses fixed priority. (R/W)

### Register 34.8. APB_SARADC_FILTER_CTRL0_REG (0x0028)

| APB_SARADC_FILTER_RESET | (reserved) | APB_SARADC_FILTER_CHANNEL0 | APB_SARADC_FILTER_CHANNEL1 | (reserved) | |
|---|---|---|---|---|---|
| 31 | 30 … 26 | 25 … 22 | 21 … 18 | 17 … 0 | |
| 0 | 0 0 0 0 0 | 13 | 13 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | Reset |

**APB_SARADC_FILTER_CHANNEL1** The filter channel for SAR ADC filter 1. (R/W)

**APB_SARADC_FILTER_CHANNEL0** The filter channel for SAR ADC filter 0. (R/W)

**APB_SARADC_FILTER_RESET** Reset SAR ADC1 filter. (R/W)

### Register 34.9. APB_SARADC_1_DATA_STATUS_REG (0x002C)

| (reserved) | APB_SARADC_APB_SARADC1_DATA | |
|---|---|---|
| 31 … 17 | 16 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**APB_SARADC_ADC1_DATA** SAR ADC1 conversion data. (RO)

### Register 34.10. APB_SARADC_2_DATA_STATUS_REG (0x0030)

| (reserved) | APB_SARADC_ADC2_DATA | |
|---|---|---|
| 31 … 17 | 16 … 0 | |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | Reset |

**APB_SARADC_ADC2_DATA** SAR ADC2 conversion data. (RO)

### Register 34.11. APB_SARADC_THRES0_CTRL_REG (0x0034)

| (reserved) | APB_SARADC_THRES0_LOW | APB_SARADC_THRES0_HIGH | (reserved) | APB_SARADC_THRES0_CHANNEL | |
|---|---|---|---|---|---|
| 31 | 30 … 18 | 17 … 5 | 4 | 3 … 0 | |
| 0 | 0 | 0x1fff | 0 | 13 | Reset |

**APB_SARADC_THRES0_CHANNEL** The channel for SAR ADC monitor 0. (R/W)

**APB_SARADC_THRES0_HIGH** The high threshold for SAR ADC monitor 0. (R/W)

**APB_SARADC_THRES0_LOW** The low threshold for SAR ADC monitor 0. (R/W)

**Register 34.12. APB_SARADC_THRES1_CTRL_REG (0x0038)**

| (reserved) | APB_SARADC_THRES1_LOW | APB_SARADC_THRES1_HIGH | (reserved) | APB_SARADC_THRES1_CHANNEL | |
|---|---|---|---|---|---|
| 31 | 30 18 | 17 5 | 4 | 3 0 | |
| 0 | 0 | 0x1fff | 0 | 13 | Reset |

**APB_SARADC_THRES1_CHANNEL** The channel for SAR ADC monitor 1. (R/W)

**APB_SARADC_THRES1_HIGH** The high threshold for SAR ADC monitor 1. (R/W)

**APB_SARADC_THRES1_LOW** The low threshold for SAR ADC monitor 1. (R/W)

**Register 34.13. APB_SARADC_THRES_CTRL_REG (0x003C)**

| APB_SARADC_THRES0_EN | APB_SARADC_THRES1_EN | (reserved) | APB_SARADC_THRES_ALL_EN | (reserved) | |
|---|---|---|---|---|---|
| 31 | 30 | 29 28 | 27 | 26 0 | |
| 0 | 0 | 0 0 | 0 | 0 0 0 0 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 |

**APB_SARADC_THRES_ALL_EN** Enable the threshold monitoring for all configured channels. (R/W)

**APB_SARADC_THRES1_EN** Enable threshold monitor 1. (R/W)

**APB_SARADC_THRES0_EN** Enable threshold monitor 0. (R/W)

**Register 34.14. APB_SARADC_INT_ENA_REG (0x0040)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APB_SARADC_ADC1_DONE_INT_ENA | APB_SARADC_ADC2_DONE_INT_ENA | APB_SARADC_THRES0_HIGH_INT_ENA | APB_SARADC_THRES1_HIGH_INT_ENA | APB_SARADC_THRES0_LOW_INT_ENA | APB_SARADC_THRES1_LOW_INT_ENA | (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 |

**APB_SARADC_THRES1_LOW_INT_ENA** Enable bit of APB_SARADC_THRES1_LOW_INT interrupt. (R/W)

**APB_SARADC_THRES0_LOW_INT_ENA** Enable bit of APB_SARADC_THRES0_LOW_INT interrupt. (R/W)

**APB_SARADC_THRES1_HIGH_INT_ENA** Enable bit of APB_SARADC_THRES1_HIGH_INT interrupt. (R/W)

**APB_SARADC_THRES0_HIGH_INT_ENA** Enable bit of APB_SARADC_THRES0_HIGH_INT interrupt. (R/W)

**APB_SARADC_ADC2_DONE_INT_ENA** Enable bit of APB_SARADC_ADC2_DONE_INT interrupt. (R/W)

**APB_SARADC_ADC1_DONE_INT_ENA** Enable bit of APB_SARADC_ADC1_DONE_INT interrupt. (R/W)

Register 34.15. APB_SARADC_INT_RAW_REG (0x0044)

| APB_SARADC_ADC1_DONE_INT_RAW | APB_SARADC_ADC2_DONE_INT_RAW | APB_SARADC_THRES0_HIGH_INT_RAW | APB_SARADC_THRES1_HIGH_INT_RAW | APB_SARADC_THRES0_LOW_INT_RAW | APB_SARADC_THRES1_LOW_INT_RAW | (reserved) | |
|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 … 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 0 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 |

**APB_SARADC_THRES1_LOW_INT_RAW** Raw bit of APB_SARADC_THRES1_LOW_INT interrupt. (RO)

**APB_SARADC_THRES0_LOW_INT_RAW** Raw bit of APB_SARADC_THRES0_LOW_INT interrupt. (RO)

**APB_SARADC_THRES1_HIGH_INT_RAW** Raw bit of APB_SARADC_THRES1_HIGH_INT interrupt. (RO)

**APB_SARADC_THRES0_HIGH_INT_RAW** Raw bit of APB_SARADC_THRES0_HIGH_INT interrupt. (RO)

**APB_SARADC_ADC2_DONE_INT_RAW** Raw bit of APB_SARADC_ADC2_DONE_INT interrupt. (RO)

**APB_SARADC_ADC1_DONE_INT_RAW** Raw bit of APB_SARADC_ADC1_DONE_INT interrupt. (RO)

Register 34.16. APB_SARADC_INT_ST_REG (0x0048)

| APB_SARADC_ADC1_DONE_INT_ST | APB_SARADC_ADC2_DONE_INT_ST | APB_SARADC_THRES0_HIGH_INT_ST | APB_SARADC_THRES1_HIGH_INT_ST | APB_SARADC_THRES0_LOW_INT_ST | APB_SARADC_THRES1_LOW_INT_ST | (reserved) | |
|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 … 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 0 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 |

**APB_SARADC_THRES1_LOW_INT_ST** Status of APB_SARADC_THRES1_LOW_INT interrupt. (RO)

**APB_SARADC_THRES0_LOW_INT_ST** Status of APB_SARADC_THRES0_LOW_INT interrupt. (RO)

**APB_SARADC_THRES1_HIGH_INT_ST** Status of APB_SARADC_THRES1_HIGH_INT interrupt. (RO)

**APB_SARADC_THRES0_HIGH_INT_ST** Status of APB_SARADC_THRES0_HIGH_INT interrupt. (RO)

**APB_SARADC_ADC2_DONE_INT_ST** Status of APB_SARADC_ADC2_DONE_INT interrupt. (RO)

**APB_SARADC_ADC1_DONE_INT_ST** Status of APB_SARADC_ADC1_DONE_INT interrupt. (RO)

Register 34.17. APB_SARADC_INT_CLR_REG (0x004C)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 … 0 | |
|---|---|---|---|---|---|---|---|
| APB_SARADC_ADC1_DONE_INT_CLR | APB_SARADC_ADC2_DONE_INT_CLR | APB_SARADC_THRES0_HIGH_INT_CLR | APB_SARADC_THRES1_HIGH_INT_CLR | APB_SARADC_THRES0_LOW_INT_CLR | APB_SARADC_THRES1_LOW_INT_CLR | (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 | Reset |

**APB_SARADC_THRES1_LOW_INT_CLR** Clear bit of APB_SARADC_THRES1_LOW_INT interrupt. (WO)

**APB_SARADC_THRES0_LOW_INT_CLR** Clear bit of APB_SARADC_THRES0_LOW_INT interrupt. (WO)

**APB_SARADC_THRES1_HIGH_INT_CLR** Clear bit of APB_SARADC_THRES1_HIGH_INT interrupt. (WO)

**APB_SARADC_THRES0_HIGH_INT_CLR** Clear bit of APB_SARADC_THRES0_HIGH_INT interrupt. (WO)

**APB_SARADC_ADC2_DONE_INT_CLR** Clear bit of APB_SARADC_ADC2_DONE_INT interrupt. (WO)

**APB_SARADC_ADC1_DONE_INT_CLR** Clear bit of APB_SARADC_ADC1_DONE_INT interrupt. (WO)

Register 34.18. APB_SARADC_DMA_CONF_REG (0x0050)

| 31 | 30 | 29 … 16 | 15 … 0 | |
|---|---|---|---|---|
| APB_SARADC_APB_ADC_TRANS | APB_SARADC_APB_ADC_RESET_FSM | (reserved) | APB_SARADC_APB_ADC_EOF_NUM | |
| 0 | 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 255 | Reset |

**APB_SARADC_APB_ADC_EOF_NUM** Generate dma_in_suc_eof when sample cnt = eof_num. (R/W)

**APB_SARADC_APB_ADC_RESET_FSM** Reset DIG ADC controller status. (R/W)

**APB_SARADC_APB_ADC_TRANS** When this bit is set, DIG ADC controller uses DMA. (R/W)

**Register 34.19. APB_SARADC_APB_ADC_CLKM_CONF_REG (0x0054)**

| (reserved) | APB_SARADC_CLK_SEL | APB_SARADC_CLK_EN | APB_SARADC_CLKM_DIV_A | APB_SARADC_CLKM_DIV_B | APB_SARADC_CLKM_DIV_NUM | |
|---|---|---|---|---|---|---|
| 31 … 23 | 22 … 21 | 20 | 19 … 14 | 13 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 | 0 | 0 | 0x0 | 0x0 | 4 | Reset |

**APB_SARADC_CLKM_DIV_NUM** The integer part of ADC clock divider. Divider value = APB_SARADC_CLKM_DIV_NUM + APB_SARADC_CLKM_DIV_B/APB_SARADC_CLKM_DIV_A. (R/W)

**APB_SARADC_CLKM_DIV_B** The numerator value of fractional clock divider. (R/W)

**APB_SARADC_CLKM_DIV_A** The denominator value of fractional clock divider. (R/W)

**APB_SARADC_CLK_EN** Enable the SAR ADC register clock. (R/W)

**APB_SARADC_CLK_SEL** 0: Use APB_CLK as clock source, 1: use divided-down PLL_240 as clock source. (R/W)

**Register 34.20. APB_SARADC_APB_TSENS_CTRL_REG (0x0058)**

| (reserved) | APB_SARADC_TSENS_PU | APB_SARADC_TSENS_CLK_DIV | APB_SARADC_TSENS_IN_INV | (reserved) | APB_SARADC_TSENS_OUT | |
|---|---|---|---|---|---|---|
| 31 … 23 | 22 | 21 … 14 | 13 | 12 … 8 | 7 … 0 | |
| 0 0 0 0 0 0 0 0 0 | 0 | 6 | 0 | 0 0 0 0 0 | 0x0 | Reset |

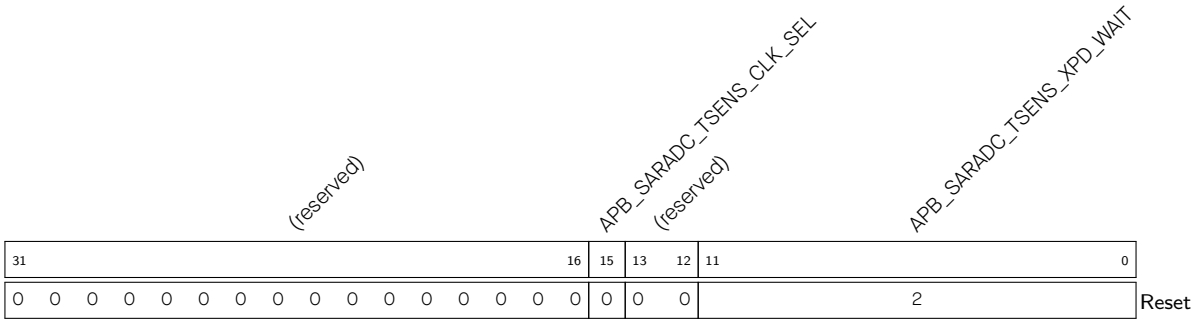
**APB_SARADC_TSENS_OUT** Temperature sensor data out. (RO)

**APB_SARADC_TSENS_IN_INV** Invert temperature sensor input value. (R/W)

**APB_SARADC_TSENS_CLK_DIV** Temperature sensor clock divider. (R/W)

**APB_SARADC_TSENS_PU** Temperature sensor power up. (R/W)

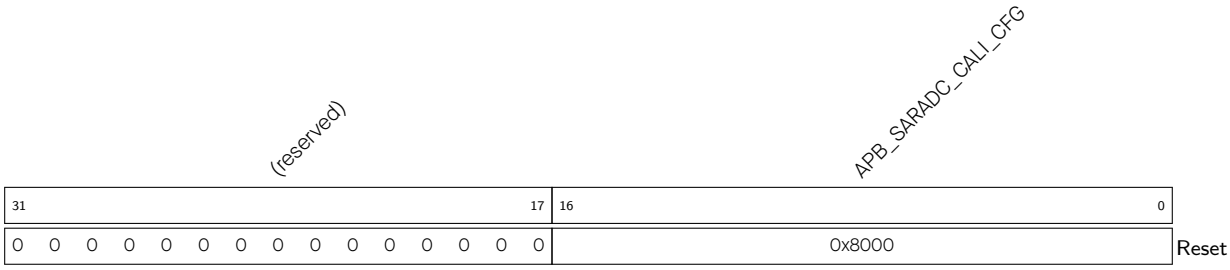
### Register 34.21. APB_SARADC_APB_TSENS_CTRL2_REG (0x005C)




**APB_SARADC_TSENS_XPD_WAIT** The wait time before temperature sensor is powered up. (R/W)

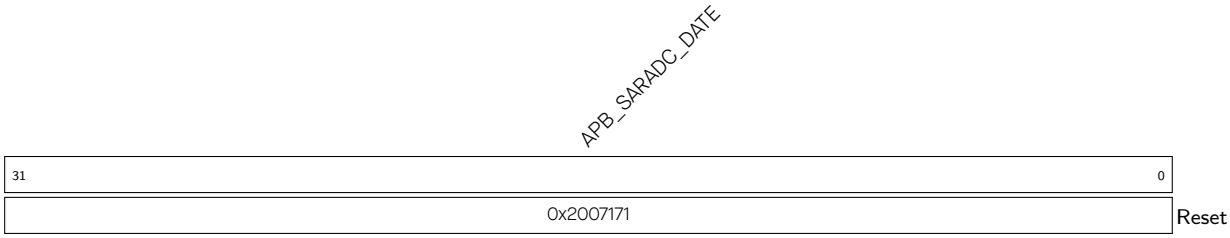
**APB_SARADC_TSENS_CLK_SEL** Choose working clock for temperature sensor. 0: RC_FAST_CLK. 1: XTAL_CLK. (R/W)

### Register 34.22. APB_SARADC_CALI_REG (0x0060)




**APB_SARADC_CALI_CFG** Configure the SAR ADC calibration factor. (R/W)

### Register 34.23. APB_SARADC_APB_CTRL_DATE_REG (0x03FC)




**APB_SARADC_DATE** Version register. (R/W)

# Part VII

# Appendix

This part contains the following information starting from the next page:

- *Related Documentation and Resources*
- *Glossary*
- *Programming Reserved Register Field*
- *Interrupt Configuration Registers*
- *Revision History*

# Related Documentation and Resources

## Related Documentation

- ESP32-C3 Series Datasheet – Specifications of the ESP32-C3 hardware.
- ESP32-C3 Hardware Design Guidelines – Guidelines on how to integrate the ESP32-C3 into your hardware product.
- ESP32-C3 Series SoC Errata – Descriptions of known errors in ESP32-C3 series of SoCs.
- *Certificates*
  https://espressif.com/en/support/documents/certificates
- *ESP32-C3 Product/Process Change Notifications (PCN)*
  https://espressif.com/en/support/documents/pcns?keys=ESP32-C3
- *ESP32-C3 Advisories* – Information on security, bugs, compatibility, component reliability.
  https://espressif.com/en/support/documents/advisories?keys=ESP32-C3
- *Documentation Updates and Update Notification Subscription*
  https://espressif.com/en/support/download/documents

## Developer Zone

- ESP-IDF Programming Guide for ESP32-C3 – Extensive documentation for the ESP-IDF development framework.
- *ESP-IDF* and other development frameworks on GitHub.
  https://github.com/espressif
- *ESP32 BBS Forum* – Engineer-to-Engineer (E2E) Community for Espressif products where you can post questions, share knowledge, explore ideas, and help solve problems with fellow engineers.
  https://esp32.com/
- *The ESP Journal* – Best Practices, Articles, and Notes from Espressif folks.
  https://blog.espressif.com/
- See the tabs *SDKs and Demos*, *Apps*, *Tools*, *AT Firmware*.
  https://espressif.com/en/support/download/sdks-demos

## Products

- *ESP32-C3 Series SoCs* – Browse through all ESP32-C3 SoCs.
  https://espressif.com/en/products/socs?id=ESP32-C3
- *ESP32-C3 Series Modules* – Browse through all ESP32-C3-based modules.
  https://espressif.com/en/products/modules?id=ESP32-C3
- *ESP32-C3 Series DevKits* – Browse through all ESP32-C3-based devkits.
  https://espressif.com/en/products/devkits?id=ESP32-C3
- *ESP Product Selector* – Find an Espressif hardware product suitable for your needs by comparing or applying filters.
  https://products.espressif.com/#/product-selector?language=en

## Contact Us

- See the tabs *Sales Questions*, *Technical Enquiries*, *Circuit Schematic & PCB Design Review*, *Get Samples* (Online stores), *Become Our Supplier*, *Comments & Suggestions*.
  https://espressif.com/en/contact-us/sales-questions

# Glossary

## Abbreviations for Peripherals

| | |
|---|---|
| AES | AES (Advanced Encryption Standard) Accelerator |
| DS | Digital Signature |
| DMA | DMA (Direct Memory Access) Controller |
| eFuse | eFuse Controller |
| HMAC | HMAC (Hash-based Message Authentication Code) Accelerator |
| I2C | I2C (Inter-Integrated Circuit) Controller |
| I2S | I2S (Inter-IC Sound) Controller |
| LEDC | LED Control PWM (Pulse Width Modulation) |
| RMT | Remote Control Peripheral |
| RNG | Random Number Generator |
| RSA | RSA (Rivest Shamir Adleman) Accelerator |
| SHA | SHA (Secure Hash Algorithm) Accelerator |
| SPI | SPI (Serial Peripheral Interface) Controller |
| SYSTIMER | System Timer |
| TIMG | Timer Group |
| TWAI | Two-wire Automotive Interface |
| UART | UART (Universal Asynchronous Receiver-Transmitter) Controller |
| WDT | Watchdog Timers |

## Abbreviations Related to Registers

| | |
|---|---|
| REG | **Register**. |
| SYSREG | **System registers** are a group of registers that control system reset, memory, clocks, software interrupts, power management, clock gating, etc. |
| ISO | **Isolation**. If a peripheral or other chip component is powered down, the pins, if any, to which its output signals are routed will go into a floating state. ISO registers isolate such pins and keep them at a certain determined value, so that the other non-powered-down peripherals/devices attached to these pins are not affected. |
| NMI | **Non-maskable interrupt** is a hardware interrupt that cannot be disabled or ignored by the CPU instructions. Such interrupts exist to signal the occurrence of a critical error. |
| W1TS | Abbreviation added to names of registers/fields to indicate that such register/field should be used to set a field in a corresponding register with a similar name. For example, the register `GPIO_ENABLE_W1TS_REG` should be used to set the corresponding fields in the register `GPIO_ENABLE_REG`. |
| W1TC | Same as *W1TS*, but used to clear a field in a corresponding register. |

## Access Types for Registers

Sections *Register Summary* and *Register Description* in TRM chapters specify access types for registers and their fields.

Most frequently used access types and their combinations are as follows:

- RO
- WO
- WT
- R/W
- R/W1
- WL
- R/W/SC
- R/W/SS
- R/W/SS/SC
- R/WC/SS
- R/WC/SC
- R/WC/SS/SC
- R/WS/SC
- R/WS/SS
- R/WS/SS/SC
- R/SS/WTC
- R/SC/WTC
- R/SS/SC/WTC
- RF/WF
- R/SS/RC
- varies

Descriptions of all access types are provided below.

R **Read.** User application can read from this register/field; usually combined with other access types.

RO **Read only.** User application can only read from this register/field.

HRO **Hardware Read Only.** Only hardware can read from this register/field; used for storing default settings for variable parameters.

W **Write.** User application can write to this register/field; usually combined with other access types.

WO **Write only.** User application can only write to this register/field.

W1 **Write Once.** User application can write to this register/field only once; only allowed to write 1; writing 0 is invalid.

SS **Self set.** On a specified event, hardware automatically writes 1 to this register/field; used with 1-bit fields.

SC **Self clear.** On a specified event, hardware automatically writes 0 to this register/field; used with 1-bit and multi-bit fields.

SM **Self modify.** On a specified event, hardware automatically writes a specified value to this register/field; used with multi-bit fields.

SU **Self update.** On a specified event, hardware automatically updates this register/field; used with multi-bit fields.

RS **Read to set.** If user application reads from this register/field, hardware automatically writes 1 to it.

RC **Read to clear.** If user application reads from this register/field, hardware automatically writes 0 to it.

RF **Read from FIFO.** If user application writes new data to FIFO, the register/field automatically reads it.

WF **Write to FIFO.** If user application writes new data to this register/field, it automatically passes the data to FIFO via APB bus.

WS **Write any value to set.** If user application writes to this register/field, hardware automatically sets this register/field.

W1S **Write 1 to set.** If user application writes 1 to this register/field, hardware automatically sets this register/field.

W0S **Write 0 to set.** If user application writes 0 to this register/field, hardware automatically sets this register/field.

WC **Write any value to clear.** If user application writes to this register/field, hardware automatically clears this register/field.

W1C **Write 1 to clear.** If user application writes 1 to this register/field, hardware automatically clears this register/field.

W0C **Write 0 to clear.** If user application writes 0 to this register/field, hardware automatically clears this register/field.

WT **Write 1 to trigger an event.** If user application writes 1 to this field, this action triggers an event (pulse in the APB bus) or clears a corresponding WTC field (see WTC).

WTC **Write to clear.** Hardware automatically clears this field if user application writes 1 to the corresponding WT field (see WT).

W1T **Write 1 to toggle.** If user application writes 1 to this field, hardware automatically inverts the corresponding field; otherwise - no effect.

W0T **Write 0 to toggle.** If user application writes 0 to this field, hardware automatically inverts the corresponding field; otherwise - no effect.

WL **Write if a lock is deactivated.** If the lock is deactivated, user application can write to this register/field.

varies **The access type varies.** Different fields of this register might have different access types.

# Programming Reserved Register Field

## Introduction

A field in a register is reserved if the field is not open to users, or produces unpredictable results if configured to values other than defaults.

## Programming Reserved Register Field

The reserved fields should not be modified. It is not possible to write only part of a register since registers must always be written as a whole. As a result, to write an entire register that contains reserved fields, you can choose one of the following two options:

1. Read the value of the register, modify only the fields you want to configure and then write back the value so that reserved fields are untouched.

   OR

2. Modify only the fields you want to configure and write back the default value of the reserved fields. The default value of a field is provided in the "Reset" line of a register diagram. For example, the default value of Field_A in Register X is 1.

**Register 34.24. Register X (Address)**

| | (reserved) | Field_C | (reserved) | Field_B | Field_A | |
|---|---|---|---|---|---|---|
| | 31 … 20 | 19 … 16 | 15 … 2 | 1 | 0 | |
| | 0 0 0 0 0 0 0 0 0 0 0 0 | 0000 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | 0 | 1 | Reset |

Suppose you want to set Field_A, Field_B, and Field_C of Register X to 0x0, 0x1, and 0x2, you can:

- Use option 1 and fill in the reserved fields with the value you have just read. Suppose the register reads as 0x0000_0003. Then, you can modify the fields you want to configure, thus writing 0x0002_0002 to the register.
- Use option 2 and fill in the reserved fields with their defaults, thus writing 0x0002_0002 to the register.

# Interrupt Configuration Registers

Generally, the peripherals' internal interrupt sources can be configured by the following common set of registers:

- **RAW** (Raw Interrupt Status) register: This register indicates the raw interrupt status. Each bit in the register represents a specific internal interrupt source. When an interrupt source triggers, its RAW bit is set to 1.
- **ENA** (Enable) register: This register is used to enable or disable the internal interrupt sources. Each bit in the ENA register corresponds to an internal interrupt source.

  By manipulating the ENA register, you can mask or unmask individual internal interrupt source as needed. When an internal interrupt source is masked (disabled), it will not generate an interrupt signal, but its value can still be read from the RAW register.
- **ST** (Status) register: This register reflects the status of enabled interrupt sources. Each bit in the ST register corresponds to a specific internal interrupt source. The ST bit being 1 means that both the corresponding RAW bit and ENA bit are 1, indicating that the interrupt source is triggered and not masked. The other combinations of the RAW bit and ENA bit will result in the ST bit being 0.

  The configuration of ENA/RAW/ST registers is shown in Table 34.6-4.
- **CLR** (Clear) register: The CLR register is responsible for clearing the internal interrupt sources. Writing 1 to the corresponding bit in the CLR register clears the interrupt source.

**Table 34.6-4. Configuration of ENA/RAW/ST Registers**

| ENA Bit Value | RAW Bit Value | ST Bit Value |
|---|---|---|
| 0 | Ignored | 0 |
| 1 | 0 | 0 |
| | 1 | 1 |

# Revision History

| Date | Version | Release notes |
|---|---|---|
| 2025-05-08 | v1.3 | Updated the following chapters:<br>• Chapter 2 *GDMA Controller (GDMA)*: Added descriptions for the GDMA_OUTFIFO_OVF_CH*n*_INT, GDMA_OUTFIFO_UDF_CH*n*_INT, GDMA_INFIFO_OVF_CH*n*_INT, and GDMA_INFIFO_UDF_CH*n*_INT interrupts<br>• Chapter 34 *On-Chip Sensor and Analog Signal Processing*: Corrected "-0.5" to "+0.5" in the ADC filter formula |
| 2025-01-14 | v1.2 | Adjusted the order of chapters<br>Updated the following chapters:<br>• Chapter 1 *ESP-RISC-V CPU*: Fixed the bit position of mte and added clarification that the tcontrol register complies with the RISC-V External Debug Support Version 0.13.2<br>• Chapter 2 *GDMA Controller (GDMA)*: Updated the descriptions of suc_eof and the EOF flag<br>• Chapter 8 *Interrupt Matrix (INTERRUPT)*: Updated I2S1_INT to I2S_INT<br>• Chapter 9 *Low-power Management*: Updated the description of predefined low-power modes<br>• Chapter 16 *System Registers (SYSREG)*:<br>– Updated the DMA sources in the notes under Table 16.3-2 according to the GDMA chapter<br>– Removed description about I2S1<br>• Chapter 17 *Debug Assistant (ASSIST_DEBUG)*: Updated the DMA sources in Table 17.4-3 according to the GDMA chapter<br>• Chapter 26 *UART Controller (UART)*: Updated descriptions about clearing the wake_up signal<br>• Chapter 27 *SPI Controller (SPI)*: Updated the description of register SPI_DIN_MODE_REG and added the description to clk_hclk.<br>• Chapter 30 *USB Serial/JTAG Controller (USB_SERIAL_JTAG)*: Added a note in Section 30.4 stating that burning certain eFuse can affect the normal operation of USB Serial/JTAG controller.<br>• Chapter 32 *LED PWM Controller (LEDC)*: Updated the lowest resolution in Table 32.3-1<br>• Chapter 34 *On-Chip Sensor and Analog Signal Processing*: Removed descriptions about the internal voltage/signal in SAR ADC2 measurement |

Cont'd on next page

Cont'd from previous page

| Date | Version | Release notes |
|---|---|---|
| 2024-01-19 | v1.1 | Added Section *Programming Reserved Register Field* and Section *Interrupt Configuration Registers*<br>Updated register prefix APB_CTRL to SYSCON<br>Updated the following chapters:<br>• Chapter 5 *IO MUX and GPIO Matrix (GPIO, IO MUX)*: Updated the description in Section 5.9, and deleted the GPIO_PCPU_NMI_INT_REG register and related information<br>• Chapter 7 *Chip Boot Control*: Added SPI Download Boot mode, renamed Download Boot mode to Joint Download mode in Section 7.2, and provided more details about how FUSE_DIS_FORCE_DOWNLOAD and EFUSE_DIS_DOWNLOAD_MODE control chip boot mode<br>• Chapter 8 *Interrupt Matrix (INTERRUPT)*: Deleted the INTERRUPT_CORE0_GPIO_INTERRUPT_PRO_NMI_MAP_REG register and related information<br>• Chapter 9 *Low-power Management*: Updated the description of register RTC_CNTL_WDT_WKEY<br>• Chapter 11 *Timer Group (TIMG)*: Updated the description of TIMG_WDT_CLK_PRESCALE<br>• Chapter 14 *Permission Control (PMS)*: Removed ROM_Table related description<br>• Chapter 26 *UART Controller (UART)*: Updated Figure 26.3-1 *UART Architecture Overview* and the number of rising edges required to generate the wake_up signal<br>• Chapter 28 *I2C Controller (I2C)*: Updated I2C timeout configuration and the corresponding descriptions of I2C_TIME_OUT_VALUE, I2C_COMD0_REG, I2C_SDA_FORCE_OUT and I2C_SCL_FORCE_OUT |

Cont'd on next page

Cont'd from previous page

| Date | Version | Release notes |
|---|---|---|
| 2023-05-19 | v1.0 | Added Chapter 14 *Permission Control (PMS)*<br>Updated the following chapters:<br>• Chapter 2 *GDMA Controller (GDMA)*: Updated the descriptions of the GDMA_IN_SUC_EOF_CH*n*_INT interrupt and the GDMA_INLINK_DSCR_ADDR_CH*n* field<br>• Chapter 3 *System and Memory*: Updated Table 3.3-3<br>• Chapter 4 *eFuse Controller (EFUSE)*: Added a note on programming the XTS-AES key<br>• Chapter 9 *Low-power Management*: Removed UART as a reject to sleep cause, and added more description about register RTC_CNTL_GPIO_WAKEUP_REG<br>• Chapter 11 *Timer Group (TIMG)*: Updated the procedures to read the timer's value<br>• Chapter 12 *Watchdog Timers (WDT)*: Removed ULP-RISC-V references<br>• Chapter 26 *UART Controller (UART)*: Added descriptions about the break condition, and updated Figure UART Architecture Overview, Figure UART Structure, and Figure Hardware Flow Control Diagram , and updated the maximum length of stop bits and related descriptions<br>• Chapter 30 *USB Serial/JTAG Controller (USB_SERIAL_JTAG)*: Added the specific pull-up values configured by the USB_SERIAL_JTAG_PULLUP_VALUE bit<br>• Chapter 32 *LED PWM Controller (LEDC)*: Added the formula to calculate duty cycle resolution and Table *Commonly-used Frequencies and Resolutions* |

Cont'd on next page

Cont'd from previous page

| Date | Version | Release notes |
|---|---|---|
| 2022-12-16 | v0.7 | Added the following chapter:<br>• Chapter 15 *World Controller (WCL)*<br>Updated the following chapters:<br>• Chapter 1 *ESP-RISC-V CPU*<br>• Chapter 3 *System and Memory*<br>• Chapter 4 *eFuse Controller (EFUSE)*<br>• Chapter 9 *Low-power Management*<br>• Chapter 17 *Debug Assistant (ASSIST_DEBUG)*<br>• Chapter 23 *External Memory Encryption and Decryption (XTS_AES)*<br>• Chapter 24 *Random Number Generator (RNG)*<br>• Chapter 29 *I2S Controller (I2S)*<br>• Chapter 34 *On-Chip Sensor and Analog Signal Processing*<br>Updated clock names:<br>• FOSC_CLK: renamed as RC_FAST_CLK<br>• FOSC_DIV_CLK: renamed as RC_FAST_DIV_CLK<br>• RTC_CLK: renamed as RC_SLOW_CLK<br>• SLOW_CLK: renamed as RTC_SLOW_CLK<br>• FAST_CLK: renamed as RTC_FAST_CLK<br>• PLL_160M_CLK: renamed as PLL_F160M_CLK<br>• PLL_240M_CLK: renamed as PLL_D2_CLK<br>Updated the Glossary section |
| 2022-02-16 | v0.6 | Added the following chapters:<br>• Chapter 27 *SPI Controller (SPI)*<br>• Chapter 29 *I2S Controller (I2S)* |
| 2022-01-12 | v0.5 | Added the following chapters:<br>• Chapter 9 *Low-power Management*<br>• Chapter 23 *External Memory Encryption and Decryption (XTS_AES)*<br>Updated the following Chapters:<br>• Chapter 1 *ESP-RISC-V CPU*, Section 1.4.1 by adding three GPIO Access CSRs; Section 1.5 by removing the list of CPU interrupt registers and providing redirection to Chapter 8 *Interrupt Matrix (INTERRUPT)*<br>• Chapter 3 *System and Memory*<br>• Chapter 4 *eFuse Controller (EFUSE)*<br>• Chapter 19 *HMAC Accelerator (HMAC)*<br>• Chapter 20 *RSA Accelerator (RSA)*<br>• Chapter 22 *Digital Signature (DS)* |

Cont'd on next page

Cont'd from previous page

| Date | Version | Release notes |
|---|---|---|
| 2021-10-28 | v0.4 | Added the following chapters:<br>• Chapter 8 *Interrupt Matrix (INTERRUPT)*<br>• Chapter 17 *Debug Assistant (ASSIST_DEBUG)*<br>• Chapter 28 *I2C Controller (I2C)*<br>• Chapter 34 *On-Chip Sensor and Analog Signal Processing*<br>• Chapter VII<br>Updated the following Chapters:<br>• Chapter 4 *eFuse Controller (EFUSE)*<br>• Chapter 33 *Remote Control Peripheral (RMT)* |
| 2021-08-05 | v0.3 | Added the following chapters:<br>• Chapter 10 *System Timer (SYSTIMER)*<br>• Chapter 12 *Watchdog Timers (WDT)*<br>• Chapter 13 *XTAL32K Watchdog Timers (XTWDT)*<br>• Chapter 16 *System Registers (SYSREG)*<br>• Chapter 19 *HMAC Accelerator (HMAC)*<br>• Chapter 22 *Digital Signature (DS)*<br>• Chapter 30 *USB Serial/JTAG Controller (USB_SERIAL_JTAG)*<br>• Chapter 33 *Remote Control Peripheral (RMT)*<br>Updated the following Chapters:<br>• Chapter 4 *eFuse Controller (EFUSE)*<br>• Chapter 5 *IO MUX and GPIO Matrix (GPIO, IO MUX)*<br>• Chapter 7 *Chip Boot Control*<br>• Chapter 31 *Two-wire Automotive Interface (TWAI)* |
| 2021-05-27 | v0.2 | Added the following chapters:<br>• Chapter 2 *GDMA Controller (GDMA)*<br>• Chapter 4 *eFuse Controller (EFUSE)*<br>• Chapter 11 *Timer Group (TIMG)*<br>• Chapter 26 *UART Controller (UART)*<br>• Chapter 32 *LED PWM Controller (LEDC)*<br>Updated the Chapter 5 *IO MUX and GPIO Matrix (GPIO, IO MUX)* Adjusted the order of chapters |
| 2021-04-08 | v0.1 | Preliminary release |

## Disclaimer and Copyright Notice

Information in this document, including URL references, is subject to change without notice.

ALL THIRD PARTY'S INFORMATION IN THIS DOCUMENT IS PROVIDED AS IS WITH NO WARRANTIES TO ITS AUTHENTICITY AND ACCURACY.

NO WARRANTY IS PROVIDED TO THIS DOCUMENT FOR ITS MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, NOR DOES ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

All liability, including liability for infringement of any proprietary rights, relating to 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 © 2025 Espressif Systems (Shanghai) Co., Ltd. All rights reserved.**

www.espressif.com